

C#.NET 2: C# Programming in the .NET Framework

Lesson 1: **Introduction**

[Understanding the Learning Sandbox Environment](#)

[The OST Plug-In](#)

[Let's Do Something!](#)

[Closing and Reopening a Solution](#)

[Studio IDE](#)

[A Quick Tour](#)

[Creating Your Second Project](#)

[Programming Code](#)

[The Source Code Window](#)

Lesson 2: **Statements, Keywords, Scope, Types, Variables, and Conversions**

[Programming Statements](#)

[Identifiers and Reserved Words](#)

[Scope](#)

[Scope Definition](#)

[Local Scope](#)

[Block Scope](#)

[Class Scope](#)

[Scope Importance](#)

[Data Types](#)

[Built-In Types](#)

Lesson 3: **Expressions, Precedence, and Assignment, and Increment/Decrement Operators**

[Programming Expressions](#)

[Assignments and Assignment Shortcuts](#)

[Simple Assignment Operator](#)

[Compound Assignment Operators](#)

[Mathematical Compound Assignment Operators](#)

[Bitwise and Shift Compound Assignment Operators](#)

[C# Operators and Precedence](#)

[Increment and Decrement Operators](#)

Lesson 4: **Relational, Logical, and Conditional Operators**

[Relational Operators](#)

[Boolean and Bitwise Logical Operators](#)

[Body Fat Calculator: Using Relational and Boolean Logical Operators](#)

[Conditional Operator](#)

Lesson 5: **Branching Statements: Conditional and Unconditional**

[Unconditional Branching](#)

[Conditional Branching](#)

Lesson 6: **Iteration (Looping)**

[The For Loop](#)

[The While Loop](#)

[The Do..While Loop](#)

[For Loop Versus While Loop](#)

[Putting Looping To Work](#)

Lesson 7: **Input and Output Statements**

[Windows GUI Application Versus Console Application](#)

[Simple Console Input and Output](#)

[GUI Input and Output](#)

Lesson 8: **File Input and Output**

[Using File Object](#)

[Using Streams](#)

Lesson 9: **Models, Classes, and Objects**

[Object-Oriented Programming Constructs](#)

[Models: A World View](#)

[Classes: A Blueprint and More](#)

[Example: The Alphabet Aquarium](#)

[Classes: A New Data Type](#)

[Objects: A Multiplicity of Instances](#)

Lesson 10: **Defining Classes and Class Relationships**

[Conceptual Versus Design](#)

[Defining a Class](#)

[Defining Relationships](#)

[Visibility and Cardinality](#)

Lesson 11: **Methods, Return Types, and Constructors**

[Methods](#)

[Class Function Members](#)

[Class Methods](#)

[Access Modifiers](#)

[Method Parameters](#)

[Method Return Types](#)

[Coding: Camel I](#)

[The ref Keyword and Passing By Value Versus Passing By Reference](#)

[Class Constructors](#)

Lesson 12: **Initializers, Object Initializers, and Anonymous Types**

[Types of Data Types](#)

[Value Data Types](#)

[Reference Data Types](#)

[Boxing and Unboxing](#)

[The new Keyword](#)

[Object Initialization](#)

[Constructors and Object Initialization](#)

[Accessors and Object Initialization](#)

[Object Initializers](#)

[Automatic Properties](#)

[Coding: The Employee Database](#)

[Anonymous Types](#)

[Popup Forms and Dialogs](#)

Lesson 13: **The this Keyword, Static and Instance Members**

[Class Instances](#)

[Instances](#)

[Self-Referencing Using the 'this' Keyword](#)

[Coding: Boxing](#)

[Static Members and Classes](#)

[Static Keyword](#)

[Class Instances Versus Static Instances](#)

Lesson 14: **Finalizing Objects, and Memory Concepts (Stack versus Heap)**

[Computer Memory](#)

[Computer Memory and Allocation](#)

[Stack Versus Heap](#)

[Garbage Collection](#)

[Finalizing Objects and Destructors](#)

[Coding](#)

Lesson 15: **Overloading Methods, Overloading Operators, and Returning Multiple Values Using out**

[Overloading](#)

[Method Overloading](#)

[Operator Overloading](#)

[Coding](#)

[The out Keyword](#)

[Parameter Modifier Review](#)

[The out Parameter Modifier](#)

[Method Fingerprint/Signature and Parameter Modifiers](#)

[Multiple Return Values Using out](#)

[Coding](#)

Lesson 16: **Debugging**

[A Sample Debugging Program](#)

[Breakpoints, Expression Evaluation, and Watches](#)

Lesson 17: **One-Dimensional Arrays, the foreach Keyword, Initializing, and the params Keyword**

[Array Fundamentals](#)

[What Is An Array?](#)

[Array Declaration, Allocation, and Initialization](#)

[Accessing Array Elements](#)

[Coding](#)

[Advanced Array Topics](#)

[Array Assignments](#)

[Array Iteration and the foreach Statement](#)

[The params Parameter Modifier](#)

[Anonymous Arrays](#)

[Course Project](#)

[The Assignment](#)

[The Materials](#)

[Coding Requirements](#)

[The Milestones](#)

[Testing](#)

[Hints](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Introduction

Welcome to the O'Reilly School of Technology **C#.NET 2: C# Programming in the .NET Framework** course!

Course Objectives

When you complete this course, you will be able to:

- program C# statements, identifiers, expressions, and assignments.
- apply relational, boolean, and bitwise logical operators.
- branch and iterate C# code.
- use Windows GUI and Console applications.
- define object-oriented models, initializers, and class relationships.
- demonstrate knowledge of overloading and memory concepts such as stack and heap.
- debug programs using breakpoints, expression evaluation, and watches.

In this course you will gain a deeper understanding of object oriented programming. You will learn about data types and scope, and create programs using models, classes, objects, methods, and constructors, as well as the ways in which these elements relate to one another. File Input/Output (I/O) topics are also covered, allowing you to both read from and write to files. As you create increasingly complex projects, you will learn ways to debug your code as well.

You will create several applications throughout the course, which will enhance your professional portfolio and also contribute toward certificate completion.

Note

This lesson duplicates the first two lessons of the first C# course. If you've taken that course, you may either skip this lesson, or browse through it to refresh your knowledge of the material. However, you'll still complete the homework to demonstrate that you know the material—and to get your certificate for this course!

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add *looks like this*.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type *look like this*.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

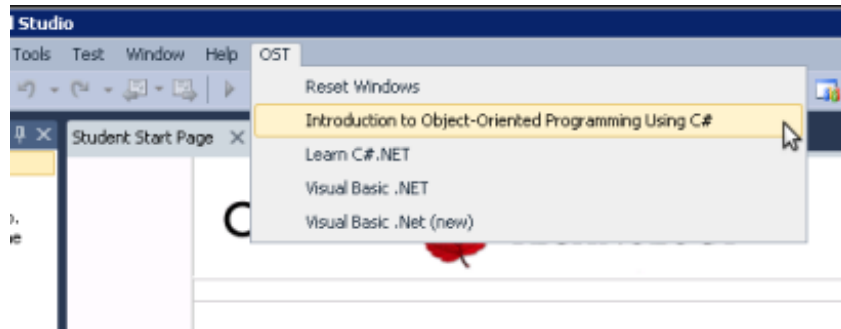
Understanding the Learning Sandbox Environment

We'll be doing lots of work in Visual Studio, Microsoft's Integrated Development Environment (IDE) for working with C#. In the next lesson, we will cover in-depth how to use Visual Studio. For this first lesson, we'll introduce you to the visual cues to help you learn and experiment, and then we'll help you create your first C# application!

 When you see this icon, you should save your work. You can save whenever you like, and we strongly encourage you to get in the habit of saving your projects frequently. Whenever you pause to think, let your mouse pointer find  on the Toolbar! It's a great habit to get into!

The OST Plug-In

We've added a menu item to the Visual Studio system for your convenience. Use the **OST** menu to get to your syllabus for this course at any time. Your menu may have different courses, or only this course, listed.



Tip

This may serve as a "panic button," too; if your Visual Studio menus start to get confusing in the course of your work, you can always get back to the default view by selecting this course from the OST menu.

Let's Do Something!

Okay, enough talk; let's create our first C# program! Traditionally, a first program is one that simply displays the text "Hello, World!" so we'll walk you through the steps. The goal is to make Studio do something NOW! We'll learn more about the Studio development environment in the next lesson. We'll move quickly through making this first program; here's an overview of the steps:

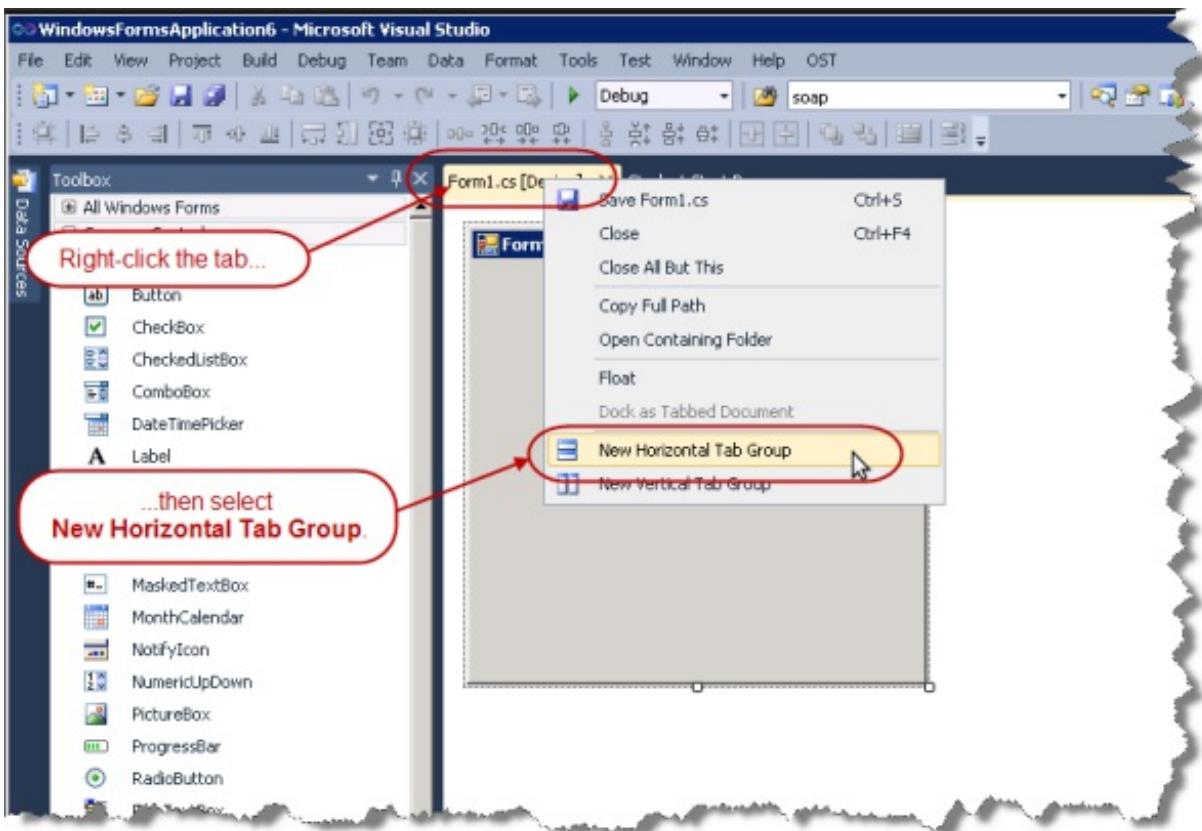
1. Create a new C# Windows Form Application project called HelloWorld.
2. Open and "pin" the Toolbox.
3. Drag a Label control from the Toolbox onto the Windows Form.
4. Open the Properties Window.
5. Set the display Text for the Label control to "Hello, World!"
6. Save the HelloWorld project.
7. Run the HelloWorld project.

Let's get started!

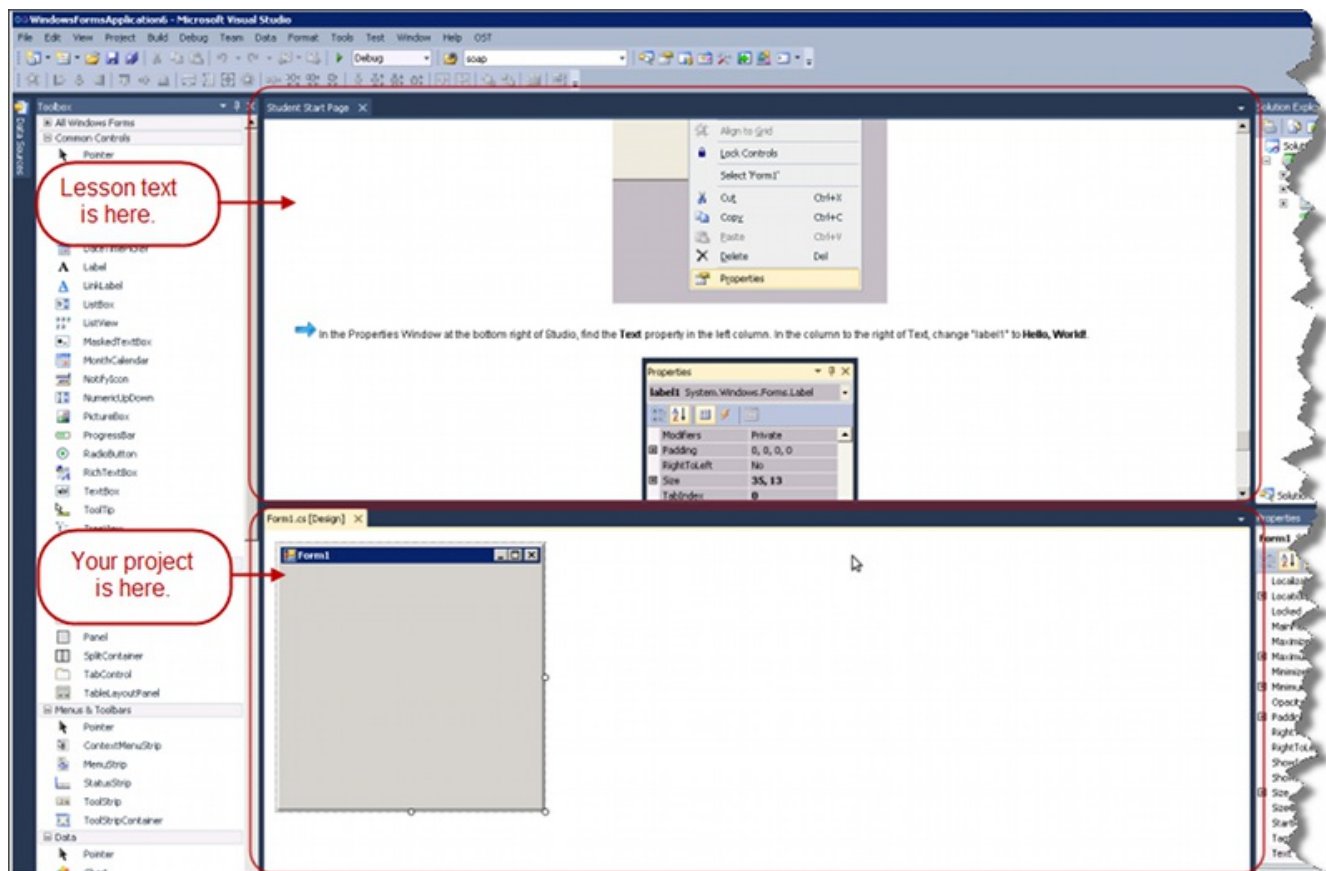
Note

After you create the project below, you won't be able to see this lesson text. To get it back, right-click on the tab marked **Form.cs[Design]** and select **New Horizontal Tab Group**. This will split the screen with the lesson page at the top and the form designer at the bottom. You will need to do this each time you create a new project. If more than one project is open, you can just drag its tab to the lower tab group to move it.

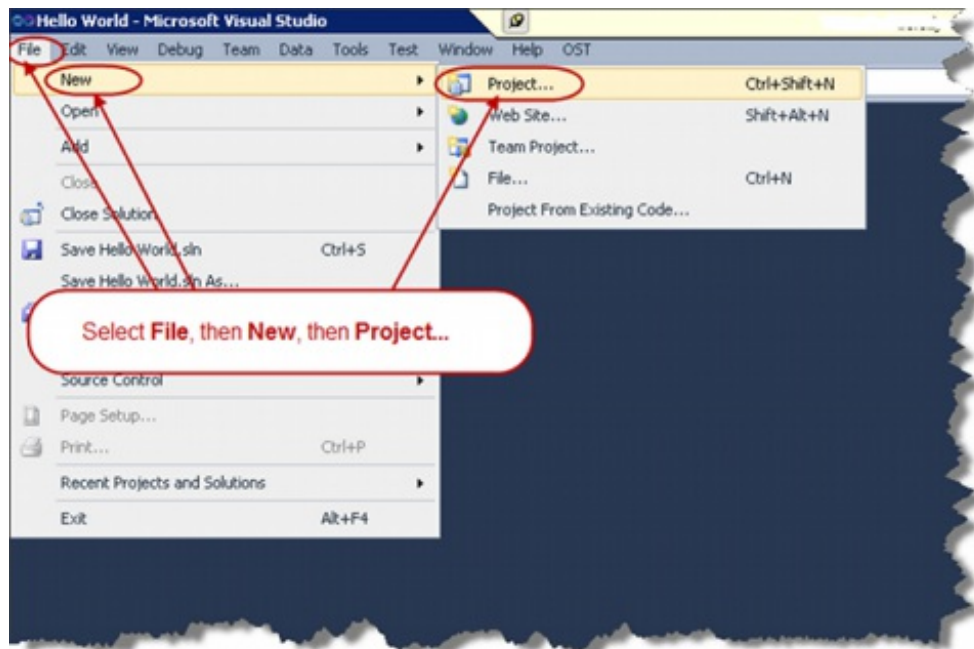
Creating a New Horizontal Tab Group



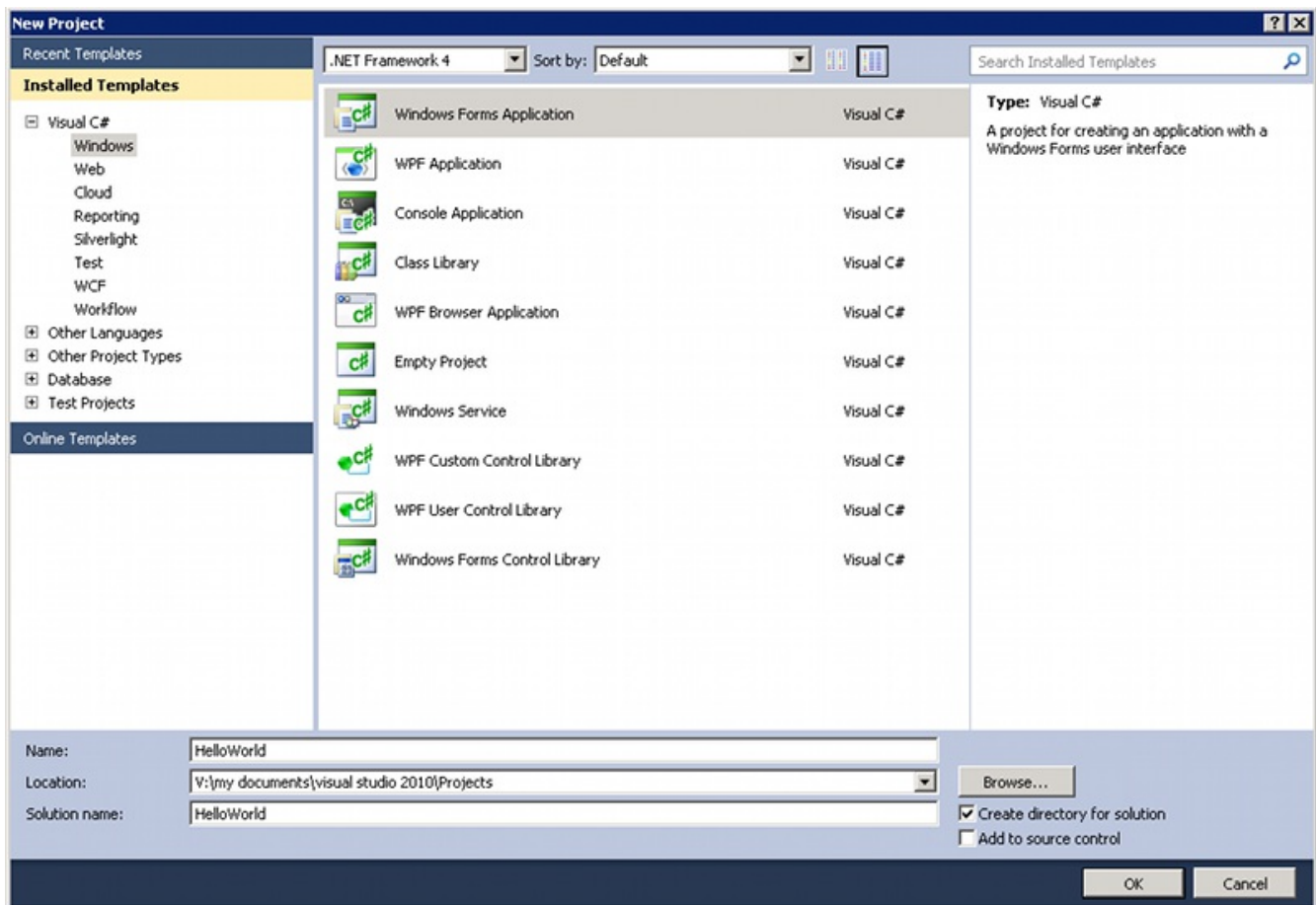
When you finish, you should see something like this.



Now, remembering those steps, let's create our project. Select the **File** menu, then **New**, then **Project** (in the future, we'll show this kind of menu sequence like this: **File | New | Project**):

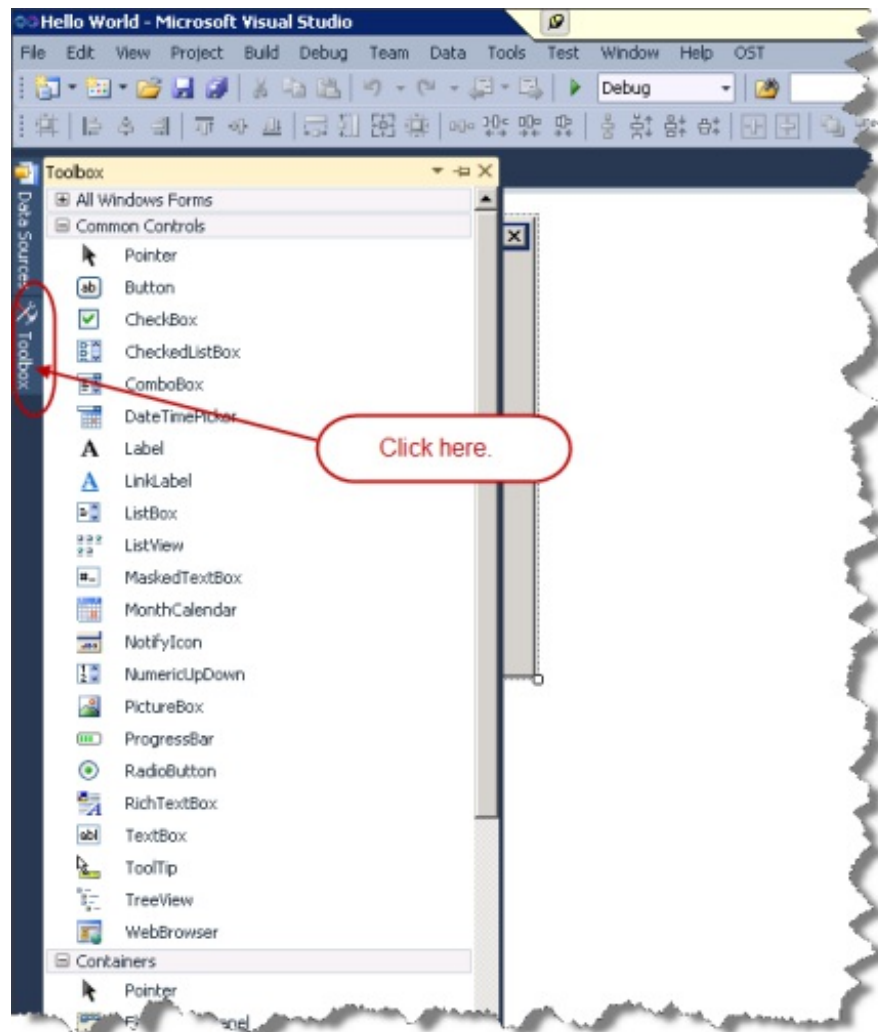


In the New Project dialog, in the left column, check that **Visual C# | Windows** is selected. In the middle column, make sure **Windows Forms Application** is selected (these options should already be selected by default). At the bottom of the dialog, in the Name textbox, replace **WindowsFormsApplication1** with **HelloWorld** or **Hello World** (you can include spaces in project names to make them more readable, if you like). Click **OK** when you finish typing the name.

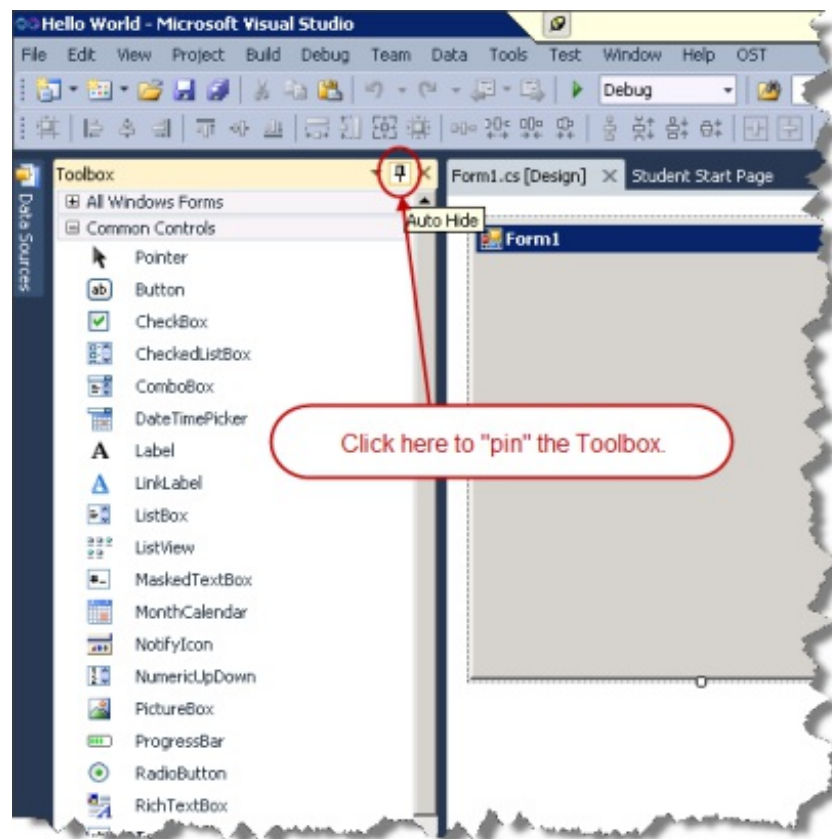


At this point, you'll need to perform the steps described earlier to get the lesson text back.

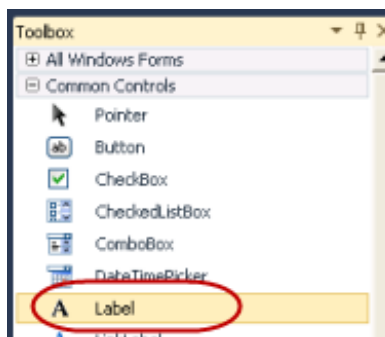
Now, we want to have our tools available all the time, so click **Toolbox**:



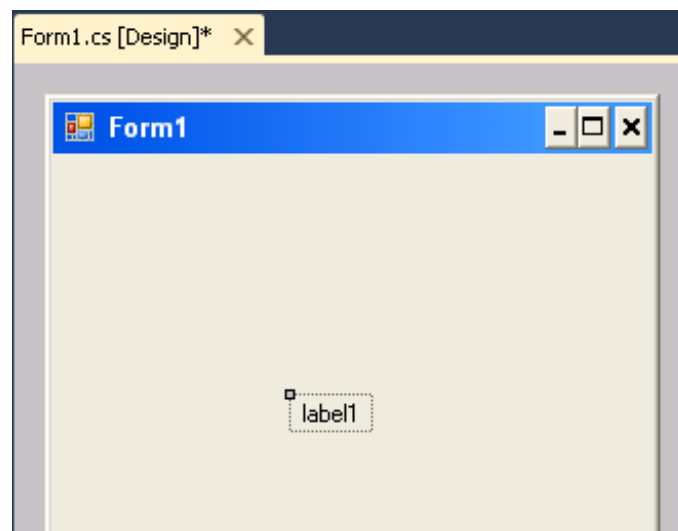
When the Toolbox expands, click on the **Auto Hide** "pin" icon at the top of the Toolbox window so that the pin is pointing down rather than horizontal. The Toolbox will now stay displayed.



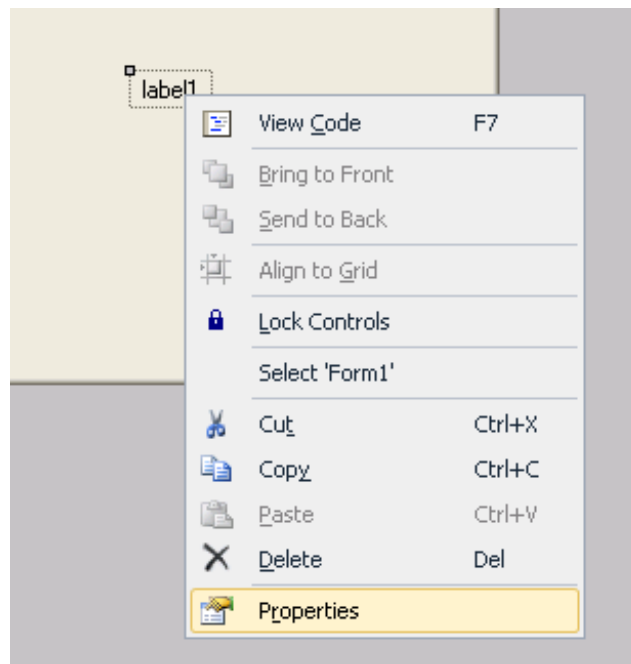
Find the **Label** control under the Common Controls option.



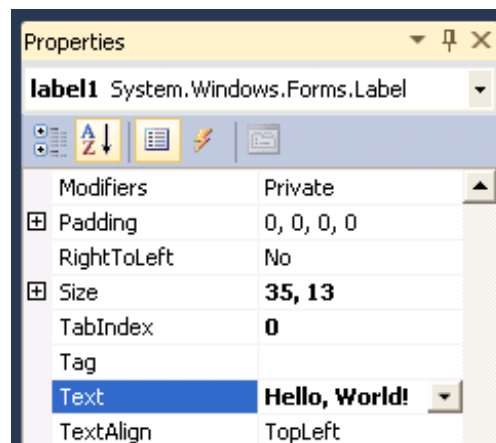
Click and drag the **Label** onto the grey area under the box named Form1 in the middle of Visual Studio.



Right-click on the "label1" text on Form1, and choose **Properties**.

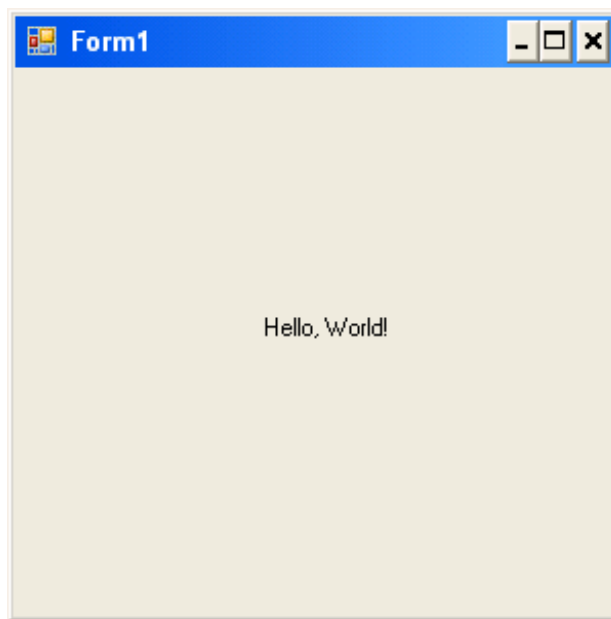


In the Properties Window at the bottom right of Studio, find the **Text** property in the left column. In the column to the right of Text, change "label1" to **Hello, World!**.



 Click **Save** in the Studio Toolbar (at top) to save the project.

Find the green **Start Debugging** button  in the Studio Toolbar. Click it to run the program!



Click the small black "x" or **Close** icon  to close your first Windows C# application.

Congratulations!!! You've just created and run your first C# application!!!

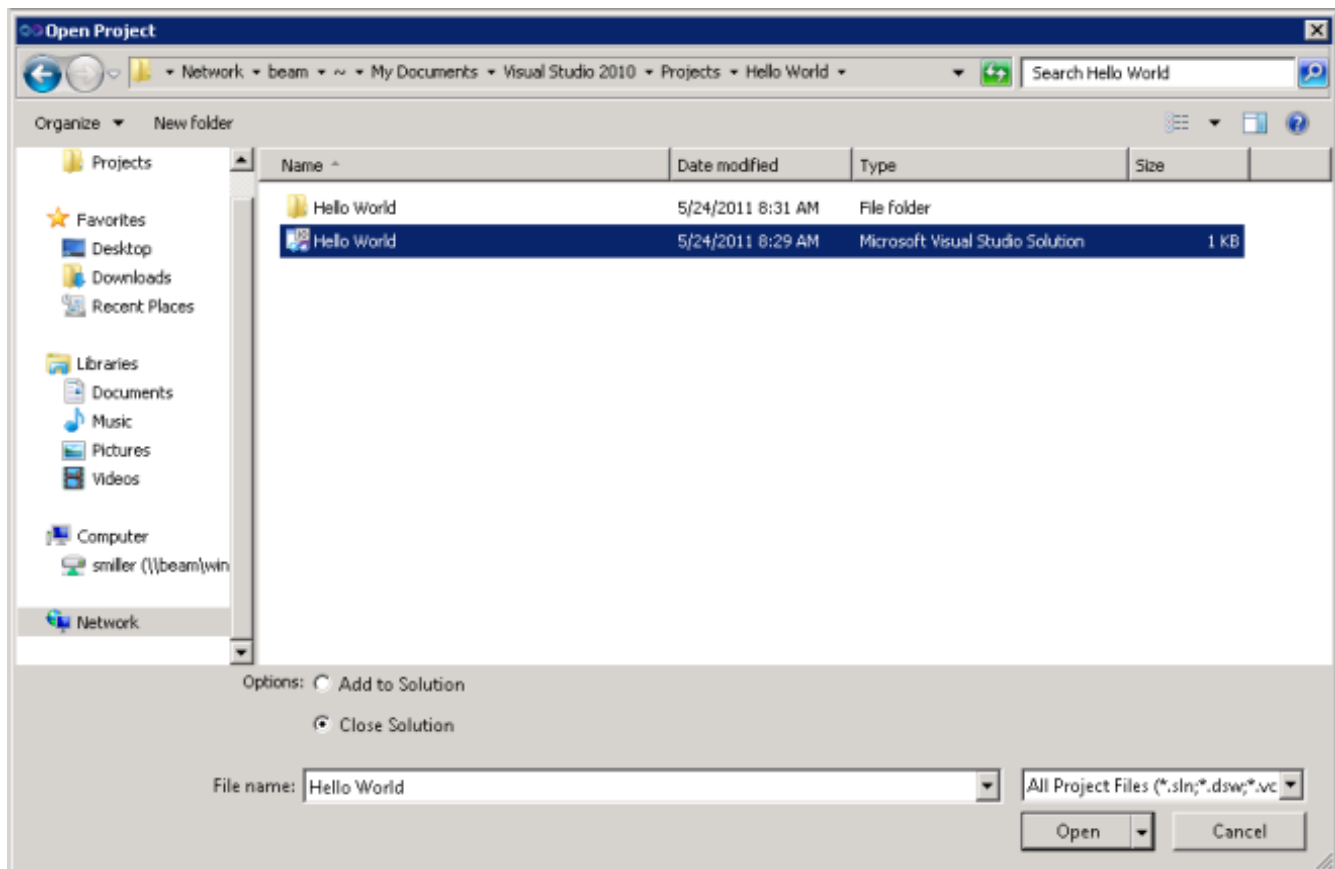
Closing and Reopening a Solution

Eventually, of course, you will love programming so much you'll never want to stop! Still, you might need to take a break from time to time, so you need to know how to close a solution and to reopen it when you return.

With your Hello World solution selected in the Solution Explorer, select **File | Close Solution**. The Solution Explorer will now be empty.

To reopen the solution, select **File | Open | Project/Solution...**

In the dialog that appears, double-click your **HelloWorld** (or **Hello World**) project folder, and then select your Hello World solution.



Double-click **Form1.cs** to bring back the form designer (and the tools in the Toolbox).

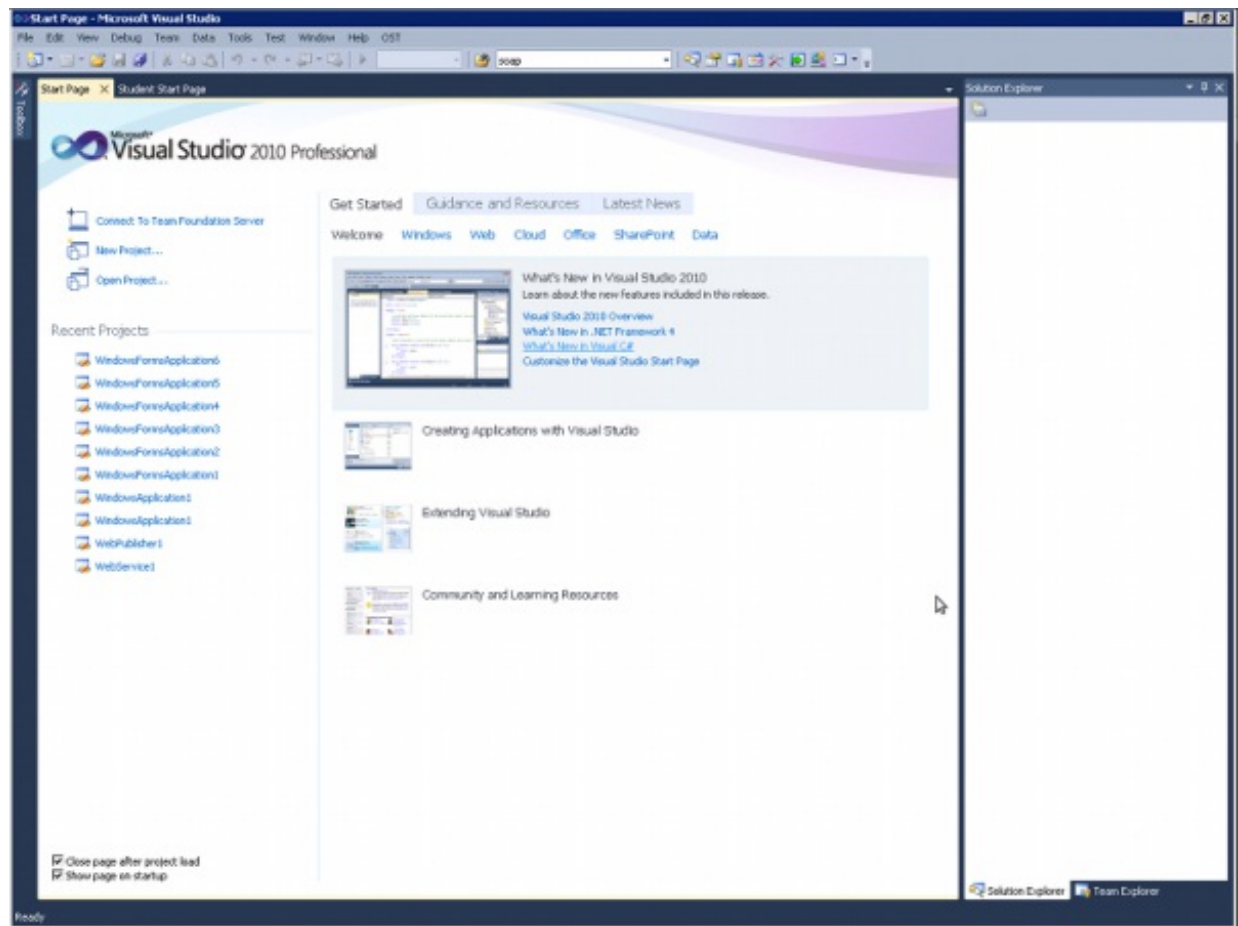
Studio IDE

Software development in .NET uses the Microsoft Visual Studio Integrated Development Environment (IDE). During this course, we will refer to Visual Studio as the Studio IDE, Studio, or IDE.

An IDE supplies most if not all of the tools you will need to develop software. This lesson will familiarize you with the features of the Studio IDE, and show how to use those features and components to begin creating your own software.

A Quick Tour

The Studio IDE initial Start Page looks like this (a popup dialog from the System Tray may alert you to an optional Customer Experience Improvement Program feature. You may participate or not by clicking on the popup dialog):



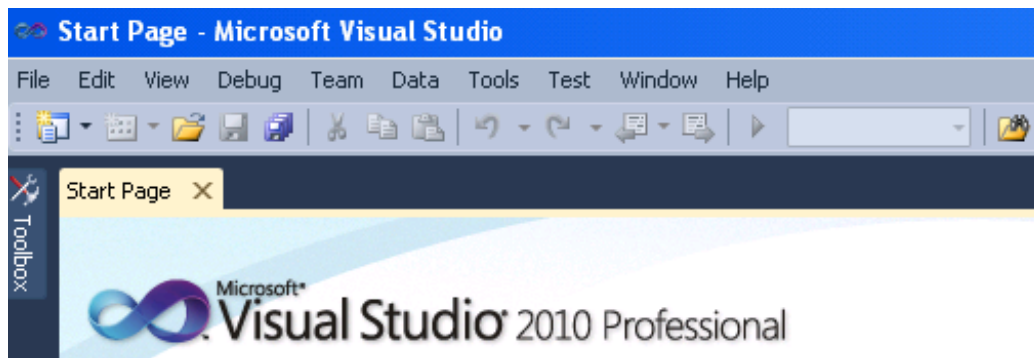
The Start Page is a convenient presentation of many initial tasks you may want to perform when first opening Studio.

Close this page by clicking on the small **x** on the tab labeled Start Page.

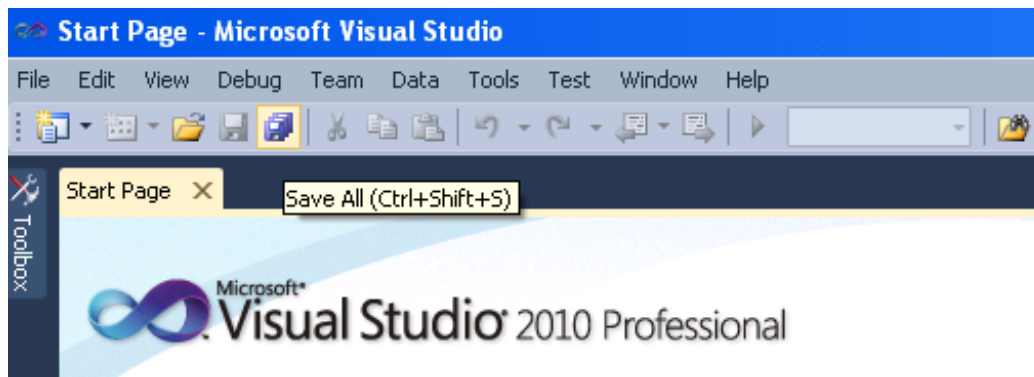
Start Page X

Tip To view the Start Page again at any time, select **View | Start Page**.

The Studio IDE includes a File menu along the top of the application, buttons just below the menu, and components below the buttons. All Studio IDE elements are context sensitive, which means that certain elements may be disabled or unavailable depending on what you are doing. You will notice that many of the buttons look dim, or gray. These buttons are disabled:



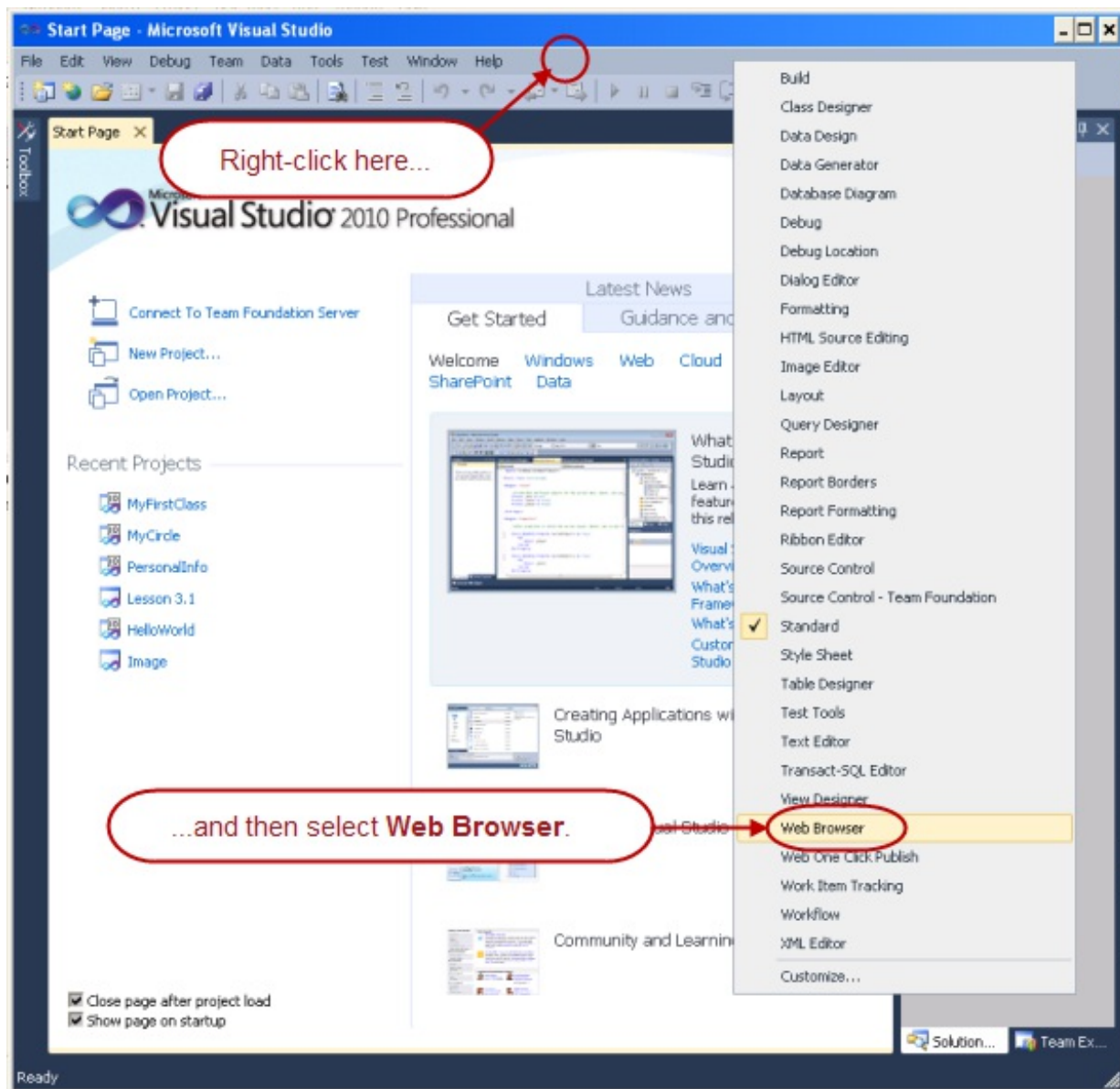
Move your mouse pointer over one of the icons and pause until a popup appears; here, for example, is the Save All icon popup:

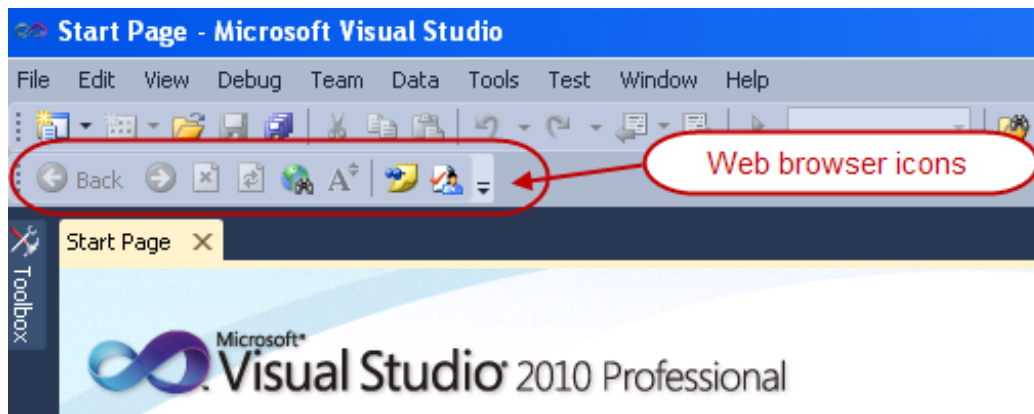


These popups display the icon's function (and keyboard shortcut, if available). The icon function corresponds to the same functionality available through the menu. For example, the Save All functionality is also available from the File menu.

The Studio IDE contains many more elements than initially appear, and as you work more with Studio, you will learn to add the elements that you prefer.

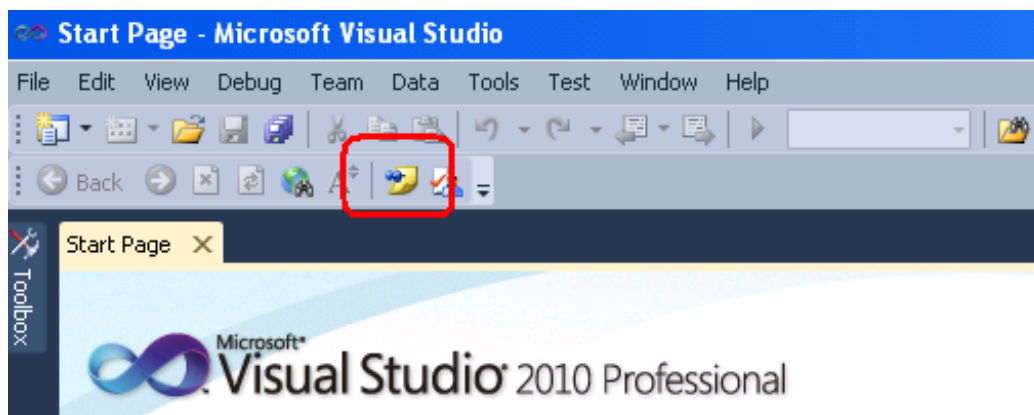
Right-click in the blank area at the top of Studio next to the menu items or buttons, and select **Web Browser**. Icons appropriate to a web browser will now be visible:



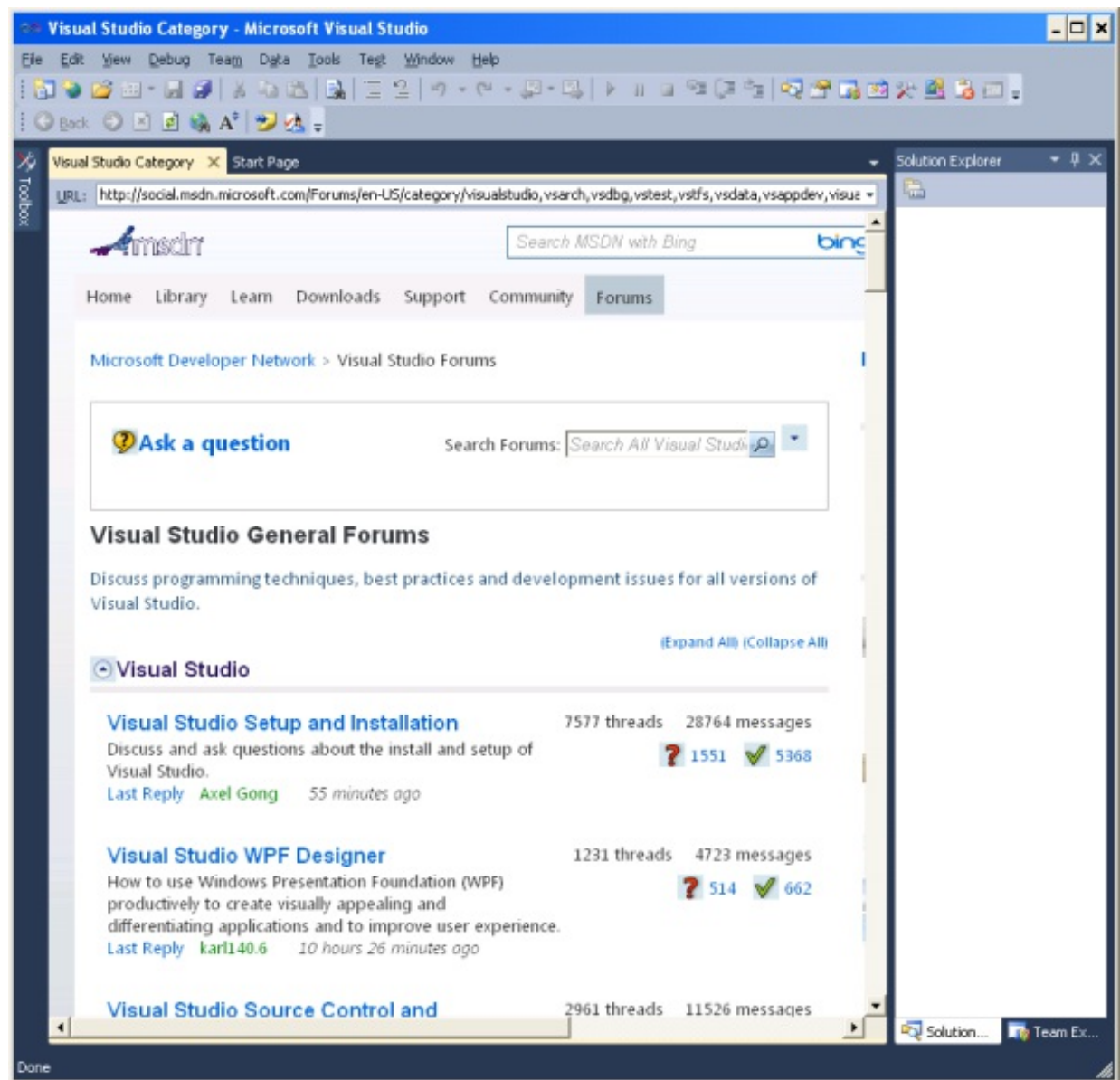


Note The Studio IDE contains an embedded web browser for viewing online or offline content.

To see the Microsoft Developer Network Forums home page, click the **MSDN Forums** icon:



You will see content similar to this:



This is a valuable resource for learning more about developing software using Visual Studio—we'll visit it again later.

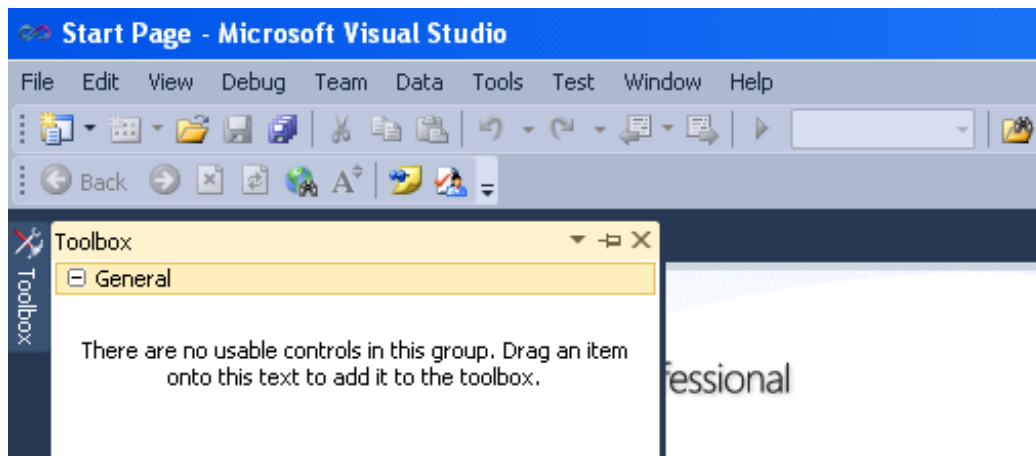
Close the Forums tab by clicking the  in the Visual Studio Category tab header.

Tip

If you lose this lesson window, or otherwise get confused by the rearranging of windows that Visual Studio may do, select **OST | Reset Windows** from the top menu.

At the top left corner of the Studio window, just below the icons, you will see the Toolbox panel, which we "pinned" in the last lesson.

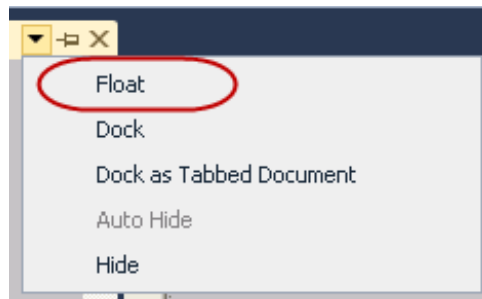
If it's not already open, click the word **Toolbox** to expand the button into a window:



You'll also see an expanded area to the right of the Studio window labeled Solution Explorer.

Both the Toolbox and Solution Explorer panels contain the same three controls at the top right:   . They are labeled **Window Position** (the arrowhead image), **Auto Hide** (the pushpin image), and **Close** (the X). Window Position controls the location and appearance of the panel. Auto Hide, if selected (the pushpin being horizontal) controls the display of the panel as your mouse moves over its icon. Close will close and hide the panel.

Click the Toolbox **Window Position** and select **Float**:



Now you can move the floating Toolbox panel around the screen by clicking and dragging it.

Tip If you double-click on the top of the panel (the title bar), the panel will maximize. To restore it to its original size, double-click the title bar again.

Click the Toolbox **Window Position** button again and select **Dock as Tabbed Document**. Note that the three panel controls    are gone. To restore the Toolbox to floating, you need to select the **Window | Float** menu option.

To restore the Toolbox as a docked element on the left of the screen, select the Toolbox **Window Position** button again, and select **Dock**.

Tip If you close or hide the Toolbox button or window, you can unhide it by selecting **View | Toolbox**.

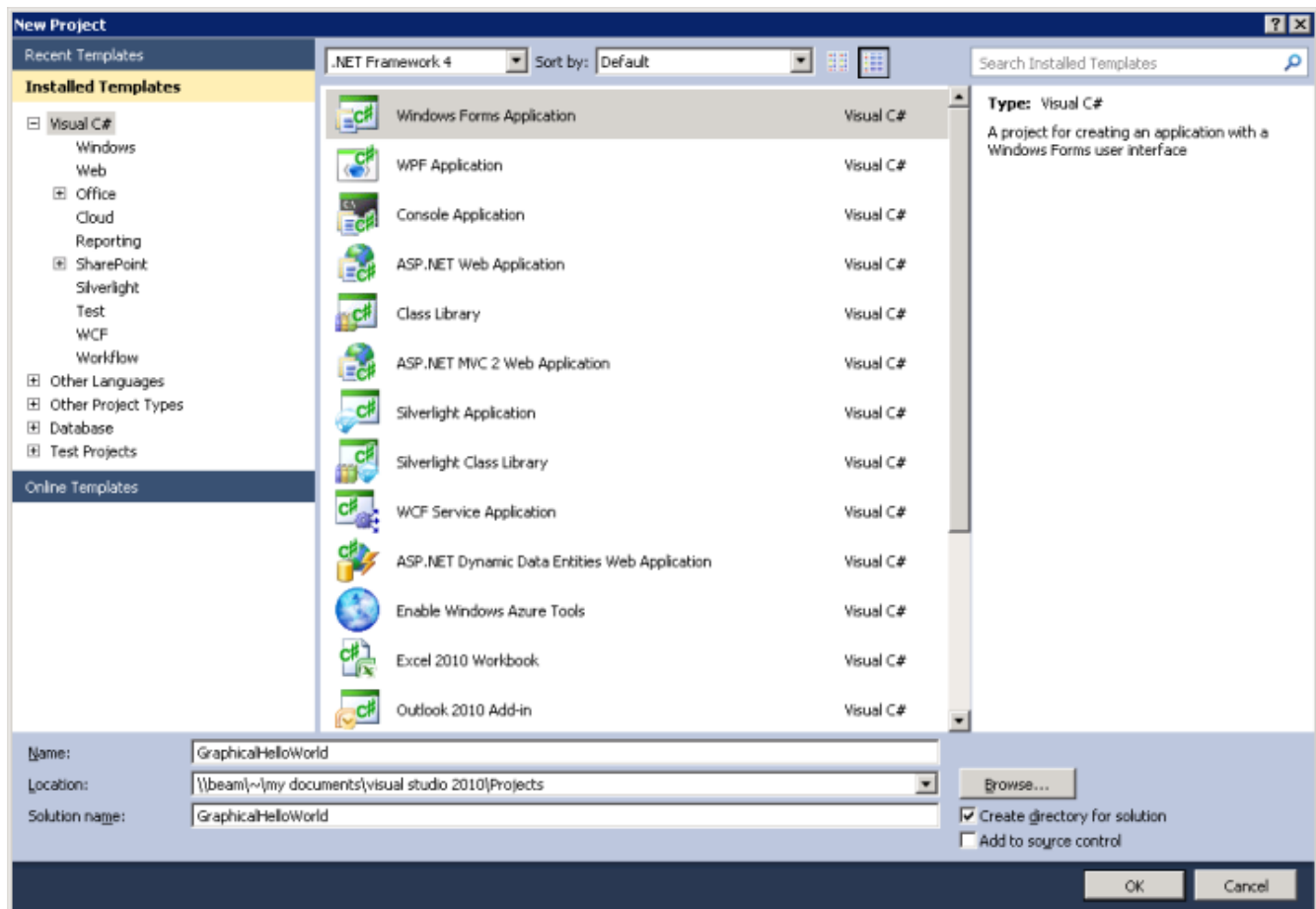
Toggle the Auto Hide pushpin and note the behavior when the pushpin is vertical versus when it is horizontal. When you finish, make sure it is vertical so the Toolbox remains onscreen.

Creating Your Second Project

Creating software is an interactive process, and experimenting is strongly encouraged! Feel free to experiment more with the Studio IDE.

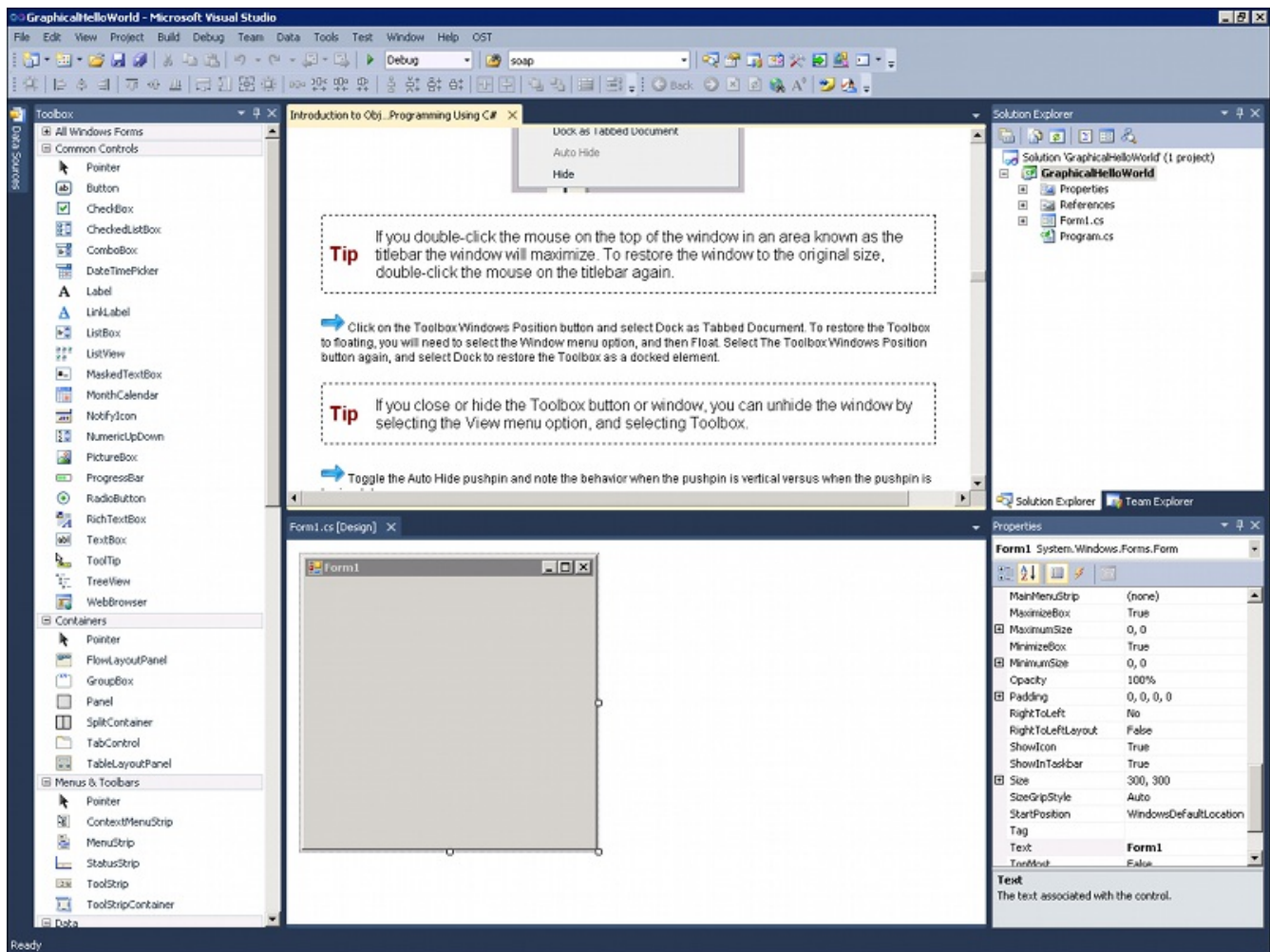
Now that you have an idea of how the buttons work, and how to navigate around a few of the Studio elements, let's create another C# project. As we learned in the first lesson, software developers often create a simple program that displays the words "Hello World." Let's create this traditional first program again, but with a twist.

Select **File | New | Project**. The New Project dialog presents a rather daunting list of possible project types; just select **Windows Forms Application**. Then, change the default project name to **GraphicalHelloWorld**:



Note Remember to right-click the Form1.cs tab and select **New Horizontal Tab Group** so you can see this lesson text again!

You now see the Studio environment including the GraphicalHelloWorld project. Note the Form1 form. This form represents the visual element of the program we will create. A Windows Form Application uses the concept of a rectangular form for presenting information to the user. You will also notice that the Solution Explorer now contains information.

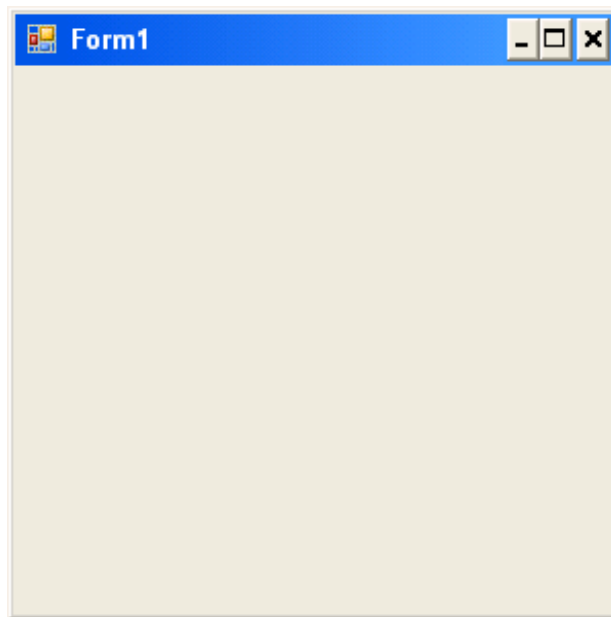


The Form1 component is a Visual Designer known as the Windows Forms Designer. Studio contains many Visual Designers that we will use throughout these lessons.

The Solution Explorer lists information in a hierarchical display related to solutions and projects. A project represents a single item, such as the GraphicalHelloWorld application, but a solution contains one or more projects. Every project belongs to a solution by default. You will use the Solution Explorer to add or access items within a project or solution, as well as to configure build settings, publish projects and solutions, and perform other activities.

Applications typically contain lines of programming code that run, or execute. When you run an application on your computer, you are running a published version of that application. As an application developer, you can run both a published version, and a development, or debug, version.

Locate the green play button  labeled **Start Debugging**. You may also press **F5**, or select **Debug | Start Debugging** from the menu. The same form that you saw in Studio appears as a separate window:



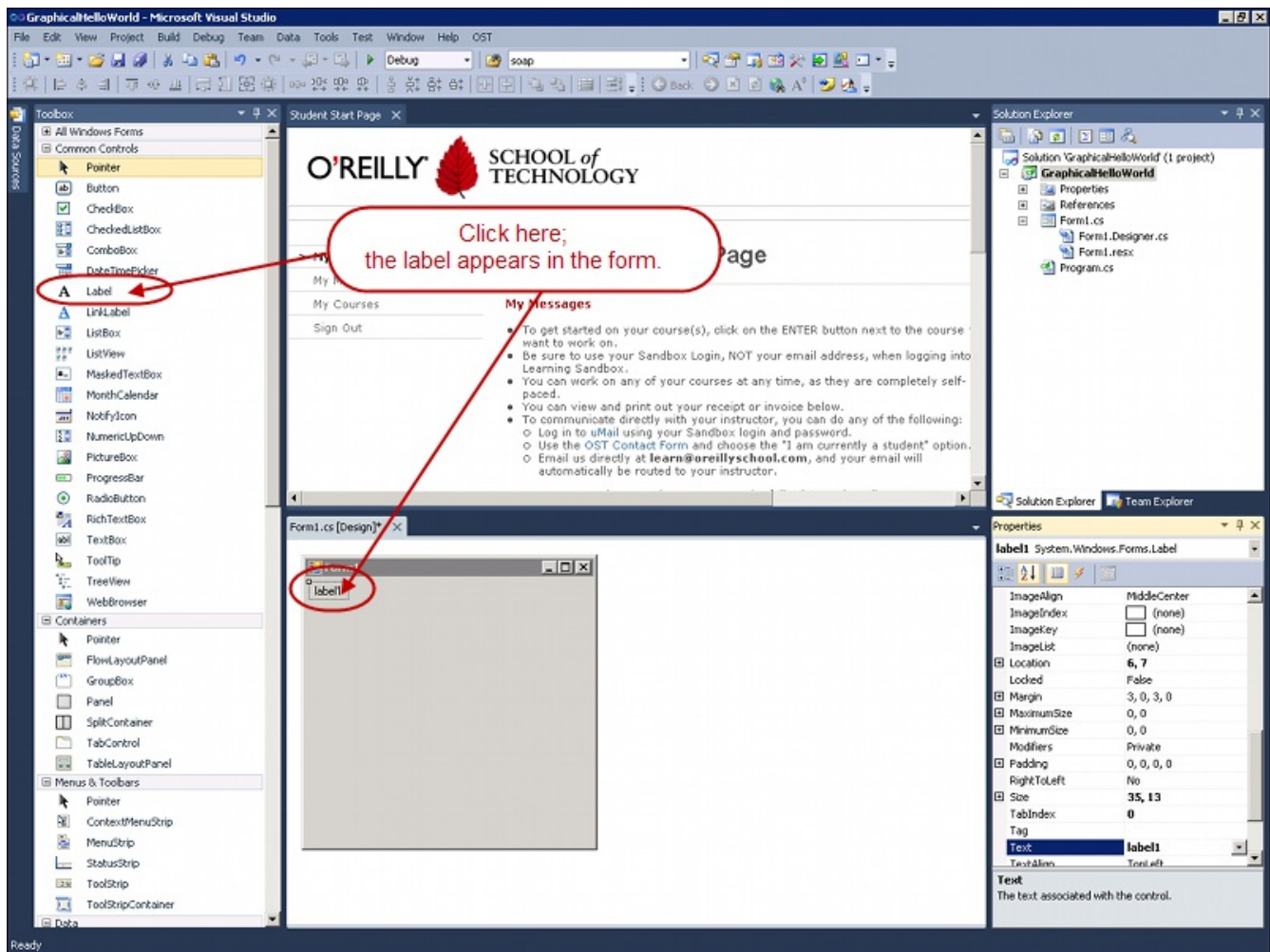
Although this form looks like an independent application, Studio is still connected to it. You can see how Studio is still linked to the running GraphicalHelloWorld application by the choices under the Debug menu.

To stop running an application, you can either close it using the  icon at the top-right corner of the application, or you can switch back to Studio and click the blue **Stop** icon . All of the Debug commands are also available from the top **Debug** menu.

Click **Close**  in the top-right corner of the running GraphicalHelloWorld application.

By default, a Windows Form Application is pretty empty. In our first C# project, we added the Hello World text using the Toolbox. The Toolbox contains components we can use in our application. Now we'll add the text again, and then change it to a graphical version of the text. Again, we will first use a label control.

Double-click the **Label** control in the Common Controls option in the list of hierarchical groupings of the Toolbox. A new label control appears on the form.

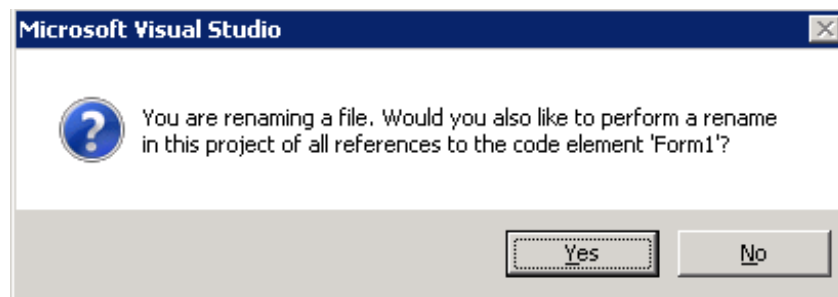


A label control may be used to display simple text for a variety of reasons. A text control is typically not a control that a user can click on.

Right-click on the label control in the form, and select **Properties**. Note the appearance of the Properties window in the bottom right of Studio (it may already have been visible, as in the image above). Scroll down the left column to the Text property, click on the field next to the Text property, and type **Hello World** to replace the "label1" text. To change the text of the label control in the form, you can either press **Enter** or click away from the field.

We will learn more about the different windows components of the Toolbox later.

Notice in the Solution Explorer that the name of the Form1 file is Form1.cs. You need to change the name of Form to something meaningful, in this case, GraphicalHelloWorld.cs. Click on **Form1.cs** to select it in the Solution Explorer. Once it is selected, the Properties window will show the Form1 properties. Change the File Name property from Form1.cs to **GraphicalHelloWorld.cs** and press **Enter**. You may see this dialog box:



This indicates that changing the filename will impact other portions of your code, and that Studio can automatically update all references to the old name to the new one. This feature of having the IDE assist you when you make such changes is known as refactoring, and we will see more refactoring opportunities later. For now, click **Yes** to perform the updates.

Press **F5** to run the GraphicalHelloWorld application and view the Hello World text (if it doesn't run, click in the Design

panel area where the form is, and try again). To stop running the application, click its Close button .

Tip

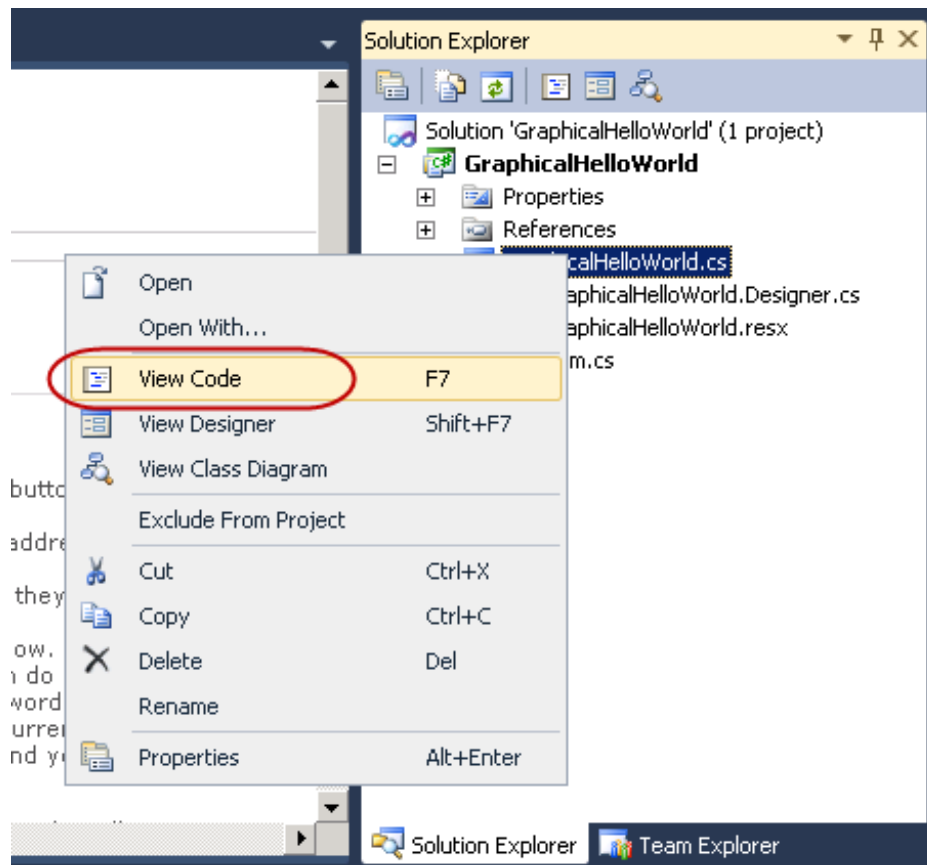
Get in the habit of saving your work. Often, we will pause while we think about doing something. Train yourself to click **Save** or **Save All** during these idle times.

Click **Save** or **Save All**, or press **Ctrl+S**.

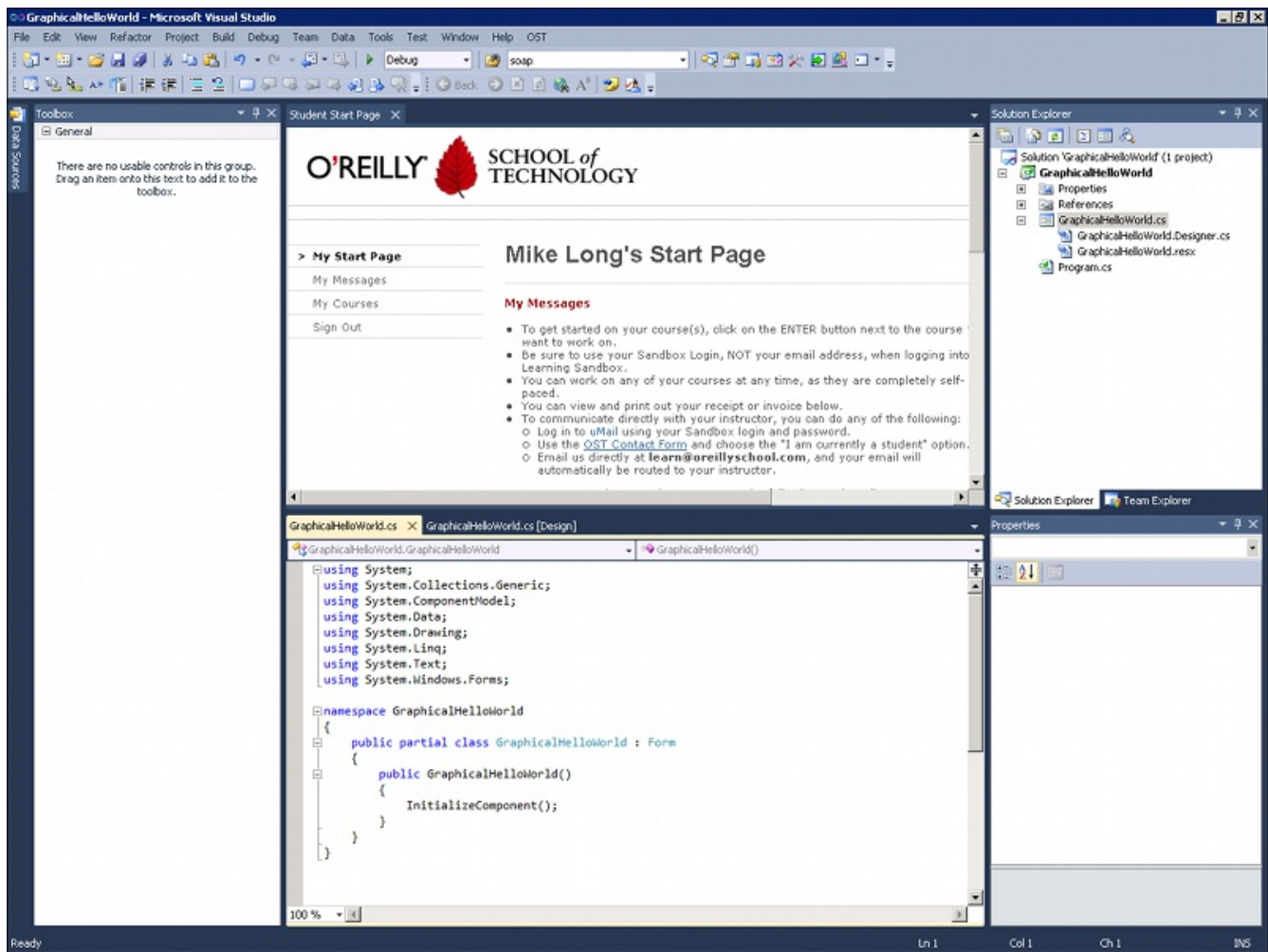
Programming Code

Before we can put a bit of a twist on the GraphicalHelloWorld application, we need to discuss the program code that makes it work. Studio includes a number of text or code editors to make writing your code a very interactive and efficient process.

Right-click the **GraphicalHelloWorld.cs** entry in the Solution Explorer and select **View Code** (again, you'll want to display it in a **New Horizontal Tab Group**):



Studio adds a new tab to the central window showing the programming code, also known as source code, for the form. This window, shown below, is a C# Code Editor. We will learn more about the code later.



Tip To increase the size of the source code window, you can hide the Toolbox window by "unpinning" the AutoHide pushpin.

Modify the form code to look exactly like the code shown below (remember: type the **highlighted** code yourself rather than cutting and pasting!). Note the addition of the line of code below InitializeComponent().

CODE TO TYPE: GraphicalHelloWorld

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

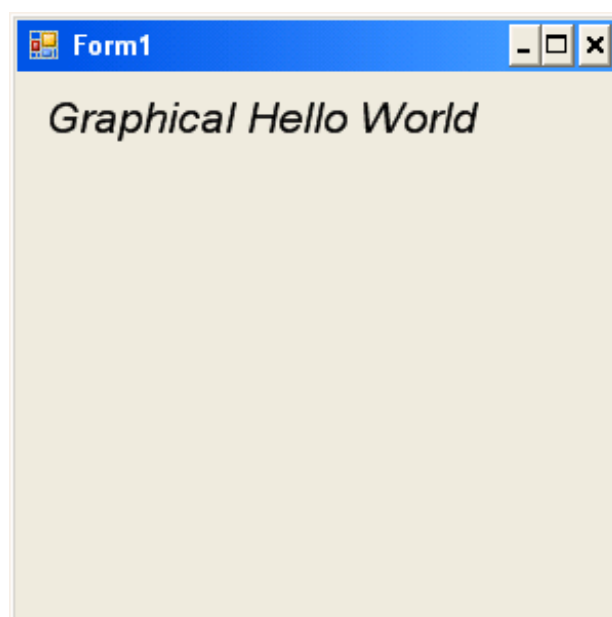
namespace GraphicalHelloWorld
{
    public partial class GraphicalHelloWorld : Form
    {
        public GraphicalHelloWorld()
        {
            InitializeComponent();
            // Hide the label to prevent obscuring graphical Hello World
            this.label1.Visible = false;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            // Call the OnPaint method of the base class
            base.OnPaint(e);
            // Call methods of the System.Drawing.Graphics object
            e.Graphics.DrawString("Graphical Hello World",
                new System.Drawing.Font("Arial", 16.0F, FontStyle.Italic),
                new SolidBrush(ForeColor), 10.0F, 10.0F);
        }
    }
}
```

Note

The "using" section won't change much; you can collapse (and thus hide) the section by clicking the minus (-) icon to the left of it; to expand and view it again, click the plus (+).

Save your changes, and press **F5** to run the modified program. Now the "Graphical Hello World" text in italics is drawn on the form, rather than using the label control:



So how does this code work? Let's OBSERVE and discuss it.

OBSERVE: GraphicalHelloWorld Form

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace GraphicalHelloWorld
{
    public partial class GraphicalHelloWorld : Form
    {
        public GraphicalHelloWorld()
        {
            InitializeComponent();
            // Hide the label to prevent obscuring graphical Hello World
            this.label1.Visible = false;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            // Call the OnPaint method of the base class
            base.OnPaint(e);
            // Call methods of the System.Drawing.Graphics object
            e.Graphics.DrawString("Graphical Hello World",
                new System.Drawing.Font("Arial", 16.0F, FontStyle.Italic),
                new SolidBrush(ForeColor), 10.0F, 10.0F);
        }
    }
}
```

Note

In the OBSERVE box, we added color to different elements that we want to focus on. As you proceed with the lessons, you'll see this color-coding to identify different elements in the code, and to help clarify what is happening.

In the code above, **this.label1.Visible = false;** prevents the display of the Label control to prevent obscuring the drawing of "Hello World" text. Throughout the code, you see lines starting with "//." These are single-line comments, used to help document the code (and ignored at runtime). As you learn to program, you should add comments that help you and that document the functionality of your code.

Add // before the **this.label1.Visible = false;** line and save and rerun the program, then remove the // and save and run it again, to see what that line of code does.

The second change to the code adds an **OnPaint** method that draws the actual "Graphical Hello World" text on the Form. We will learn more about how this code works later.

Note

When you make changes to the code, Studio highlights these changes with a yellow bar at the left of the Code Editor window. Once you have saved your changes, the yellow bar changes to green. These "changed" bars will remain until the file is closed.

The Source Code Window

The source code created by Studio, and the code you later typed in, follow very strict rules, or language semantics. As you learn more about programming in C#, you will learn these rules. The source code window includes a number of helpful features, such as collapsible source code outlines shown by the gray lines with the minus signs at the left of the window. You can click on the minus signs to collapse one or more sections of code to ease the adding and editing of other code.

Click on the minus sign next to the line with the OnPaint text. Click on the plus sign that appears to redisplay the OnPaint source code.

The source code editor also includes an *Intellisense* feature that assists you with the language semantics, prompting for available elements when coding, and providing a *autocompletion* feature that, for example, adds a closing parenthesis) when you enter an opening one (. We will focus more on this and other features of the source code editor later.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then, select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Now you're ready to move on where you'll dive right in, learning more about Visual Studio, and create another C# application!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Statements, Keywords, Scope, Types, Variables, and Conversions

In the first course, the emphasis was on moving through a lot of basic material to get up and running quickly. This second course will again emphasize *hands-on* programming, but will be adding more depth to your experience with the C# language and the Visual Studio IDE.

In this lesson, we will learn more about the syntax, data types, and variables used with C#.

As with the first course, we encourage you to type in all sample code, as well as experiment on your own!

Programming Statements

Previously, we've seen numerous *programming statements*. For the purpose of this course, programming statements are like sentences in a spoken language, composed of elements of the programming language to represent the smallest valid unit. The following are examples of simple programming statements in C#:

- **Assignment:** `int myAge = 21;`
- **Method call:** `getEmployeeName (employeeNumber);`
- **Simple branching:** `break;`
- **Method return:** `return myAge;`

You can also use compound programming statements:

- **Branching:** `if (myAge >= 21) ...`
- **Iteration:** `for (int rowCounter = 1; rowCounter <= totalRows; rowCounter++) ...`

We construct programming statements from the building blocks of the programming language. These building blocks consist of *reserved keywords* (which, as we've mentioned, may not be used as identifier names) and *identifiers* (classes, methods, variables) assembled using the C# *language syntax*.

Identifiers and Reserved Words

The concept of programming statements, identifiers, reserved words, and language syntax should be familiar to you from the first course. You've used quite a few of the C# reserved words already, such as *class*, *if*, *else*, *for*, *int*, *string*, and so on. Microsoft provides a complete list of reserved words used with the C# language and Visual Studio 2010 here:

[Visual Studio 2010 C# Keywords](#)

We've defined the rules for a C# variable earlier, but we'll repeat those rules here as they apply to all identifiers (classes, methods, and variables):

- Must use only letters, digits, or the underscore character.
- Must start with a letter or underscore.
- Is case-sensitive, so `radiusOfCircle` is NOT the same as `RadiusOfCircle`.
- Must *not* be a C# keyword, such as `class`, `int`, etc.

Note

Feel free to investigate some of the keywords you're familiar with, and others you're not familiar with. Also, you may note a second table of keywords referred to as "Contextual Keywords." We'll discuss these keywords later, as we cover material related to them.

Scope

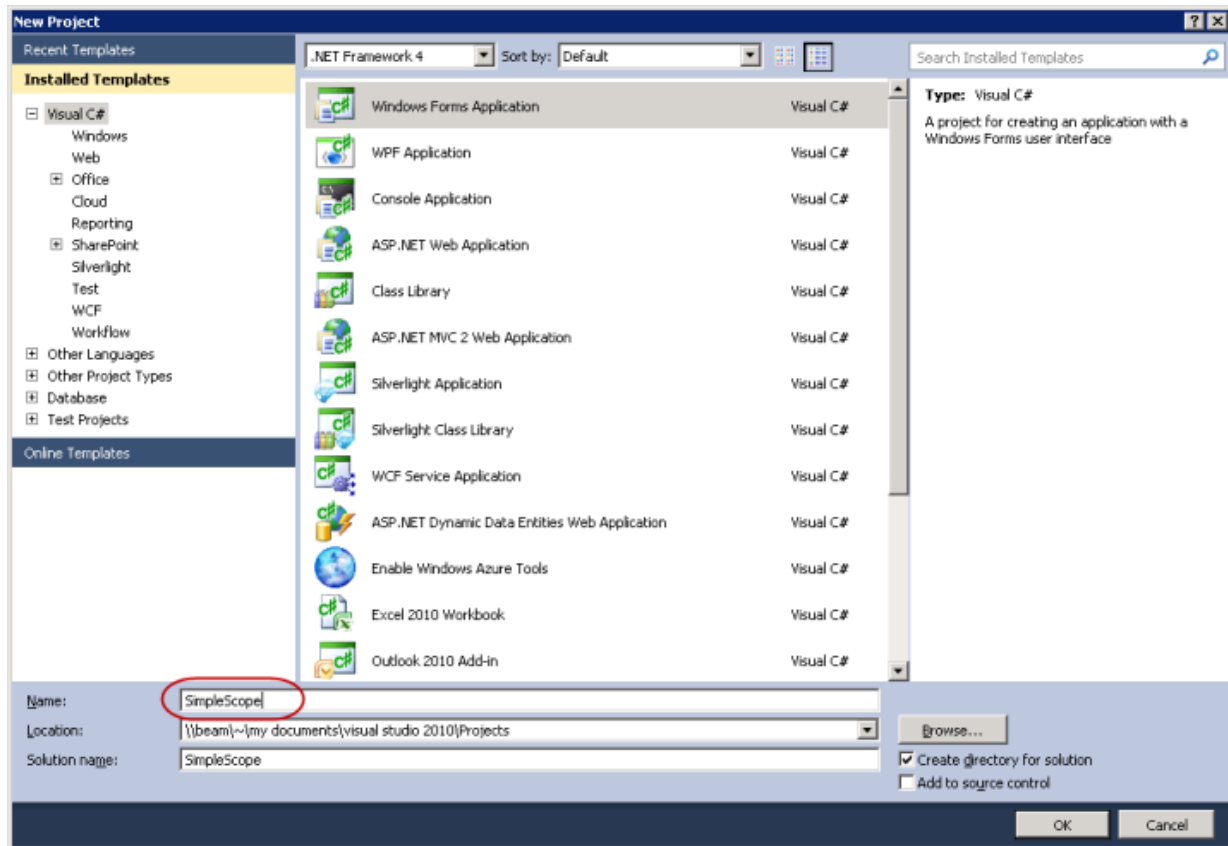
Scope Definition

Remember all of those curly braces (`{` and `}`) we used in our programs? They are part of an important concept known as *scope*. Just as we use a microscope to reveal things we may not be able to see without assistance,

programming scope refers to the visibility of the variables, classes, objects, and methods throughout our software. Let's create a new project to explore the concepts of scope, including the use of braces.

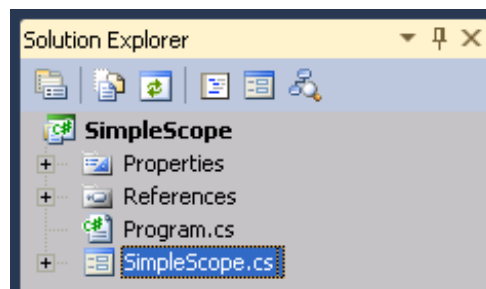
Local Scope

Select **File | New | Project**. In the New Project dialog box, change the Name of the project to **SimpleScope**. All other dropdowns and checkboxes should have their default settings:

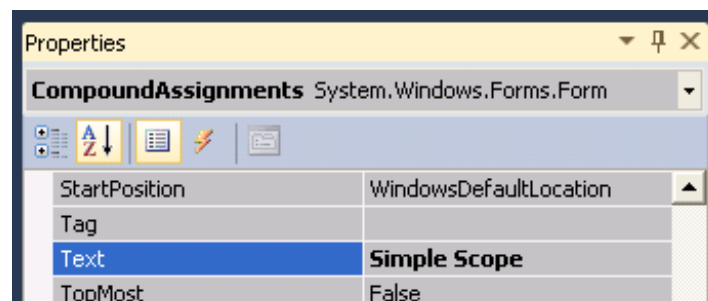


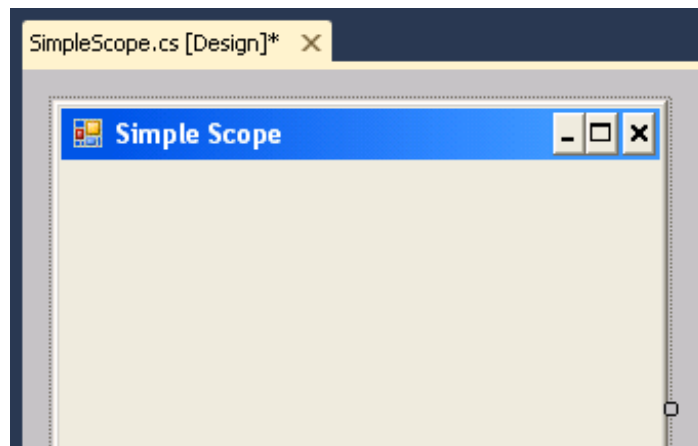
Click **OK**. Your new Windows Form Application appears.

Locate and click on the entry for **Form1.cs** and replace the highlighted text with **SimpleScope.cs**.

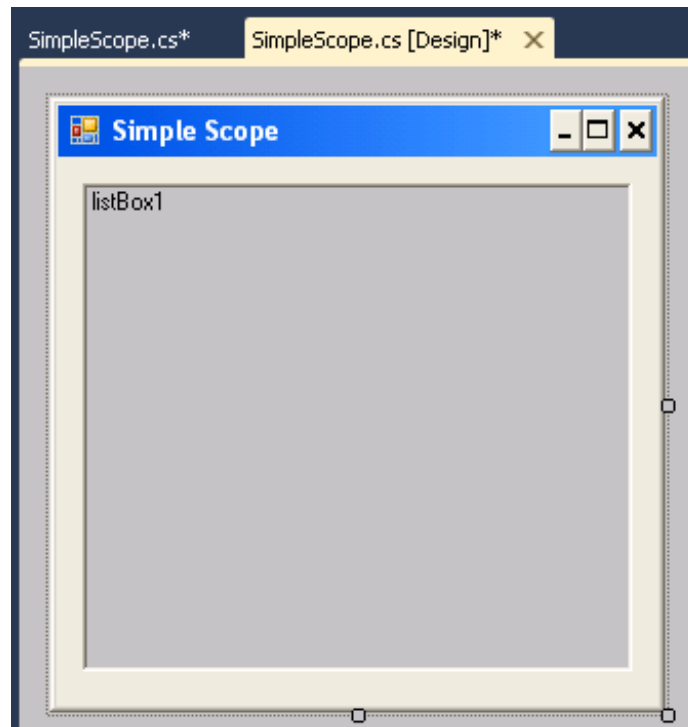


Click on the **Form1** Form title bar, and find the Text property in the Property Window. Change the Text property from Form1 to **Simple Scope**:

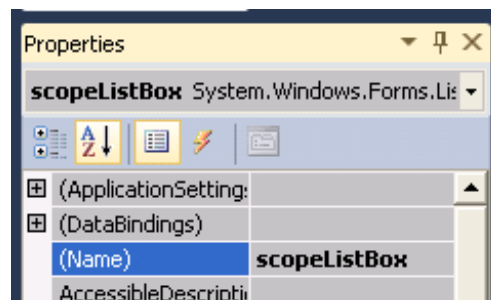




Drag a ListBox control from the Toolbox onto your form, and resize the ListBox to take up most of the space in the form:



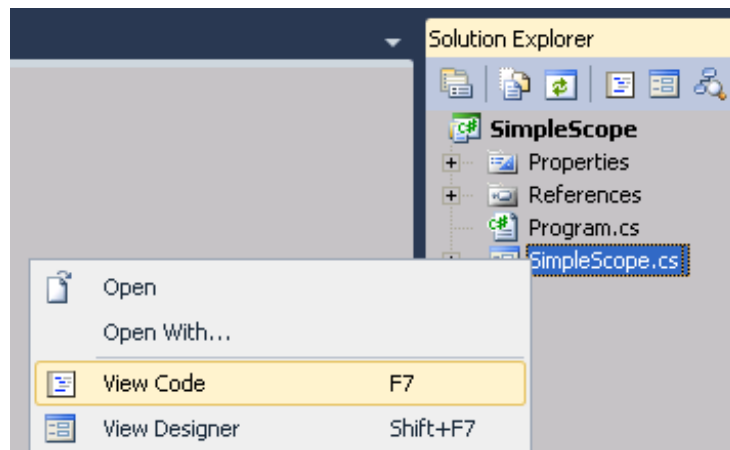
Click the new ListBox control, and in the Properties Window, change the Name property to **scopeListBox**.



This is a good time to click .

Next, we'll add code to our project, so we'll need to view the Code Editor for the Form.

Right-click the **SimpleScope.cs** Form entry in the Solution Explorer, and select **View Code**:



You should see the default Windows Form code for your project in the Code Editor:

```
SimpleScope.cs x SimpleScope.cs [Design]
SimpleScope.SimpleScope
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        public SimpleScope()
        {
            InitializeComponent();
        }
    }
}
```

Now, let's add our C# code to explore the concept of scope.

Modify the SimpleScope.cs code as shown below to add the decision-making code:

SimpleScope.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        public SimpleScope()
        {
            InitializeComponent();

            // Scope variables
            int numberToSquare = 3;
            int squaredResult = 0;

            // Call square method and document values.
            scopeListBox.Items.Add("Number value BEFORE method call: " + numberTo
oSquare);

            squaredResult = square(numberToSquare);
            scopeListBox.Items.Add("Number value AFTER method call: " + numberTo
Square);

            scopeListBox.Items.Add("Result: " + squaredResult);
        }

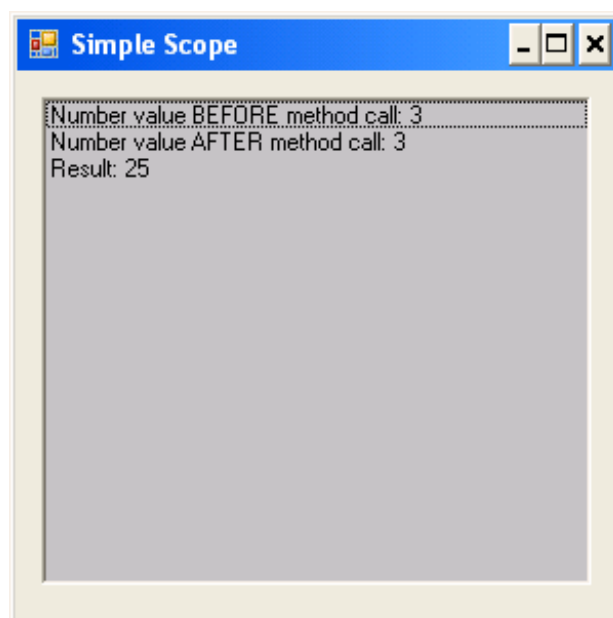
        private int square(int numberToSquare)
        {
            numberToSquare = 5;
            return (numberToSquare * numberToSquare);
        }
    }
}
```



and



to run the program. You should see this:



As you can see, the value of **numberToSquare** remains the same before and after the call to the **square()** method, but inside the method, the **numberToSquare** value assignment worked, as the result of the method

call is the square of 5, or 25.

Let's discuss how this code works. As we did in the first course, we'll omit the code that doesn't concern us.

OBSERVE: SimpleScope.cs

```
.
.
.
    public SimpleScope()
    {
        InitializeComponent();

        // Scope variables
        int numberToSquare = 3;
        int squaredResult = 0;

        // Call square method and document values
        scopeListBox.Items.Add("Number value BEFORE method call: " + numberTo
oSquare);
        squaredResult = square(numberToSquare);
        scopeListBox.Items.Add("Number value AFTER method call: " + numberTo
Square);
        scopeListBox.Items.Add("Result: " + squaredResult);
    }

    private int square(int numberToSquare)
    {
        numberToSquare = 5;
        return (numberToSquare * numberToSquare);
    }
.
.
.
```

Examine the use of the variable **numberToSquare**/**numberToSquare**. Yes, they are two different variables. **numberToSquare** is used in the SimpleScope method, and **numberToSquare** is a method parameter or argument to the **square** method. Consider these questions:

- Are these two variables really the same variable, perhaps because they both have the *same name*?
- Is the value of **numberToSquare** the same *before* and *after* the call to **square**?
- What happens to **numberToSquare** *after* the call to **square**?

As you consider these questions, recognize that what we're really asking about is *scope*. Does a variable declared within one method become the same variable in another method if both variables have the same name as in our code? The answer is No. The scope, or visibility, of a variable is determined by where the variable is declared. A variable declared within a method is said to have *local scope*. Once that method has been executed, any local variables, including method parameters and arguments, no longer exist. So what are the answers to the scope questions?

- The two **numberToSquare** variables are not the same, as each exists within its own local scope.
- The value of **numberToSquare** remains the same, even if you changed **numberToSquare** within the **square** method.
- Once the **square** method exits, **numberToSquare** no longer exists.

Block Scope

Within a method, you can have additional levels of scope known as *block scope*. This level of scope is created whenever you use those pernicious {braces}! That means if you use C# keywords like `if()`, you're creating a block! Let's try that.

Modify the SimpleScope.cs code as shown below.

SimpleScope.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        public SimpleScope()
        {
            InitializeComponent();

            // Scope variables
            int numberToSquare = 3;
            int squaredResult = 0;

            // Call square method and document values
            scopeListBox.Items.Add("Number value BEFORE method call: " + numberToSquare);

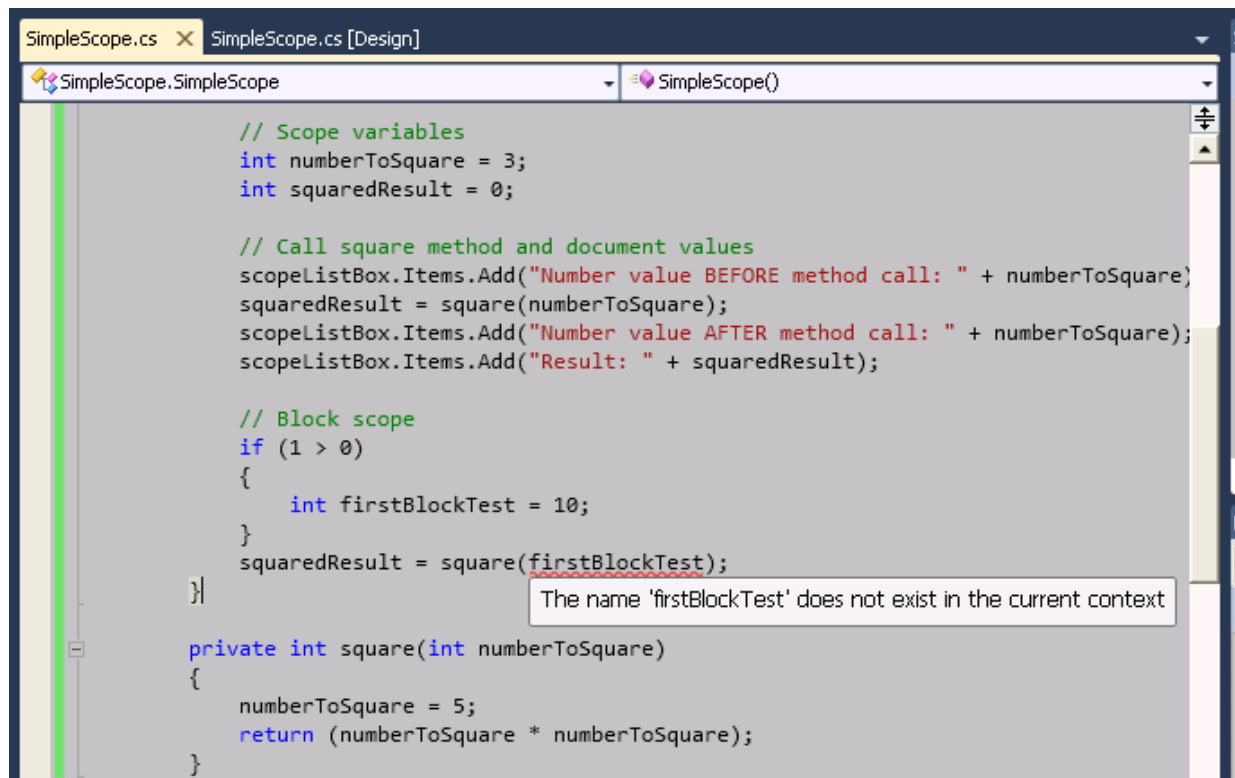
            squaredResult = square(numberToSquare);
            scopeListBox.Items.Add("Number value AFTER method call: " + numberToSquare);

            scopeListBox.Items.Add("Result: " + squaredResult);

            // Block scope
            if (1 > 0)
            {
                int firstBlockTest = 10;
            }
            squaredResult = square(firstBlockTest);
            scopeListBox.Items.Add("Result after block: " + squaredResult);
        }

        private int square(int numberToSquare)
        {
            numberToSquare = 5;
            return (numberToSquare * numberToSquare);
        }
    }
}
```

After adding the block code, you'll see a squiggly red error line under the `firstBlockTest` variable. If you hover your mouse pointer over the error, you will see a popup message that indicates the variable "`firstBlockTest`" does not exist in the current context:



But we initialized `firstBlockTest` to 10 just a couple of lines earlier! What does this error mean? It means that the variable `firstBlockTest`, even though it was declared within the block `if()` scope, is no longer valid because the variable has gone *out of scope*. Variables are often referred to as *in scope* or *out of scope*, to indicate whether or not the variable exists and is visible to the current scope. Let's fix the code so we can use the `firstBlockTest` variable.

Note

Previously, we saw that you do not have to use braces with the `if()` syntax if you only have a single line, although we've encouraged you to be consistent and include the braces in all cases. When using a single-line `if()` block without braces in C#, you cannot declare a new variable. You must use braces if you want to declare new block-level variables.

Modify the **SimpleScope.cs** code as shown below:

SimpleScope.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        public SimpleScope()
        {
            InitializeComponent();

            // Scope variables
            int numberToSquare = 3;
            int squaredResult = 0;

            // Call square method and document values
            scopeListBox.Items.Add("Number value BEFORE method call: " + numberToSquare);
            squaredResult = square(numberToSquare);
            scopeListBox.Items.Add("Number value AFTER method call: " + numberToSquare);
            scopeListBox.Items.Add("Result: " + squaredResult);

            // Block scope
            int firstBlockTest = 0;
            if (1 > 0)
            {
                int firstBlockTest = 10;
            }
            squaredResult = square(firstBlockTest);
            scopeListBox.Items.Add("Result after block: " + squaredResult);
        }

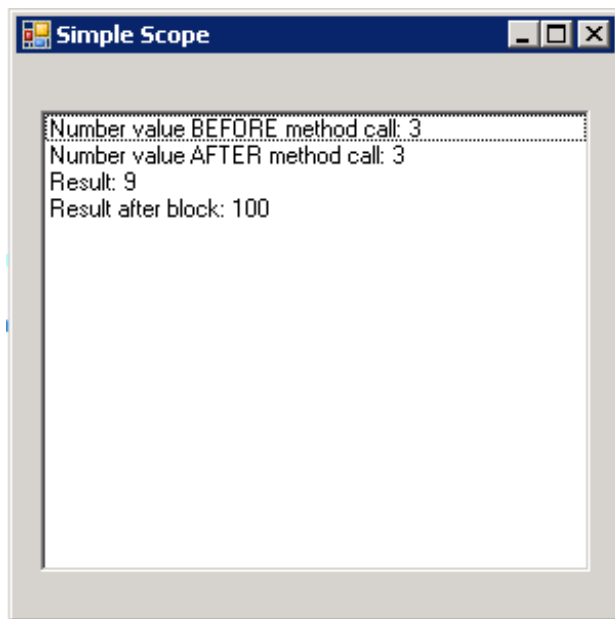
        private int square(int numberToSquare)
        {
            return (numberToSquare * numberToSquare);
        }
    }
}
```



and



to run the program. You should see this:



Let's discuss how this code works.

```
OBSERVE: SimpleScope.cs

.
.
.

    // Block scope
    int firstBlockTest = 0;
    if (1 > 0)
    {
        firstBlockTest = 10;
    }
    squaredResult = square(firstBlockTest);
    scopeListBox.Items.Add("Result after block: " + squaredResult);
.
.
.
```

Because the variable **firstBlockTest** is now declared outside the if block, its scope is broadened; within the braces block, we're simply changing its value. You should note that you *may not have a block scope variable with the same name as a local scope variable*.

Note Although the technique is rarely used, you can create a block scope simply by using braces! Go ahead, try it!

Class Scope

In previous lessons, we've seen that we can declare variables that serve as properties for our class. These class variables have *class scope*, and are visible to all of the methods within the class in which they are declared. Unlike with local and block scope variables, *you are allowed to have a class scope variable with the same name as a local or block scope variable; however, a local or block scope variable takes precedence and will be used over the class scope variable!* We'll learn in a later lesson how to select the correct variable when a class and local variable have the same name.

Let's add a class-level variable.

Modify **SimpleScope.cs** as shown below.

SimpleScope.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        private int myClassVariable = 25;

        public SimpleScope()
        {
            InitializeComponent();

            // Scope variables
            int numberToSquare = 3;
            int squaredResult = 0;

            // Call square method and document values
            scopeListBox.Items.Add("Number value BEFORE method call: " + numberToSquare);
            squaredResult = square(numberToSquare);
            scopeListBox.Items.Add("Number value AFTER method call: " + numberToSquare);
            scopeListBox.Items.Add("Result: " + squaredResult);

            // Block scope
            int firstBlockTest = 0;
            if (1 > 0)
            {
                firstBlockTest = 10;
                firstBlockTest = 10;
            }
            squaredResult = square(firstBlockTest);
            scopeListBox.Items.Add("Result after block: " + squaredResult);

            // Use class scope variable.
            scopeListBox.Items.Add("Class variable result: " + square(myClassVariable));
        }

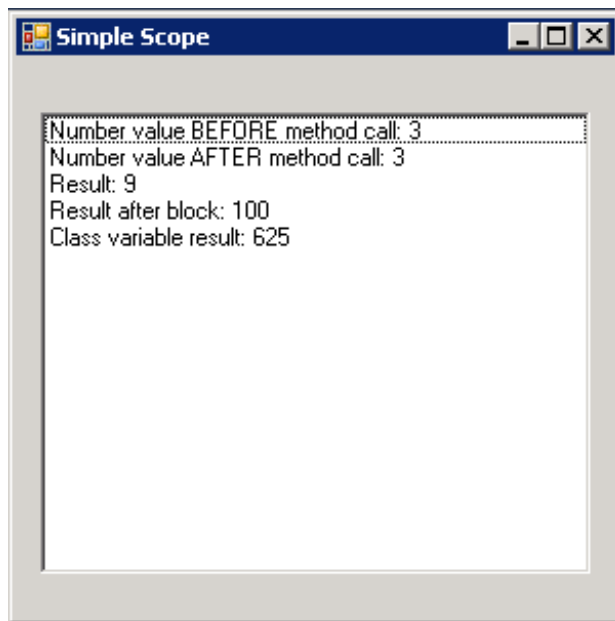
        private int square(int numberToSquare)
        {
            return (numberToSquare * numberToSquare);
        }
    }
}
```



and



to run the program. You should see this:



Let's discuss how this code works.

```
OBSERVE: SimpleScope.cs

.
.
.
namespace SimpleScope
{
    public partial class SimpleScope : Form
    {
        private int myClassVariable = 25;

        public SimpleScope()
        {
            .
            .
            .

            // Use class scope variable
            scopeListBox.Items.Add("Class variable result: " + square(myClassVar
iable));
        }

        private int square(int numberToSquare)
        {
            return (numberToSquare * numberToSquare);
        }
    }
    .
    .
    .
}
```

The class variable **myClassVariable** is declared outside of any method, at the class scope. We use the variable in our call to **square** and output the results. As you can see from the code, we did not need to declare **myClassVariable** within the local scope to use the variable.

Scope Importance

We've discovered the differences between local, block, and class scope, but why is scope so important? Why not just declare all variables at the class level? Variables in general are what we use to hold information as our programs perform the tasks we've programmed THEM to perform, so ensuring our variables only have the data and values we require is very important for *data integrity*. As we implement an object-oriented approach to software development, we want to employ the concept of *information hiding*: limiting access to data and variables to only those parts of our program, and other software, that needs access. Variables require *memory*, so we should limit class variables to reduce a program's memory requirements, and use

local variables that effectively use memory only temporarily. Finally, declaring and using variables in *close proximity* to where they are used, such as within a block, makes *implementing, understanding,* and later *deciphering* a program a simpler task.

Data Types

Built-In Types

We've used a number of different data types so far, such as *bool, int, double, float, long,* and *string*. These are just some of the data types available in C#. Here's a complete list of available data types (note table below as well):

Visual Studio 2010 C# Data Types

As with reserved words, we'll continue to explore the data types as we work through these lessons, but feel free to click on any of the data types and read more about what they represent. One common topic related to data types is the amount of computer memory they use, and the ranges of values they can contain. Below are all data types, their ranges of values, the amount of memory they use, a reference to the underlying .NET data type, and a column about precision, if applicable.

Type	Range	Size	.NET Framework Type	Precision
bool	true, false	Unsigned 8-bit integer	System.Boolean	
sbyte	-128 to 127	Signed 8-bit integer	System.SByte	
short	-32,768 to 32,767	Signed 16-bit integer	System.Int16	
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32	
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64	
float	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	Signed 32-bit integer	System.Single	7 digits
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	Signed 64-bit integer	System.Double	15-16 digits
decimal	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / (10^0 \text{ to } 28)$	Signed 128-bit integer	System.Decimal	28-29 significant digits
byte	0 to 255	Unsigned 8-bit integer	System.Byte	
ushort	0 to 65,535	Unsigned 16-bit integer	System.UInt16	
uint	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32	
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64	
char	U+0000 to U+FFFF	Unicode 16-bit character	System.Char	
string	Character data	Operating system dependant	System.String	
object	Object data	Operating system dependent	System.Object	

You'll notice that the table contains some familiar data types and unfamiliar data types. There are a number of integer data types, or numbers that do not have any decimal value: sbyte, short, int, and long. There are data types that have decimal places: float, double, and decimal. There is the familiar boolean data type bool. There are data types that are like integers, but must be the value 0 or higher (non-negative): byte, ushort, uint, and

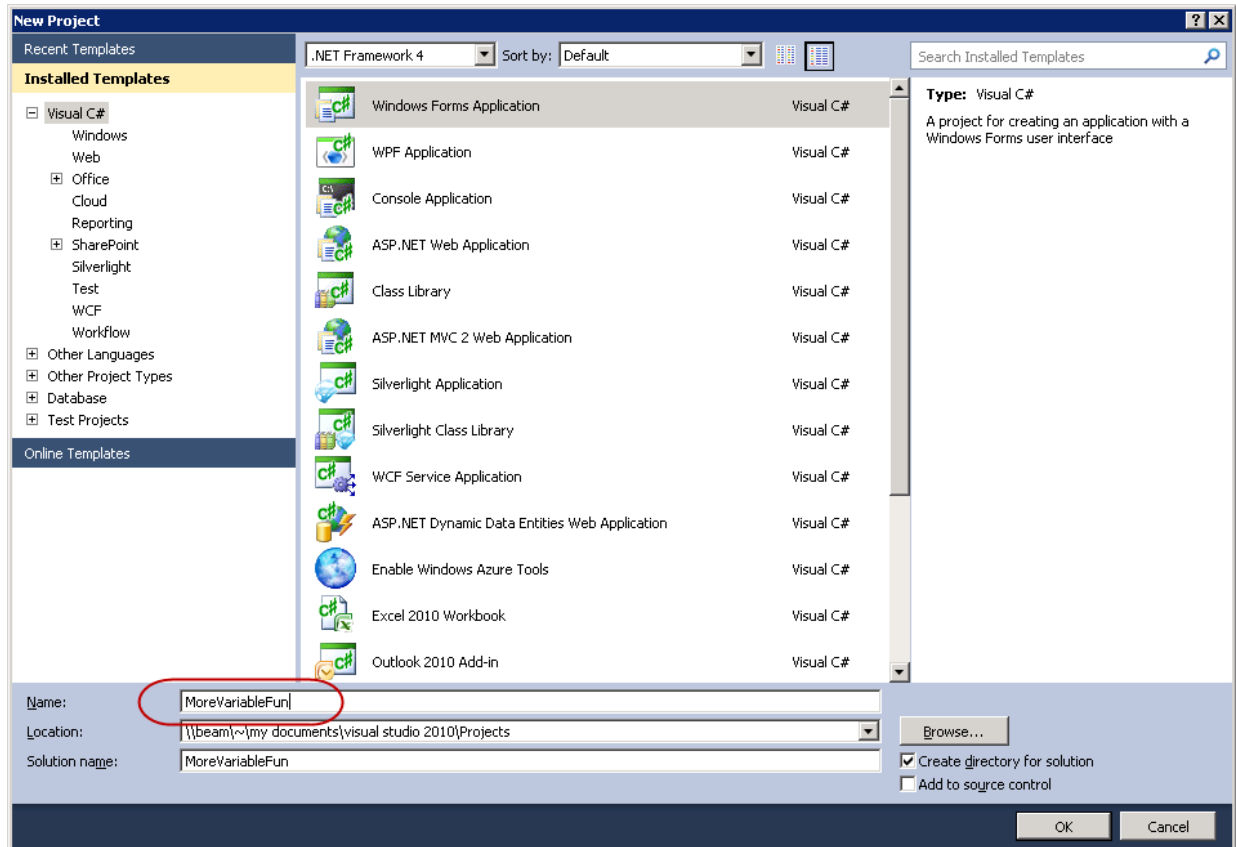
ulong. There is the string data type we've used, and another called `char` that only allows a single character. And, finally, there is the object data type, which is the parent data type that all predefined and user-defined data types inherit.

Note

As C# is but one programming language supported by the .NET framework, and as with all other .NET-based languages, all data types used in C# represent an underlying .NET data type. The **.NET Framework Type** column indicates the actual underlying .NET data type. They are interchangeable; you can use the C# or the .NET data type in your C# programs. For consistency, we recommend you stick with the C# data type names.

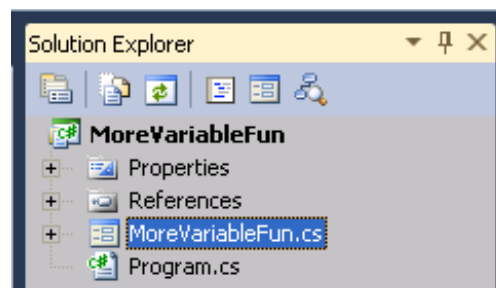
Let's experiment more with the different data types.

Selecting **File | New | Project**. In the New Project dialog box, change the Name to **MoreVariableFun**; all other dropdowns and checkboxes should have their default settings. Click **OK**.



The new Windows Form Application appears.

Locate and click the entry for **Form1.cs** and replace it with **MoreVariableFun.cs**. Click  to save your changes.



Next, let's add an `sbyte` and `SByte` to our code, and see what Intellisense tells us about this smallest of integer data types.

Modify **MoreVariableFun.cs** as shown below:

MoreVariableFun.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            sbyte firstVariable = 1;
            SByte secondVariable = firstVariable;
        }
    }
}
```

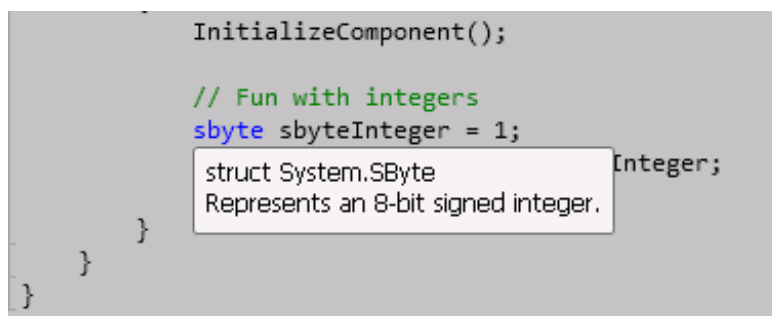
As we mentioned, `sbyte` and `SByte` (`System.SByte`) are synonymous in C#. You should use the C# `sbyte` data type rather than the .NET underlying variable data type, but this example illustrates that you can use them interchangeably. The code also illustrates that—as you would expect with compatible data types—you can assign the value of one variable, `firstVariable`, to another of the same data type, `secondVariable`.

MoreVariableFun.cs

```
.
.
.
    InitializeComponent();

    // Fun with integers.
    sbyte firstVariable = 1;
    SByte secondVariable = firstVariable;
.
.
.
```

Further, if you hover your mouse pointer over either `sbyte` or `SByte`, you'll see the same Intellisense popup:



Two key differences with numeric data types are the range of the data type, and the amount of memory needed to hold a variable declared as that specific data type. You must be aware of the data type range, or you will experience what is known as *overflow*, which means you have attempted to store a number that exceeds the capacity of the data type you've chosen—much like pouring 14 ounces of water into a 12-ounce container.

Let's see what happens if we try to set one of the variables to a value larger than it can hold. For `sbyte`, according to our table, the range is -127 to 127.

Modify **MoreVariableFun.cs** as shown:

Note

Remember, you can hide or display different sections of your source code to make it easier to view using the plus or minus symbol that appears to the left of the first line in the section. Here, we're hiding the *using* section.

MoreVariableFun.cs

```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            sbyte firstVariable = 1128;
            SByte secondVariable = firstVariable;
        }
    }
}
```

As soon as you change the assignment from 1 to 128, you will see the red squiggly error line under the 128. Hover your mouse pointer over the 128 and note the popup error box indicating that you can't assign 128 to sbyte, which is what you'd expect from the range of values for an sbyte from the table:

```
// Fun with integers
sbyte firstVariable = 128;
SByte secondVariable =
```

struct System.Int32
Represents a 32-bit signed integer.

Error:
Constant value '128' cannot be converted to a 'sbyte'

Note

You might think that you should try to use the smallest possible data type based on the range of the number you might use, but in general, when declaring method level variables, you should use data types that are divisible by 32, as these numbers store more efficiently in memory, and process faster through the computer processor. So, for an integer, you should use int or long. For numbers with decimal places, you can use float, although the default numeric value in C# is double, so using double data types is more common.

Let's change the code again to allow this larger value by using integer data types that allow a much larger range of values. From our table, the next larger integer data type is short, then int, and, finally, long.

Modify **MoreVariableFun.cs** as shown below.

MoreVariableFun.cs

```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            sbyteshort firstVariable = 128;
            SByteint secondVariable = firstVariable;
        }
    }
}
```

Remember to  periodically!

As you can see, we changed the `sbyte` to `short`, and `SByte` to `int`. Remember, you can almost always assign numeric data of a smaller data type to variables of a larger data type. This process is called *widening*, which we've discussed before, and is always possible when the resulting assignment will not result in a loss of data. In fact, the `short` data type is *promoted* using *implicit casting* during the assignment from a `short` to an `int`. C# uses implicit casting whenever necessary to allow assignment where no data loss will occur. In instances where implicit conversion is not allowed, Visual Studio will indicate that fact with an error.

Note

What does signed and unsigned mean? We've previously indicated that unsigned means the data type can only store values that are greater than or equal to 0, but from a memory point of view, you'll notice that unsigned data types have a slightly larger range than their equivalent signed data types. The reason is that unsigned data types do not have to store the sign, and have more storage space to increase their range. The same recommendation applies when using unsigned data types: use a data type that is divisible by 32.

In our table, for `float`, `double`, and `decimal` types, the Precision column indicates an important point about numbers with decimal places, also known as *precision*. These three data types are not identical. `float` and `double` are both *floating-point* data types, which means that as the number gets larger, the decimal point floats, and the large number is represented using scientific, or exponential, notation. For most calculations, using a floating point data type is sufficient, but if you need to control the precision more precisely, for example with scientific calculations or currency, you should use `decimal`. The **significant digits** refers to precision, which can be fixed and maintained in decimal, but may be lost in the floating point data types.

Let's work with these data types as well.

Modify **MoreVariableFun.cs** as shown below.

MoreVariableFun.cs

```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            short firstVariable = 128;
            int secondVariable = firstVariable;

            // Fun with float, double, and decimal.
            float thirdVariable = 0.0;
            double fourthVariable = thirdVariable;
            decimal fifthVariable = fourthVariable;
        }
    }
}
```

More squiggly red line errors!

```
// Fun with float, double, and decimal|
float thirdVariable = 0.0;
double fourthVariable = thirdVariable;
decimal fifthVariable = fourthVariable;
```

Hover over the squiggly red error line under the 0.0 assignment, and examine the error message.

```
// Fun with float, double, and decimal|
float thirdVariable = 0.0;
double fourthVariable
decimal fifthVariable
```

struct System.Double
Represents a double-precision floating-point number.

Error:
Literal of type double cannot be implicitly converted to type 'float'; use an 'F' suffix to create a literal of this type

You might remember that numeric literals in C#, by default, are doubles. Attempting to assign a double to a float could result in data loss, so Visual Studio will not allow it. The error message gives you a hint on how to fix the problem: add an **F** after the numeric literal **0.0** to change it from a double to a float.

Modify **MoreVariableFun.cs** as shown below:

MoreVariableFun.cs

```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            short firstVariable = 128;
            int secondVariable = firstVariable;

            // Fun with float, double, and decimal.
            float thirdVariable = 0.00F;
            double fourthVariable = thirdVariable;
            decimal fifthVariable = fourthVariable;
        }
    }
}
```

Hover over the squiggly red error line under the fourthVariable variable and examine the error message:

```
// Fun with float, double, and decimal.
float thirdVariable = 0.0F;
double fourthVariable = thirdVariable;
decimal fifthVariable = fourthVariable;
```

(local variable) double fourthVariable

Error:

Cannot implicitly convert type 'double' to 'decimal'. An explicit conversion exists (are you missing a cast?)

This message might be puzzling—why can't a double be converted to a decimal? If we examine the table, you'll note that even though a decimal can hold a much larger range of decimal places, the range of values is much smaller, so Visual Studio doesn't allow you to implicitly cast from double to decimal. In fact, you also can't reverse the process and implicitly cast from decimal to double, as that conversion could result in a loss of precision. So, what can we do? The error message again indicates a possible solution by stating that an *explicit* conversion exists.

Modify **MoreVariableFun.cs** as shown below.

MoreVariableFun.cs

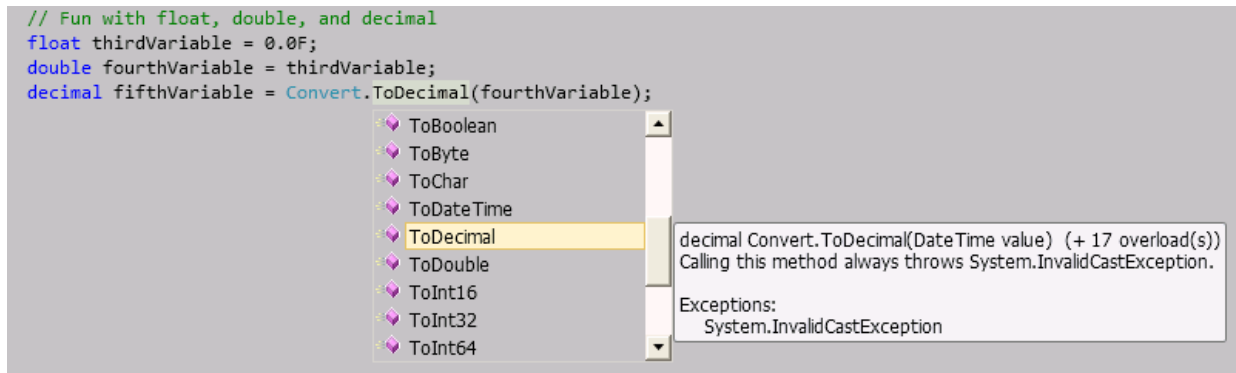
```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            short firstVariable = 128;
            int secondVariable = firstVariable;

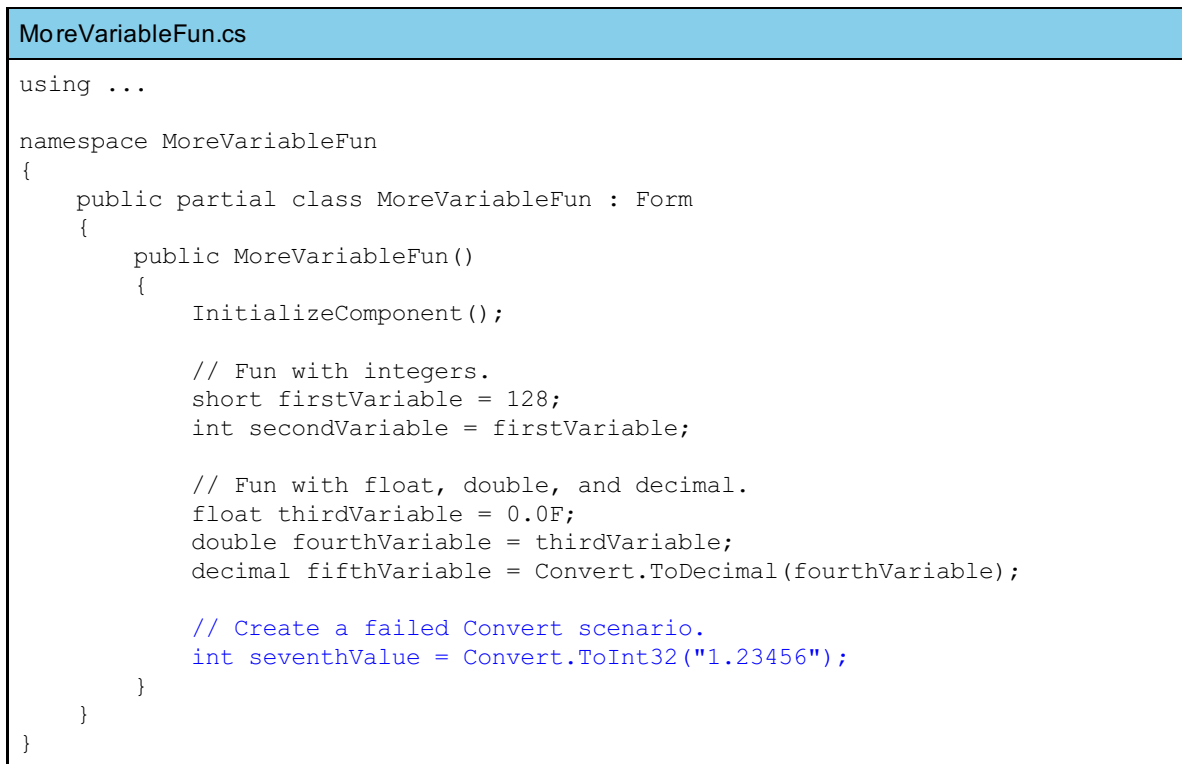
            // Fun with float, double, and decimal.
            float thirdVariable = 0.0F;
            double fourthVariable = thirdVariable;
            decimal fifthVariable = Convert.ToDecimal(fourthVariable);
        }
    }
}
```

That fixed the problem, and showed us a handy object we can use for conversion between different data types: *Convert*. As you type in **Convert**, then type the period, you'll notice the Intellisense dropdown providing an extensive list of conversion options:

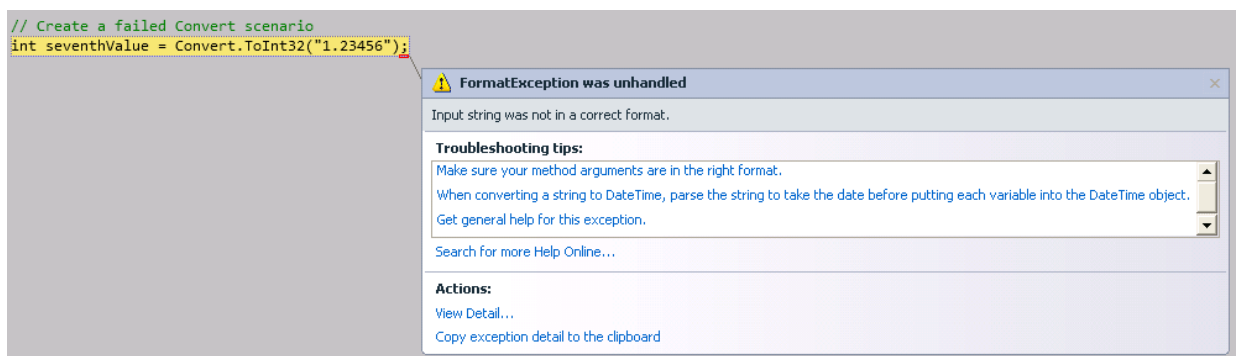


You can use the `Convert` object for most of your conversion needs, and you'll see it used by many C# software developers, but, there is one slight problem with this object. What will happen if you attempt to do a conversion, but the conversion fails? Let's further modify our code to create a failed `Convert` scenario using the `Convert.ToInt32()` conversion to convert a string literal to an int.

Modify **MoreVariableFun.cs** as shown below.



Save your changes, and click  to execute the program. An error, or *exception*, occurs:



So, the Convert object might not always work! We'll learn how to handle such errors later, but in the meantime we can use a different conversion method that will let us know if the conversion was successful or not, using *TryParse*.

Modify **MoreVariableFun.cs** as shown below:

```
MoreVariableFun.cs

using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            short firstVariable = 128;
            int secondVariable = firstVariable;

            // Fun with float, double, and decimal.
            float thirdVariable = 0.0F;
            double fourthVariable = thirdVariable;
            decimal fifthVariable = Convert.ToDecimal(fourthVariable);

            // Create a failed Convert scenario.
int seventhValue = Convert.ToInt32("1.23456");
            int seventhValue;
            if (int.TryParse("1.23456", out seventhValue))
                MessageBox.Show("Conversion successful!");
            else
                MessageBox.Show("Conversion did not work!");
        }
    }
}
```

Save your changes, and click  to execute the program. You'll no longer get an exception; instead, you'll see a message box saying "Conversion did not work!".

Let's discuss how this code works.

```
MoreVariableFun.cs

.
.
.

        // Create a failed Convert scenario.
        int seventhValue;
        if (int.TryParse("1.23456", out seventhValue))
            MessageBox.Show("Conversion successful!");
        else
            MessageBox.Show("Conversion did not work!");

.
.
.
```

Each data type includes a *TryParse* method that works to try to convert from a data type to itself. In our case, we called on the **int** version of **TryParse** to convert the string literal "1.23456" to an int. TryParse will return a boolean—true if the conversion was successful, and false if the conversion failed—which allows us to handle either scenario. Also, if the conversion is successful, the converted value is stored in a parameter passed into the TryParse method. In our case, we used the int variable **seventhValue**. Note the C# keyword parameter modifier **out** is used. The **out** modifier allows us to change the value of **seventhValue** inside the TryParse method.

Note

You may wonder which method is faster: Convert or TryParse. The winner is Convert, but only by a very, very small margin when no conversion error is detected. If an error occurs, TryParse is much slower, but such errors are indeed supposed to be rare, so, in general, using TryParse is preferred for string-to-numeric data type conversions.

To be thorough, let's add one final conversion technique using explicit casting that you may see in code, and you may use yourself. This syntax is a legacy format, and is often used to prevent any implicit casting C# may provide. This type of casting uses a data type surrounded by parentheses.

Modify **MoreVariableFun.cs** as shown below.

MoreVariableFun.cs

```
using ...

namespace MoreVariableFun
{
    public partial class MoreVariableFun : Form
    {
        public MoreVariableFun()
        {
            InitializeComponent();

            // Fun with integers.
            short firstVariable = 128;
            int secondVariable = firstVariable;

            // Fun with float, double, and decimal.
            float thirdVariable = 0.0F;
            double fourthVariable = thirdVariable;
            decimal fifthVariable = Convert.ToDecimal(fourthVariable);

            // Create a failed Convert scenario.
            int seventhValue;
            if (int.TryParse("1.23456", out seventhValue))
                MessageBox.Show("Conversion successful!");
            else
                MessageBox.Show("Conversion did not work!");

            // Force explicit casting.
            decimal eighthVariable = (decimal)fourthVariable;
        }
    }
}
```

Save your changes. Note that there are no squiggly lines—always a good thing!

Let's discuss how this code works.

MoreVariableFun.cs

```
.
.
.    // Force explicit casting.
    decimal eighthVariable = (decimal)fourthVariable;
.
.
.
```

This last example is the same one we used earlier with the Convert object, but this time we've prefixed the **fourthVariable** with **(decimal)**, which forces the data type to be converted to a decimal. This does not guarantee a conversion, and could still result in an error, so when possible, use the TryParse method for strings, and Convert for numeric conversions.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select

Back. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Expressions, Precedence, and Assignment, and Increment/Decrement Operators

In the previous lesson, we introduced the concept of a *programming statement*. In this lesson, we'll further deconstruct statements into *programming expressions*, as well as explore C# mathematical programming, mathematical expressions, and the increment and decrement operators.

Programming Expressions

If you've studied math, you're probably familiar with the concept of an expression. A C# *programming expression* consists of C# language elements that may be combined to create a meaningful computation. Programming expressions produce, or return, a result. These expressions may consist of other expressions. Below is a list of C# expressions. Note that an expression does not need to be a complete programming statement.

- **4+3**: Returns the value of the mathematical calculation.
- **aVariable**: Returns the value or memory address of the variable.
- **aVariable + 3**: Returns the value of the mathematical calculation.
- **aVariable != true**: Returns a boolean value of the relational comparison.
- **getNumber(aVariable)**: Returns a result from a method.

Assignments and Assignment Shortcuts

Simple Assignment Operator

Previously, we've used the assignment operator, the equals sign (=), in a number of different programming circumstances. The assignment operator is used to assign the result of an expression to a variable or other C# construct that accepts assignment. The equals sign is known as the *simple assignment operator*, as there are other, more complex assignment operators. The simple assignment operator, as with the other assignment operators, require that the left side of the equals sign, known as the *left operand*, consist of a C# element that accepts assignment, such as a variable.

Compound Assignment Operators

In addition to the simple assignment operator, C# also includes these *compound assignment operators*:

- **+=**: Compound addition operator
- **-=**: Compound subtraction operator
- ***=**: Compound multiplication operator
- **/=**: Compound division operator
- **%=**: Compound modulus operator
- **&=**: Compound bitwise AND operator
- **|=**: Compound bitwise OR operator
- **^=**: Compound bitwise XOR operator
- **<<=**: Compound shift left operator
- **>>=**: Compound shift right operator

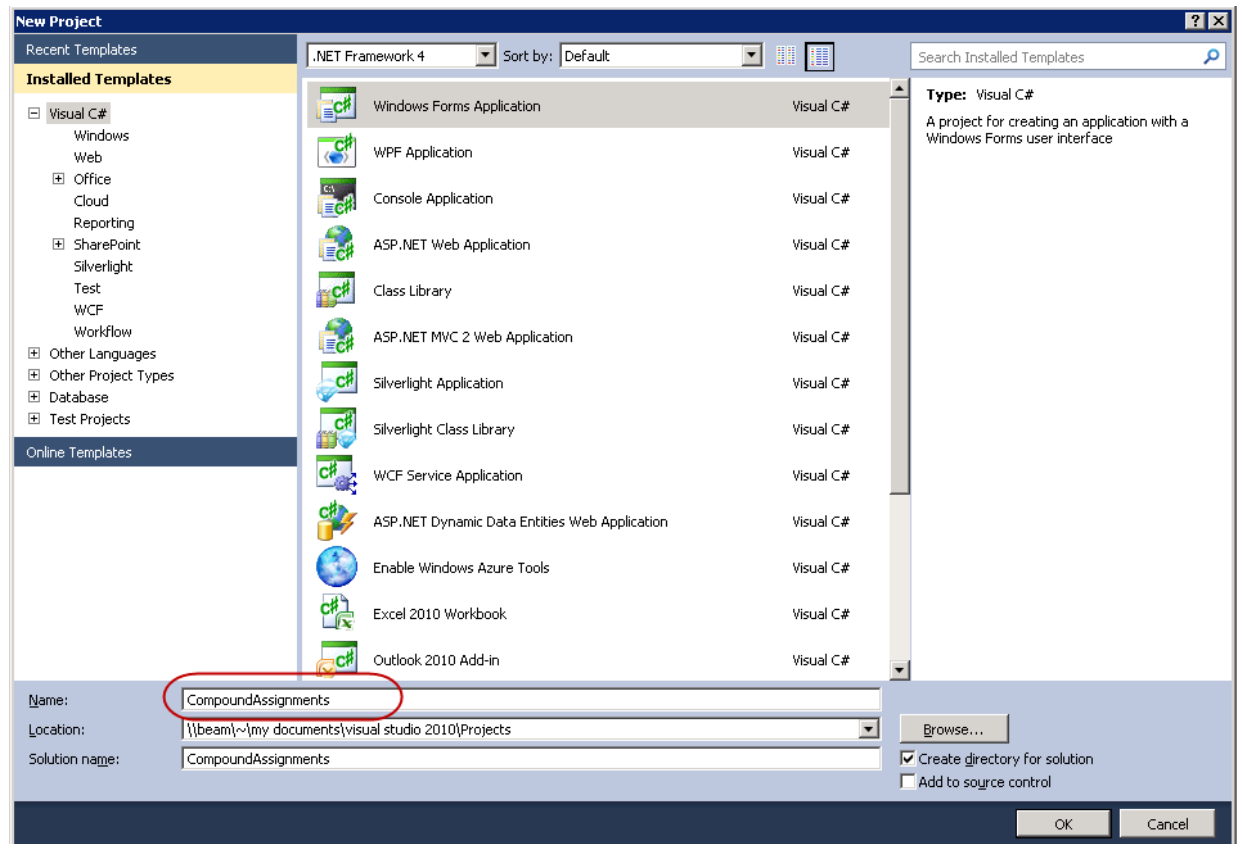
The compound assignment operators are shortcuts. For example, the **+=** operator is a shortcut for the expression **x = x + 5**.

Let's explore these cool shortcuts!

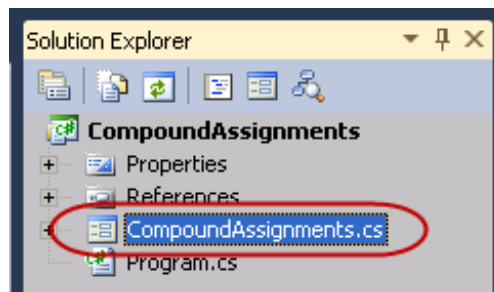
Mathematical Compound Assignment Operators

We will add form controls and code that will allow us to display a grid of different sizes.

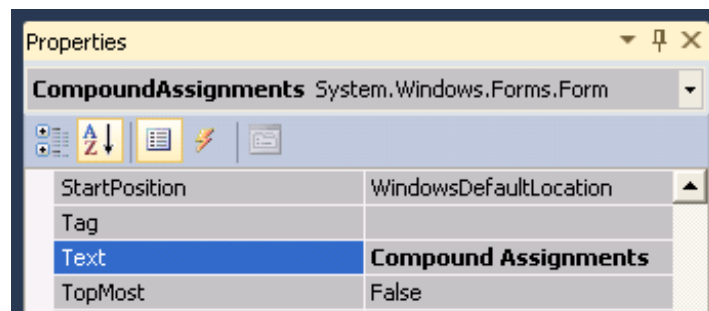
Create a new Visual Studio Windows Forms Application by selecting **File | New | Project**. Change the Name of the project to **CompoundAssignments**, leaving all other dropdowns and checkboxes with their default settings, and click **OK**:

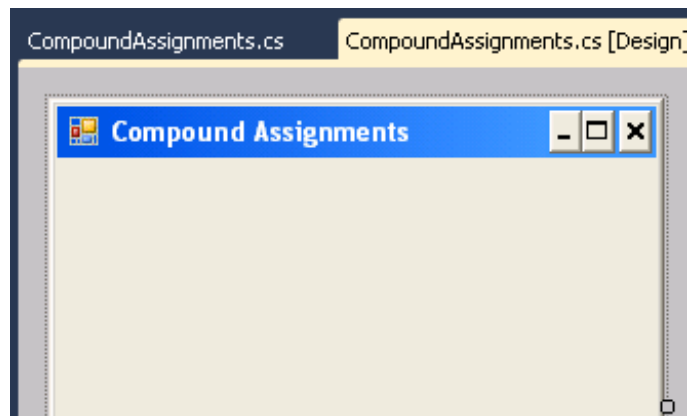


Locate and click on the entry for **Form1.cs** and replace it with **CompoundAssignments.cs**.

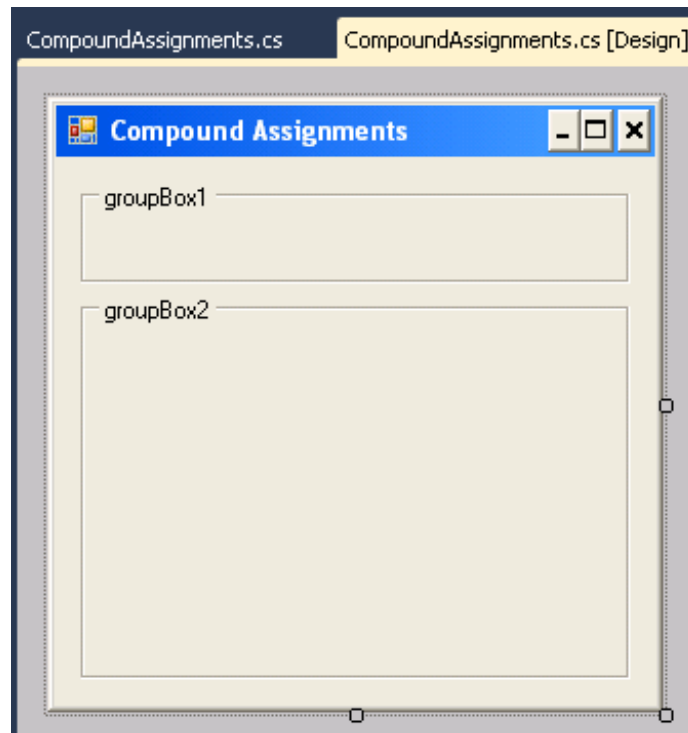


Click on the **Form1** titlebar, and find the Text property in the Properties Window. Change the Text property to **Compound Assignments**. Click  to save your changes.

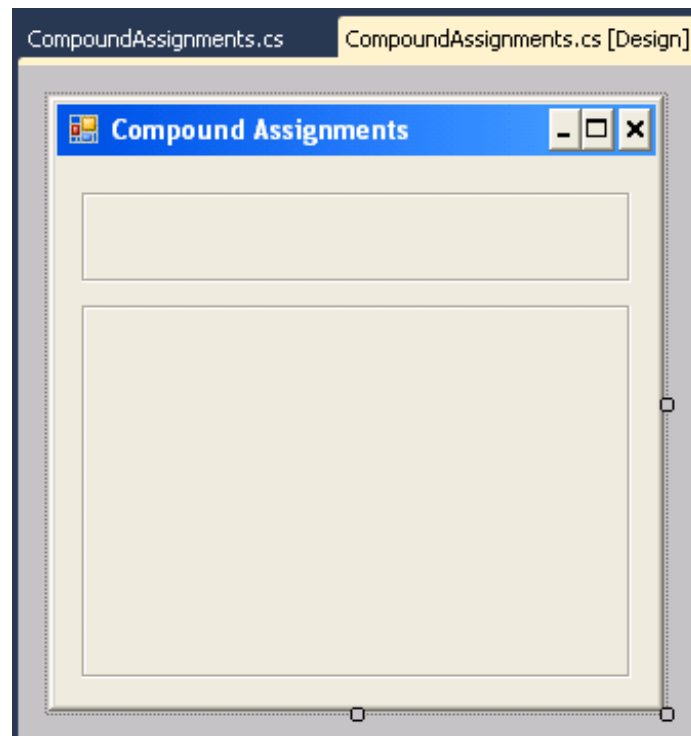




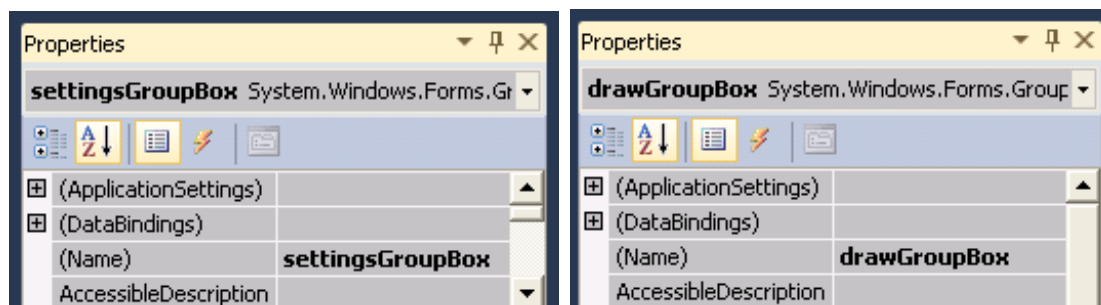
Drag a GroupBox control from the Toolbox onto your form, and resize the GroupBox to take up all of the horizontal space at the top of the form. Add a second GroupBox control from the Toolbox, and place it below the first GroupBox, also taking up all of the horizontal space on the form. Your form should look like this:



For each of the two GroupBoxes, in the Properties Window, clear the Text property. Your form should now look like this:



For the two GroupBoxes, in the Properties Window, change the Name properties to **settingsGroupBox** and **drawGroupBox**, respectively. Save your changes .

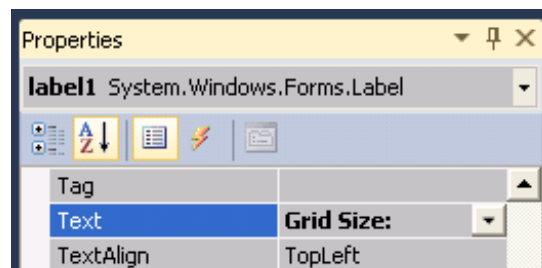


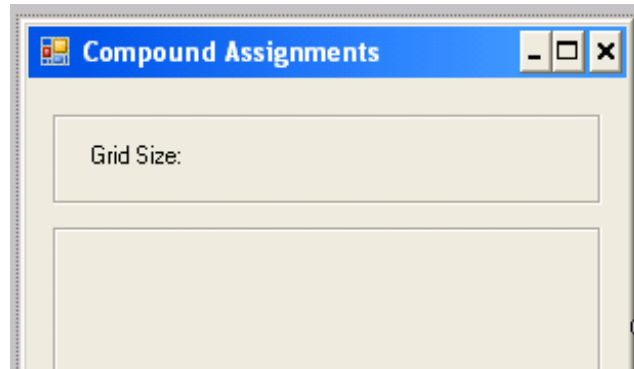
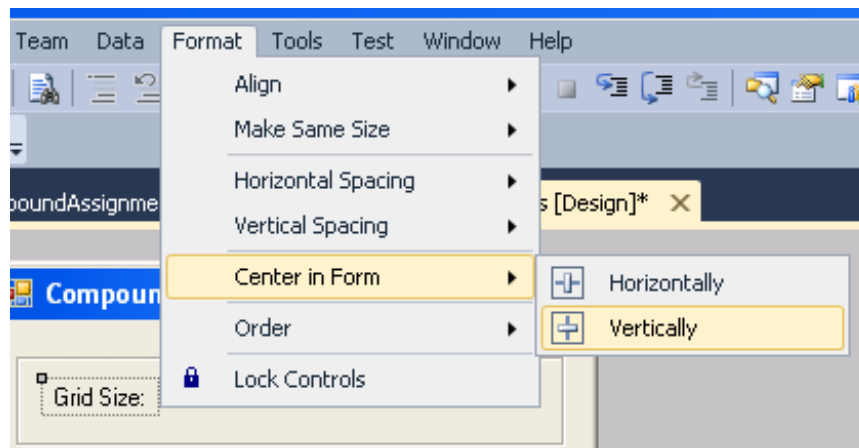
Next, we'll add Label and ComboBox controls. The ComboBox will be a DropDownList that will allow us to select the grid size, such as 4x4, 10x10, etc.

Note

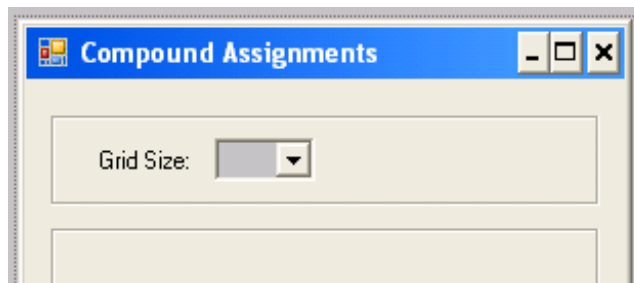
A ComboBox can have one of three styles, set in the DropDownStyle property: Simple, DropDown, and DropDownList.

Drag a **Label** control onto the settingsGroupBox. In the Properties Window, change the Text property to **Grid Size:**. In the top menu, select **Format | Center in Form | Vertically**, to center the label vertically in the GroupBox. Save your changes .

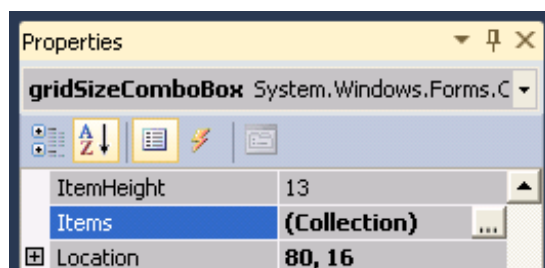
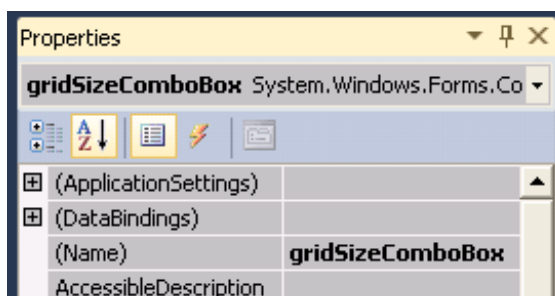


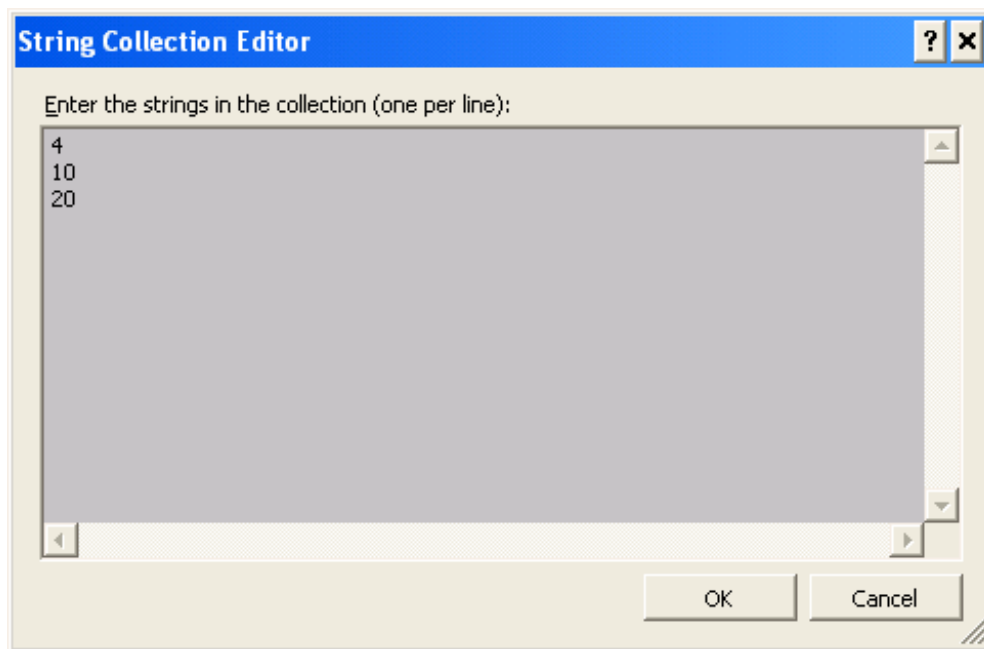


Drag a **ComboBox** control onto the settingsGroupBox and reduce its width:



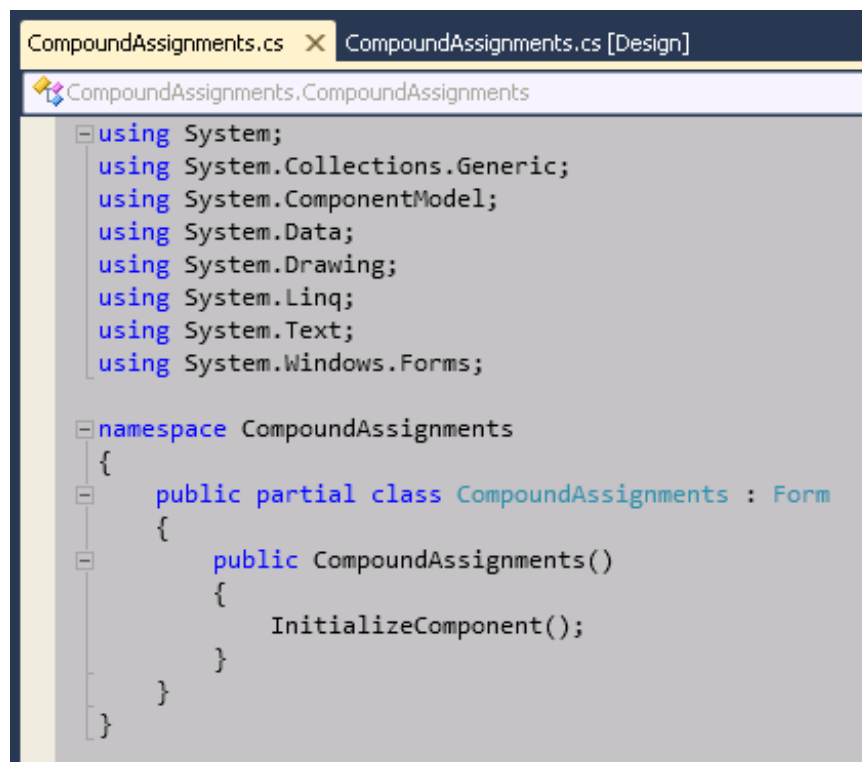
In the top menu, select **Format | Center in Form | Vertically** to center the combobox vertically in the GroupBox. In the Properties Window, change the Name property to **gridSizeComboBox**. Find and select the **Items** property and click on the ellipsis (...). In the String Collection Editor dialog box that appears, enter the numbers 4, 10, and 20, each on a separate line, and click **OK**. Remember to .





Now we'll add code to our project, so we'll need to view the Code Editor for the Form.

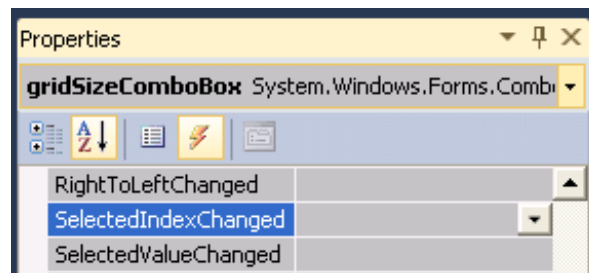
Right-click the **CompoundAssignments.cs** entry in the Solution Explorer, and select **View Code**. The default Windows Form code for your project appears in the Code Editor:



We'll change our code so that any time a user changes the selected item in the `gridSizeComboBox`, the grid is automatically drawn in the `drawGroupBox`.

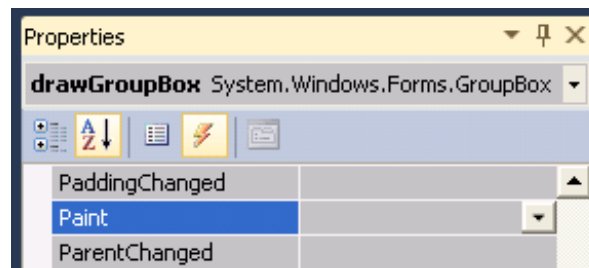
First, let's add the code to respond to whenever the user changes the selected item in the `gridSizeComboBox`. This event is the **SelectedIndexChanged**.

Select the **gridSizeComboBox**, then at the top of the Properties Window, click the Events icon . Locate the **SelectedIndexChanged** event handler, and double-click on the row. You'll again see the code editor, with the new **gridSizeComboBox_SelectedIndexChanged()** event handler method added. Save your changes .



Next, we need to handle the Paint event for the drawComboBox control. The Paint event is called every time a request is made to draw (paint) the contents of a control.

Select the **drawGroupBox**, then in the Properties Window, click the Events icon . Locate the **Paint** event handler, and double-click on the row. You'll again see the code editor with the new **drawGroupBox_Paint()** event handler method handler added. Save your changes .



Modify **CompoundAssignments.cs** as shown below to add code that uses the event handlers and selects the first entry in the gridSizeComboBox. (Again, we've hidden the **using** section by clicking the minus (-) icon to its left.)

CompoundAssignments.cs

```
using ...

namespace CompoundAssignments
{
    public partial class CompoundAssignments : Form
    {
        public CompoundAssignments()
        {
            InitializeComponent();

            // Select the first item in the combo box.
            gridSizeComboBox.SelectedIndex = 0;
        }

        private void DrawGrid(Graphics drawingArea)
        {
        }

        private void gridSizeComboBox_SelectedIndexChanged(object sender, EventArgs e)
        {
            // Notify the group box control that it should redraw itself, which
            // will call the Paint event below.
            drawGroupBox.Refresh();
        }

        private void drawGroupBox_Paint(object sender, PaintEventArgs e)
        {
            // Call the base class OnPaint event to ensure any other controls that
            // at need to be redrawn are also redrawn.
            base.OnPaint(e);
            // Call our draw grid method.
            DrawGrid(e.Graphics);
        }
    }
}
```

This program will not yet draw the grid because we haven't added code in the **DrawGrid** method. Instead, the code we've added prepares everything we need to have the grid be drawn:

OBSERVE: CompoundAssignments.cs

```
.  
.   
.   
    public CompoundAssignments()  
    {  
        InitializeComponent();  
  
        // Select the first item in the combo box.  
        gridSizeComboBox.SelectedIndex = 0;  
    }  
  
    private void DrawGrid(Graphics drawingArea)  
    {  
    }  
  
    private void gridSizeComboBox_SelectedIndexChanged(object sender, EventArgs e)  
    {  
        // Notify the group box control that it should redraw itself, which  
        // will call the Paint event below.  
        drawGroupBox.Refresh();  
    }  
  
    private void drawGroupBox_Paint(object sender, PaintEventArgs e)  
    {  
        // Call our draw grid method.  
        DrawGrid(e.Graphics);  
    }  
.  
.  
.
```

The `gridSizeComboBox.SelectedIndex = 0;` code selects the first value in the `gridSizeComboBox`. The `drawGroupBox.Refresh();` code in the `gridSizeComboBox_SelectedIndexChanged` event handler will force the `drawGroupBox` to redraw itself, which will call the `drawGroupBox_Paint` method. The code in the `drawGroupBox_Paint` method will then call `DrawGrid`. From previous lessons, you may remember that to draw, we need to get a `Graphics` object. The `drawGroupBox_Paint` method includes a `PaintEventArgs` parameter, `e`. The `PaintEventArgs` parameter includes a reference to the drawing surface or area of the `drawGroupBox` control, and we use that reference, `e.Graphics`, as a parameter to our `DrawGrid` method.

Now, modify `CompoundAssignments.cs` as shown below to draw the grid.

Note

As you type the new code, note the extensive use of comments. Think about the comments to help you understand the process and implementation.

CompoundAssignments.cs

```
using ...

namespace CompoundAssignments
{
    public partial class CompoundAssignments : Form
    {
        public CompoundAssignments()
        {
            InitializeComponent();

            // Select the first item in the combo box.
            gridSizeComboBox.SelectedIndex = 0;
        }

        private void DrawGrid(Graphics drawingArea)
        {
            float gridMargin = 10.0F;          // Margin to keep grid away from gro
up box edge.
            int gridSize = 0;                  // Grid size, i.e. 4x4, 10x10, etc.

            // Try to convert the grid size in the combo box to an integer.
            if (Int32.TryParse(gridSizeComboBox.Text, out gridSize))
            {
                // Make sure we have a grid size > 0 before we attempt to draw g
rid.
                if (gridSize > 0)
                {
                    // Create a pen object for drawing.
                    Pen pen = new Pen(Color.Red, 2);

                    // Use the smaller of the drawing area width or height to en
sure a uniform grid spacing.
                    float gridWidth = drawingArea.VisibleClipBounds.Width;
                    if (drawingArea.VisibleClipBounds.Height < gridWidth)
                        gridWidth = drawingArea.VisibleClipBounds.Height;

                    // Subtract the margin value.
                    gridWidth -= gridMargin * 2;

                    // Determine grid cell width.
                    float cellWidth = gridWidth / gridSize;

                    // Set up starting x and y coordinates for both horizontal a
nd vertical lines.
                    float horizontalX = gridMargin;
                    float horizontalY = 0;
                    float verticalX = 0;
                    float verticalY = gridMargin;

                    // The number of lines in grid is equal to the size of the g
rid plus 1, so loop over size of grid + 1,
                    // drawing horizontal and vertical lines.
                    for (int i = 0; i < gridSize + 1; i++)
                    {
                        // Draw horizontal lines.
                        Console.WriteLine("Horizontal: ({0},{1}) to ({2},{3})",
horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gr
idMargin);
                        drawingArea.DrawLine(pen, horizontalX, horizontalY + gri
dMargin, horizontalX + gridWidth, horizontalY + gridMargin);
                        horizontalY += cellWidth;

                        // Draw vertical lines.
                        Console.WriteLine("Vertical: ({0},{1}) to ({2},{3})", ve
rticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
                        drawingArea.DrawLine(pen, verticalX + gridMargin, vertic
```

```

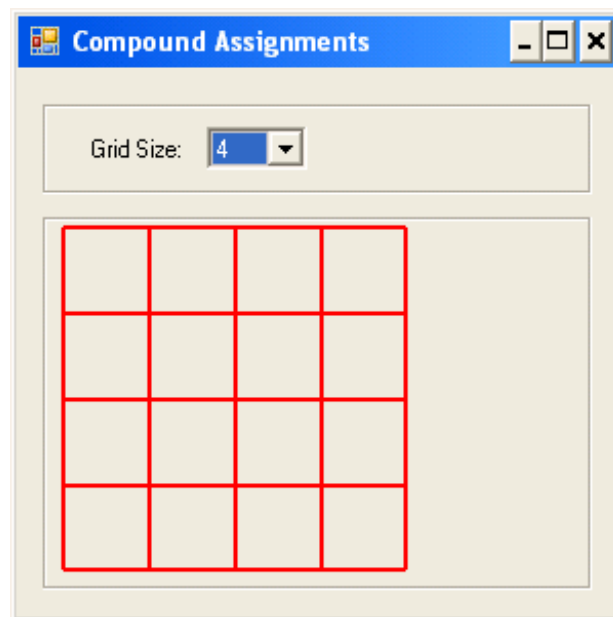
        alY, verticalX + gridMargin, verticalY + gridWidth);
        verticalX += cellWidth;
    }
}

private void gridSizeComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    // Notify the group box control that it should redraw itself, which
    // will call the Paint event below.
    drawGroupBox.Refresh();
}

private void drawGroupBox_Paint(object sender, PaintEventArgs e)
{
    // Call the base class OnPaint event to ensure any other controls that
    // need to be redrawn are also redrawn.
    base.OnPaint(e);
    // Call our draw grid method.
    DrawGrid(e.Graphics);
}
}
}

```

Whew, that was a lot of typing! Trust me, it's worth it.  and  to run the program. You should see something like this:



Let's discuss how this code works.

```

.
.
.
    private void DrawGrid(Graphics drawingArea)
    {
        float gridMargin = 10.0F;          // Margin to keep grid away from gro
up box edge.
        int gridSize = 0;                  // Grid size, i.e. 4x4, 10x10, etc.

        // Try to convert the grid size in the combo box to an integer.
        if (Int32.TryParse(gridSizeComboBox.Text, out gridSize))
        {
            // Make sure we have a grid size > 0 before we attempt to draw g
rid.
            if (gridSize > 0)
            {
                // Create a pen object for drawing.
                Pen pen = new Pen(Color.Red, 2);

                // Use the smaller of the drawing area width or height to en
sure a uniform grid spacing.
                float gridWidth = drawingArea.VisibleClipBounds.Width;
                if (drawingArea.VisibleClipBounds.Height < gridWidth)
                    gridWidth = drawingArea.VisibleClipBounds.Height;

                // Subtract the margin value.
                gridWidth -= gridMargin * 2;

                // Determine grid cell width.
                float cellWidth = gridWidth / gridSize;

                // Set up starting x and y coordinates for both horizontal a
nd vertical lines.
                float horizontalX = gridMargin;
                float horizontalY = 0;
                float verticalX = 0;
                float verticalY = gridMargin;

                // The number of lines in grid is equal to the size of the g
rid plus 1, so loop over size of grid + 1,
                // drawing horizontal and vertical lines.
                for (int i = 0; i < gridSize + 1; i++)
                {
                    // Draw horizontal lines.
                    Console.WriteLine("Horizontal: ({0},{1}) to ({2},{3})",
horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gr
idMargin);
                    drawingArea.DrawLine(pen, horizontalX, horizontalY + gri
dMargin, horizontalX + gridWidth, horizontalY + gridMargin);
                    horizontalY += cellWidth;

                    // Draw vertical lines.
                    Console.WriteLine("Vertical: ({0},{1}) to ({2},{3})", ve
rticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
                    drawingArea.DrawLine(pen, verticalX + gridMargin, vertic
alY, verticalX + gridMargin, verticalY + gridWidth);
                    verticalX += cellWidth;
                }
            }
        }
    }
.
.
.

```

We'll split the explanation of the DrawGrid method into two sections: first describing overall how the code works, and then explaining the compound assignments we're learning in this section of the lessons. First, the overall explanation:

To draw the grid of lines, we use a *for* loop to iterate over each line we'll need to draw. Because we're drawing a square grid, the number of lines to be drawn is the same both vertically and horizontally. All we need to do is set the initial starting point for each line, vertical and horizontal, then add the width of each cell, **cellWidth**, to the starting point as we increment through each grid line. We use the **VisibleClipBounds** method of the **drawingArea** object to determine our drawing surface dimensions and the best width for the grid, **gridWidth**, and the **DrawLine** method to perform the actual line drawing. We also ensure that we can see the edge grid lines by creating a margin, **gridMargin**, and adding that value to the starting coordinates of our line, and subtracting it from our ending coordinates.

Within the DrawGrid method, we use compound assignments three times. The first use is:

```
gridWidth -= gridMargin * 2
```

This subtracts two margin values from the available space for the grid, and is equivalent to:

```
gridWidth = gridWidth - gridMargin * 2
```

The second compound assignment is:

```
horizontalY += cellWidth
```

This increments the horizontal starting coordinates by the width of each grid cell, and is equivalent to:

```
horizontalY = horizontalY + cellWidth
```

The last compound expression is:

```
verticalX += cellWidth
```

This increments the vertical grid line starting coordinates by the width of each cell, and is equivalent to:

```
verticalX = verticalX + cellWidth
```

Before moving on, let's also modify our code to use the *modulus operator (%)*. Remember, the modulus operator returns the remainder of a division operation. The modulus operator is very useful when you want to perform a specific action at a specific frequency, such as every even number, every odd number, or every number divisible by ten. We'll modify our code to show a different grid line color for every odd grid line.

Modify **CompoundAssignments.cs** as shown below to display a different color for each odd-numbered grid line.

CompoundAssignments.cs

```

using ...

namespace CompoundAssignments
{
    public partial class CompoundAssignments : Form
    {
        public CompoundAssignments()
        {
            InitializeComponent();

            // Select the first item in the combo box.
            gridSizeComboBox.SelectedIndex = 0;
        }

        private void DrawGrid(Graphics drawingArea)
        {
            float gridMargin = 10.0F;          // Margin to keep grid away from gro
up box edge.
            int gridSize = 0;                  // Grid size, i.e. 4x4, 10x10, etc.

            // Try to convert the grid size in the combo box to an integer.
            if (Int32.TryParse(gridSizeComboBox.Text, out gridSize))
            {
                // Make sure we have a grid size > 0 before we attempt to draw g
rid.
                if (gridSize > 0)
                {
                    // Create a pen objects for drawing.
                    Pen penredPen = new Pen(Color.Red, 2);
                    Pen bluePen = new Pen(Color.Blue, 2);

                    // Use the smaller of the drawing area width or height to en
sure a uniform grid spacing.
                    float gridWidth = drawingArea.VisibleClipBounds.Width;
                    if (drawingArea.VisibleClipBounds.Height < gridWidth)
                        gridWidth = drawingArea.VisibleClipBounds.Height;

                    // Subtract the margin value.
                    gridWidth -= gridMargin * 2;

                    // Determine grid cell width.
                    float cellWidth = gridWidth / gridSize;

                    // Set up starting x and y coordinates for both horizontal a
nd vertical lines.
                    float horizontalX = gridMargin;
                    float horizontalY = 0;
                    float verticalX = 0;
                    float verticalY = gridMargin;

                    // The number of lines in grid is equal to the size of the g
rid plus 1, so loop over size of grid + 1,
                    // drawing horizontal and vertical lines.
                    for (int i = 0; i < gridSize + 1; i++)
                    {
                        // Select pen, red for even, blue for odd.
                        Pen pen = bluePen;
                        if (i % 2 == 0)
                            pen = redPen;

                        // Draw horizontal lines.
                        Console.WriteLine("Horizontal: ({0},{1}) to ({2},{3})",
horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gr
idMargin);
                        drawingArea.DrawLine(pen, horizontalX, horizontalY + gri
dMargin, horizontalX + gridWidth, horizontalY + gridMargin);
                    }
                }
            }
        }
    }
}

```

```

        horizontalY += cellWidth;

        // Draw vertical lines.
        Console.WriteLine("Vertical: ({0},{1}) to ({2},{3})", ve
rticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
        drawingArea.DrawLine(pen, verticalX + gridMargin, vertic
alY, verticalX + gridMargin, verticalY + gridWidth);
        verticalX += cellWidth;
    }
}

private void gridSizeComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    // Notify the group box control that it should redraw itself, which
    will call the Paint event below.
    drawGroupBox.Refresh();
}

private void drawGroupBox_Paint(object sender, PaintEventArgs e)
{
    // Call the base class OnPaint event to ensure any other controls th
    at need to be redrawn are also redrawn.
    base.OnPaint(e);
    // Call our draw grid method.
    DrawGrid(e.Graphics);
}
}

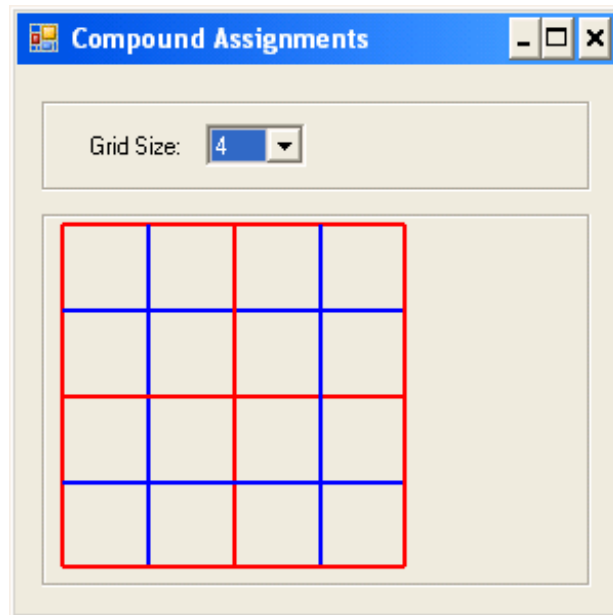
```



and



to run the program.



Let's discuss it.

OBSERVE: CompoundAssignments.cs

```
.
.
.
        // Make sure we have a grid size > 0 before we attempt to draw g
rid.
        if (gridSize > 0)
        {
            // Create pen objects for drawing.
            Pen redPen = new Pen(Color.Red, 2);
            Pen bluePen = new Pen(Color.Blue, 2);

            .
            .
            .

            // The number of lines in grid is equal to the size of the g
rid plus 1, so loop over size of grid + 1,
            // drawing horizontal and vertical lines.
            for (int i = 0; i < gridSize + 1; i++)
            {
                // Select pen, red for even, blue for odd.
                Pen pen = bluePen;
                if (i % 2 == 0)
                    pen = redPen;

                // Draw horizontal lines.
                Console.WriteLine("Horizontal: ({0},{1}) to ({2},{3})",
horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gr
idMargin);
                drawingArea.DrawLine(pen, horizontalX, horizontalY + gri
dMargin, horizontalX + gridWidth, horizontalY + gridMargin);
                horizontalY += cellWidth;

                // Draw vertical lines.
                Console.WriteLine("Vertical: ({0},{1}) to ({2},{3})", ve
rticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
                drawingArea.DrawLine(pen, verticalX + gridMargin, vertic
alY, verticalX + gridMargin, verticalY + gridWidth);
                verticalX += cellWidth;
            }
        }

        .
        .
        .
```

First, in place of the single definition we previously had for one red pen, we define our **redPen** and **bluePen**. Then we define **pen** (an instance of **Pen**), as our **bluePen**—unless **i % 2 == 0**, which means we are in an even-numbered iteration of our **for** loop. This provides for the alternation of colors of **pen** when we draw the lines.

Bitwise and Shift Compound Assignment Operators

These operators are used to access and manipulate individual *bits*. What is a bit? A bit is a *binary digit*, consisting of either 0 or 1. A bit is useful for (among other things) storing the equivalent of false (0) or true (1). The concept of a bit derives from electrical components, where a device could detect the presence of a charge (1), or no charge (0). From this fundamental concept of a bit derives a vast array of computer concepts. For compound assignments, we can use the fact that we can access and manipulate individual bits of a variable.

We will explore using bitwise and shift operators in a later lesson.

C# Operators and Precedence

You may have noted in the previous code, when we expanded one of the compound statements, we had both a subtraction and a multiplication operator within the same statement. How do you know which operator to apply first? From mathematics, you might know that the multiplication operator is used first. The term we use to indicate the correct order to follow is *precedence*. C# has a very large collection of operators, and thus a large *precedence table*. You can

view it here:

[Visual Studio 2010 Precedence Table](#)

From this table, you'll notice that compound assignment operators have less precedence than all of the mathematic operators, and that multiplication has a higher precedence than subtraction, so we would process multiplication expressions first, then subtraction, then the compound assignment operators.

Note You can always enforce a different precedence order using parentheses in your C# expressions.

Increment and Decrement Operators

We've seen the increment and decrement operators before, including in the coding we've just done in this lesson. Where? In the *for* loop code, the last section is the counting expression, and our code included `i++`, which increments `i` by one (`i--`, of course, would decrement `i` by one).

The increment and decrement operators we've seen so far represent the *postfix* versions. The operators follow the expression ("`i`") they modify. These operators can also be *prefix*, indicating that the operators precede the expression they modify. Is there a difference between prefix and postfix? Yes. In the case of prefix, the expression is first incremented or decremented, and then the expression is used, whereas with postfix, the expression is used, and then the value is incremented or decremented. Essentially, if you want the value incremented or decremented before you use it, use prefix. If you want to get the current value, then increment following the statement, use postfix. Let's modify our code to explore the difference.

Modify **CompoundAssignments.cs** as shown below to show the results of using prefix and postfix. Note that the `Console.WriteLine` statements in `DrawGrid` have been commented out.

CompoundAssignments.cs

```

using ...

namespace CompoundAssignments
{
    public partial class CompoundAssignments : Form
    {
        public CompoundAssignments()
        {
            InitializeComponent();

            // Select the first item in the combo box.
            gridSizeComboBox.SelectedIndex = 0;

            // Prefix
            string prefixValues = "0 ";
            int result = 0;
            int value = 1;
            for (int i = 1; i <= 3; i++)
            {
                result += ++value;
                prefixValues += result + " ";
            }

            // Postfix
            string postfixValues = "0 ";
            result = 0;
            value = 1;
            for (int i = 1; i <= 3; i++)
            {
                result += value++;
                postfixValues += result + " ";
            }

            Console.WriteLine("Prefix: " + prefixValues);
            Console.WriteLine("Postfix: " + postfixValues);
        }

        private void DrawGrid(Graphics drawingArea)
        {
            float gridMargin = 10.0F;           // Margin to keep grid away from group box
            int gridSize = 0;                    // Grid size, i.e. 4x4, 10x10, etc.

            // Try to convert the grid size in the combo box to an integer.
            if (Int32.TryParse(gridSizeComboBox.Text, out gridSize))
            {
                // Make sure we have a grid size > 0 before we attempt to draw grid.
                if (gridSize > 0)
                {
                    // Create pen objects for drawing.
                    Pen redPen = new Pen(Color.Red, 2);
                    Pen bluePen = new Pen(Color.Blue, 2);

                    // Use the smaller of the drawing area width or height to ensure a
                    uniform grid spacing.
                    float gridWidth = drawingArea.VisibleClipBounds.Width;
                    if (drawingArea.VisibleClipBounds.Height < gridWidth)
                        gridWidth = drawingArea.VisibleClipBounds.Height;

                    // Subtract the margin value.
                    gridWidth -= gridMargin * 2;

                    // Determine grid cell width.
                    float cellWidth = gridWidth / gridSize;

                    // Set up starting x and y coordinates for both horizontal and vert

```

```

ical lines.

        float horizontalX = gridMargin;
        float horizontalY = 0;
        float verticalX = 0;
        float verticalY = gridMargin;

        // The number of lines in grid is equal to the size of the grid plus
s 1, so loop over size of grid + 1,
        // drawing horizontal and vertical lines.
        for (int i = 0; i < gridSize + 1; i++)
        {
            // Select pen, red for even, blue for odd.
            Pen pen = bluePen;
            if (i % 2 == 0)
                pen = redPen;

            // Draw horizontal lines.
            // Console.WriteLine("Horizontal: ({0},{1}) to ({2},{3})", horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gridMargin);
            drawingArea.DrawLine(pen, horizontalX, horizontalY + gridMargin, horizontalX + gridWidth, horizontalY + gridMargin);
            horizontalY += cellWidth;

            // Draw vertical lines.
            // Console.WriteLine("Vertical: ({0},{1}) to ({2},{3})", verticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
            drawingArea.DrawLine(pen, verticalX + gridMargin, verticalY, verticalX + gridMargin, verticalY + gridWidth);
            verticalX += cellWidth;
        }
    }
}

private void gridSizeComboBox_SelectedIndexChanged(object sender, EventArgs e)
{
    // Notify the group box control that it should redraw itself, which will call the Paint event below.
    drawGroupBox.Refresh();
}

private void drawGroupBox_Paint(object sender, PaintEventArgs e)
{
    // Call the base class OnPaint event to ensure any other controls that need to be redrawn are also redrawn.
    base.OnPaint(e);
    // Call our draw grid method.
    DrawGrid(e.Graphics);
}
}
}

```



and  the program. You'll see this:

```

Output
Show output from: Debug
'CompoundAssignments.'
Prefix: 0 2 5 9
Postfix: 0 1 3 6

```

Let's discuss how this code works.

OBSERVE: CompoundAssignments.cs

```
.
.
.
// Prefix
string prefixValues = "0 ";
int result = 0;
int value = 1;
for (int i = 1; i <= 3; i++)
{
    result += ++value;
    prefixValues += result + " ";
}

// Postfix
string postfixValues = "0 ";
result = 0;
value = 1;
for (int i = 1; i <= 3; i++)
{
    result += value++;
    postfixValues += result + " ";
}

Console.WriteLine("Prefix: " + prefixValues);
Console.WriteLine("Postfix: " + postfixValues);
.
.
.
```

As you can see from the output, the prefix **++value** increments the value *before* it is added to result, as illustrated by the number sequence **0 2 5 9**, whereas the postfix **value++** increments the value *after* it is added to result, as illustrated by the number sequence **0 1 3 6**.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then, select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Relational, Logical, and Conditional Operators

In this lesson we'll explore operators that help us to make decisions through comparisons.

Relational Operators

You've seen *relational operators* before. These include:

- <: Less than
- >: Greater than
- <=: Less than or equal
- >=: Greater than or equal
- ==: Equal
- !=: Not equal

We're going to move on to *logical operators*. We assume you understand the relational operators above; if you have any questions about them, please review the earlier lessons before proceeding.

Boolean and Bitwise Logical Operators

Logical operators are of two kinds: *boolean*, and *bitwise*:

Boolean Logical Operators

- &&: AND
- ||: OR
- !: NOT

Bitwise Logical Operators

- &: AND
- |: OR
- ^: XOR (exclusive OR)
- ~: Complement

In addition to these two categories, when working with logical operators, we also use *true* and **false**.

Body Fat Calculator: Using Relational and Boolean Logical Operators

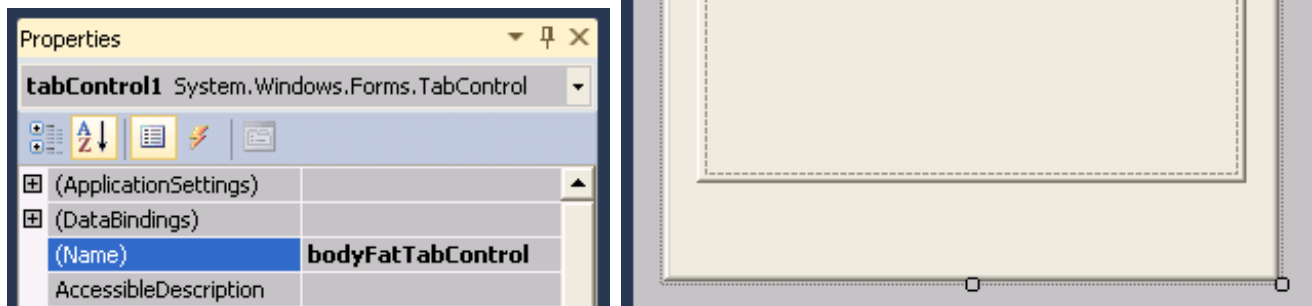
Let's create a project and explore boolean logical and relational operators. We'll save bitwise logical operators for a later lesson where we can devote time to discussing numbering systems, conversions, and bit manipulations.

Note

From now on, we'll assume you know how to create new project, rename the form, and change the form's title bar. We'll just give the name for the new project and you can do the rest.

Create a new Project named **BodyFat Calculator**. Rename the form to **BodyFat Calculator.cs**, and change the title bar to **Body Fat Calculator**.

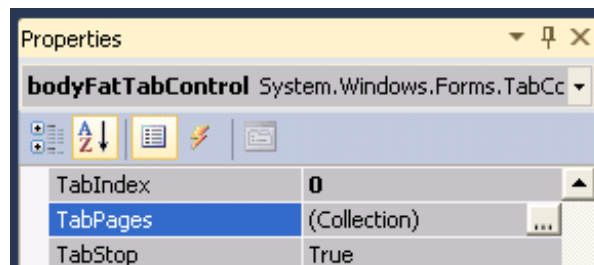
Drag a TabControl control from the Toolbox onto your form, and resize the TabControl to take up most of the space, leaving some room at the bottom for a Button. Change the Name property to **bodyFatTabControl**. Remember to save your changes periodically .



To work with each individual tab, or page, we will need to use the TabPage Collection Editor.

Select the TabPages **Collection** property in the Properties Window, and click on the ellipsis (...). In the TabPages Collection Editor dialog box, in the Members ListBox, click on **tabPage1**. Change the Name property to **calculatorPage**. Change the **Text** property to **Calculator**. Then, again in the TabPages Collection Editor dialog box, in the Members ListBox, click on **tabPage2**. Change the Name property to **chartPage**. Change the Text property to

Chart. Click **OK** and .



TabPage Collection Editor [?] [X]

Members:

0	tabPage1
1	tabPage2

↑
↓

Add Remove

tabPage1 properties:

Accessibility

AccessibleDescript	
AccessibleName	
AccessibleRole	Default

Appearance

BackColor	☐ Control
BackgroundImage	☐ (none)
BackgroundImage	Tile
BorderStyle	None
Cursor	Default
Font	Microsoft Sans Serif,
ForeColor	☒ ControlText
RightToLeft	No
Text	Calculator

OK Cancel

TabPage Collection Editor [?] [X]

Members:

0	calculatorPage
1	tabPage2

↑
↓

Add Remove

calculatorPage properties:

Tag

Design

(Name)	calculatorPage
GenerateMember	True
Locked	False
Modifiers	Private

Focus

CausesValidation	True
------------------	------

Layout

AutoScroll	False
AutoScrollMargin	0, 0
AutoScrollMinSize	0, 0
Margin	3, 3, 3, 3
Padding	2, 2, 2, 2

OK Cancel

TabPage Collection Editor [?] [X]

Members:

0	calculatorPage
1	tabPage2

↑
↓

Add Remove

calculatorPage properties:

RightToLeft		No
Text		Calculator
UseVisualStyleBackColor		True
UseWaitCursor		False
Behavior		
AllowDrop		False
ContextMenuStrip		(none)
ImeMode		NoControl
Data		
(ApplicationSettings)		
(DataBindings)		
Tag		
Design		
(Name)		calculatorPage

OK Cancel

TabPage Collection Editor [?] [X]

Members:

0	calculatorPage
1	chartPage

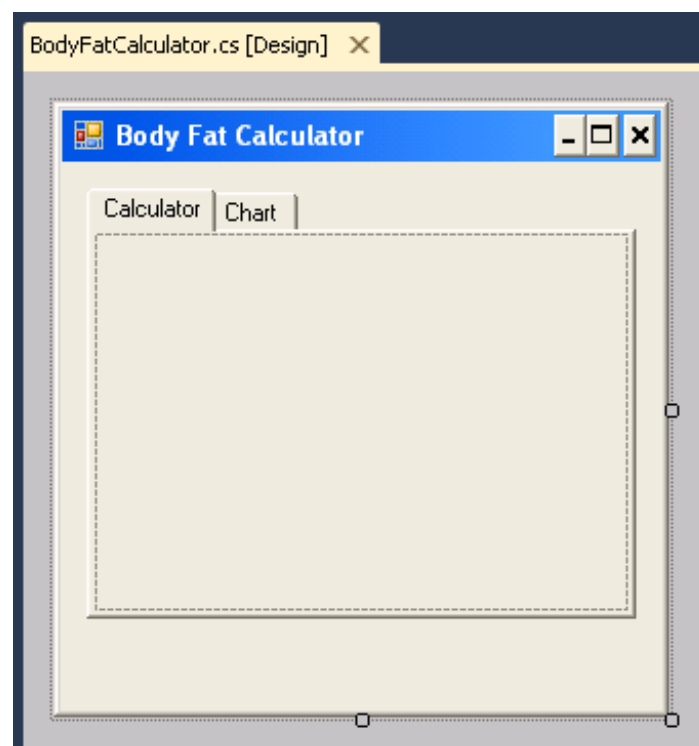
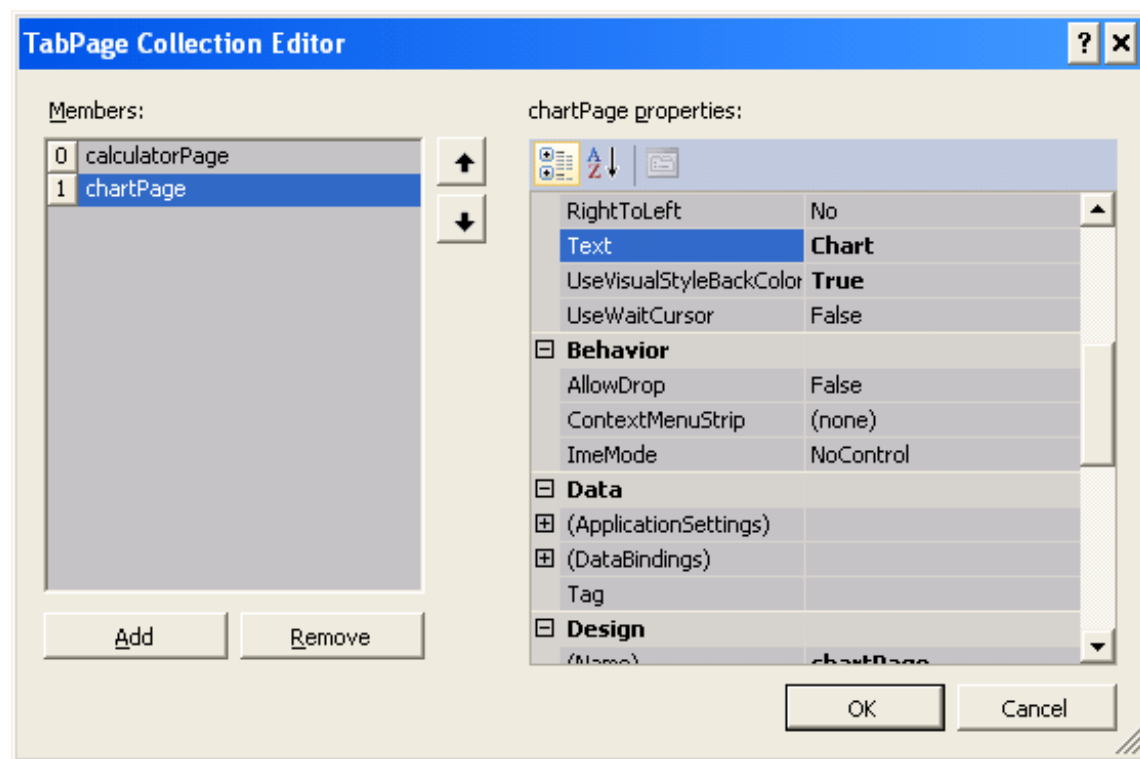
↑
↓

Add Remove

chartPage properties:

Tag		
Design		
(Name)		chartPage
GenerateMember		True
Locked		False
Modifiers		Private
Focus		
CausesValidation		True
Layout		
AutoScroll		False
AutoScrollMargin		0, 0
AutoScrollMinSize		0, 0
Margin		3, 3, 3, 3
Padding		3, 3, 3, 3

OK Cancel



Drag a Button control from the Toolbox onto your form below the TabControl. Change the Name property to **closeButton**. Change the Text property to **Close**. Save your changes .

Next, we'll add five Label controls and five TextBox controls. Rather than individual images for each control that changes the **Name** and **Text** properties, continue to use the pattern you've already established for changing these properties for each control as you add it.

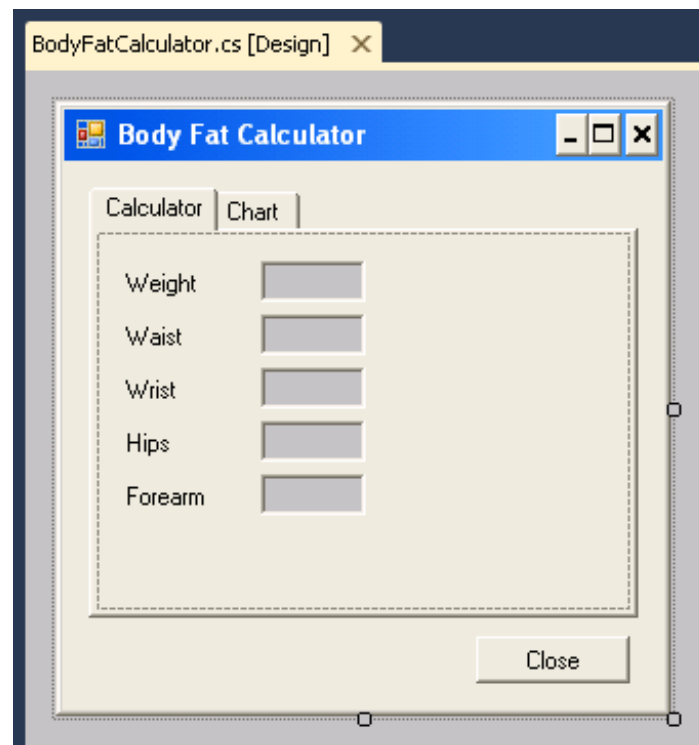
Drag five Label controls and five TextBox controls from the Toolbox onto the Calculator Tab Page. Align the Label controls in one column, and the TextBox controls in a second column. Change the Name and Text properties as follows:

Label Controls		
Original Name	New Name	Text Property

label1	weightLabel	Weight
label2	waistLabel	Waist
label3	wristLabel	Wrist
label4	hipsLabel	Hips
label5	forearmLabel	Forearm

Text Box Controls	
Original Name	New Name
textBox1	weightTextBox
textBox2	waistTextBox
textBox3	wristTextBox
textBox4	hipsTextBox
textBox5	forearmTextBox

When you finish, it should look like this:



Note

If you find yourself spending a lot of time trying to align the Label and TextBox controls to match the image, consider using the Format menu option to help. Start with the TextBoxes, because they take up more vertical space than the Labels; once you have the TextBoxes correctly placed, you can align each Label to the corresponding TextBox. You can use **Vertical Spacing | Make Equal** to equalize the vertical spacing between each TextBox. You can use **Align | Lefts** to make all of the controls line up. Then, select each TextBox, Ctrl-click its corresponding Label control, and use **Align | Middles**.

Next, we'll add two RadioButton controls to select Male or Female, a Button control to perform the calculation, and another Label control to display the results.

Drag two RadioButton controls from the Toolbox onto the Calculator Tab Page. Align each RadioButton under each column. Add a new Button control to the Calculator Tab Page below the RadioButton controls. Add a Label control beneath the Button control. For the Label, change the Visible property to **False**. Again, we'll provide a table of the Name and Text property values, rather than show each individual property.

Note

You'll need to increase the size of the Form, and the size of the TabControl, to have enough space for these changes.

Original Name	New Name	Text Property
radioButton1	femaleRadioButton	Female
radioButton2	maleRadioButton	Male
button1	calculateButton	Calculate Body Fat
label1	bodyFatLabel	Body Fat Percentage:



Your form should look like this:

The screenshot shows a Windows application titled 'Body Fat Calculator'. It features a tab control with two tabs: 'Calculator' and 'Chart'. The 'Calculator' tab is selected, displaying a form with five text input fields labeled 'Weight', 'Waist', 'Wrist', 'Hips', and 'Forearm'. Below these fields are two radio buttons labeled 'Female' and 'Male'. A 'Calculate Body Fat' button is positioned below the radio buttons. At the bottom of the form is a label 'Body Fat Percentage:'. A 'Close' button is located at the bottom right of the window.

Sometimes, you realize you forgot a control, and have to move controls around to squeeze in what you forgot. We need to add another label control between the last TextBox and the RadioButtons that will help us to distinguish which controls are only necessary for females.

Make enough room for a label between the forearmTextBox and the RadioButtons. Drag a Label control into that space on the Calculator Tab Page. Change the label's Name to **requiredForFemaleLabel** and the Text to *** Only required for Female**:

Next, let's add Label controls following each TextBox control to indicate the units, and to add the female-only indicator, an asterisk.

Drag five Label controls from the Toolbox onto the Calculator Tab Page, placing each to the right of each TextBox control. Change the Name and Text properties as shown below:

Original Name	New Name	Text Property
label1	weightUnitLabel	pounds
label2	waistUnitLabel	inches (at navel)
label3	wristUnitLabel	inches (at fullest point)*
label4	hipsUnitLabel	inches (at fullest point)*
label5	forearmUnitLabel	inches (at fullest point)*



Your form should now look like this:

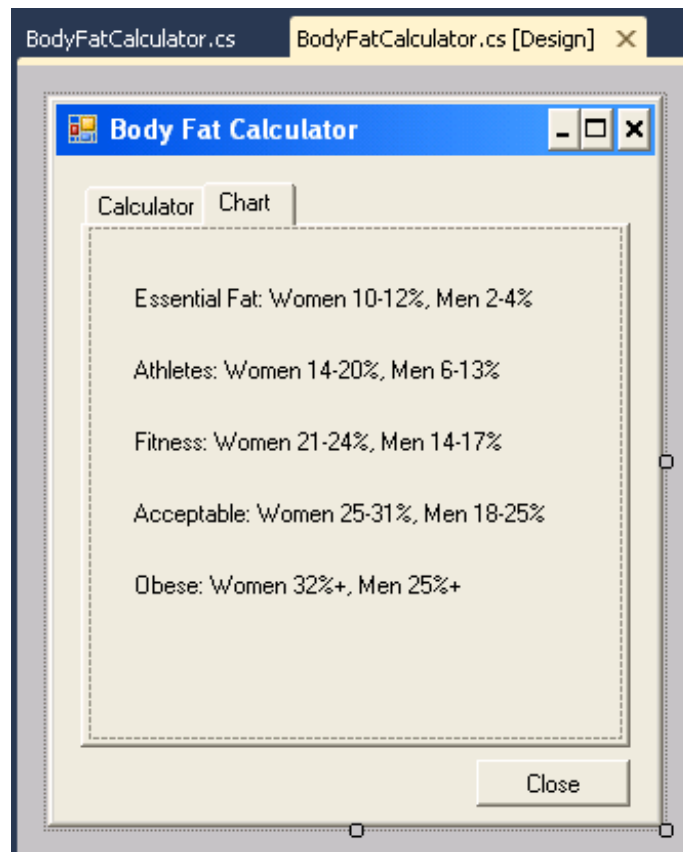
Finally, let's add the Body Fat Chart labels to the Chart Page of the TabControl control.

Drag five Label controls from the Toolbox onto the Chart Tab Page. Change the Name and Text properties as shown below.

Original Name	New Name	Text Property
label1	essentialLabel	Essential Fat: Women 10-12%, Men 2-4%
label2	athletesLabel	Athletes: Women 14-20%, Men 6-13%
label3	fitnessLabel	Fitness: Women 21-24%, Men 14-17%
label4	acceptableLabel	Acceptable: Women 25-31%, Men 18-25%
label5	obeseLabel	Obese: Women 32%+, Men 25%+



Your form should look like this:



So, maybe it's time to get back to the purpose of the lesson: boolean and relational operators! Let's start adding code!

Double-click the **closeButton** to add an event handler for this button. Add code as shown below (some code is omitted from this example for brevity):

BodyFatCalculator.cs

```
using ...

namespace BodyFatCalculator
{
    public partial class BodyFatCalculator : Form
    {
        public BodyFatCalculator()
        {
            InitializeComponent();
            // Default to female.
            femaleRadioButton.Checked = true;
        }

        private double calculateBodyFat(bool isFemale, double weightFactor, double wristFactor,
            double waistFactor, double hipsFactor, double forearmFactor)
        {
            double leanBodyMass = 0;
            double bodyFatWeight = 0;
            double bodyWeight = weightFactor;

            if (isFemale)
            {
                // Calculate for female.
                weightFactor = weightFactor * 0.732 + 8.987;
                wristFactor /= 3.140;
                waistFactor *= 0.157;
                hipsFactor *= 0.249;
                forearmFactor *= 0.434;
                leanBodyMass = weightFactor + wristFactor - waistFactor - hipsFactor +
forearmFactor;
                bodyFatWeight = bodyWeight - leanBodyMass;
            }
            else
            {
                // Calculate for male.
                weightFactor = weightFactor * 1.082 + 94.42;
                waistFactor *= 4.15;
                leanBodyMass = weightFactor - waistFactor;
                bodyFatWeight = bodyWeight - leanBodyMass;
            }

            return bodyFatWeight;
        }

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

Let's discuss how this code works.

OBSERVE: BodyFatCalculator.cs

```

.
.
.

namespace BodyFatCalculator
{
    public partial class BodyFatCalculator : Form
    {
        public BodyFatCalculator()
        {
            InitializeComponent();
            // Default to female.
            femaleRadioButton.Checked = true;
        }

        private double calculateBodyFat(bool isFemale, double weightFactor, double wristFactor,
            double waistFactor, double hipsFactor, double forearmFactor)
        {
            double leanBodyMass = 0;
            double bodyFatWeight = 0;
            double bodyWeight = weightFactor;

            if (isFemale)
            {
                // Calculate for female.
                weightFactor = weightFactor * 0.732 + 8.987;
                wristFactor /= 3.140;
                waistFactor *= 0.157;
                hipsFactor *= 0.249;
                forearmFactor *= 0.434;
                leanBodyMass = weightFactor + wristFactor - waistFactor - hipsFactor + forearmFactor;
                bodyFatWeight = bodyWeight - leanBodyMass;
            }
            else
            {
                // Calculate for male.
                weightFactor = weightFactor * 1.082 + 94.42;
                waistFactor *= 4.15;
                leanBodyMass = weightFactor - waistFactor;
                bodyFatWeight = bodyWeight - leanBodyMass;
            }

            return bodyFatWeight;
        }

        .
        .
        .

        private void closeButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}

```

Again, the comments in the code should help you understand the process used to make the code work. We added a line of code to make sure the **femaleRadioButton** is selected using the **Checked** property. We added a private method named **calculateBodyFat** to isolate the code that calculates the actual body fat. We also added a **this.Close()** method call to enable closing the application using the closeButton.

Within the **calculateBodyFat**, we used **if (isFemale)**. You may remember from previous lessons that an *if* statement must evaluate to a boolean value. **isFemale** is a boolean variable; simply using its name is by definition acceptable within an *if* statement. You can read this expression as **if isFemale is true**. You could also write this expression as **if (isFemale == true)**, or **if (isFemale != false)**.

Next, let's add the event handler and code to perform the body fat calculation.

Go back to the Design tab and double-click the Calculate Body Fat button to add the event handler. Then, add the code shown below:

BodyFatCalculator.cs

```
using ...

namespace BodyFatCalculator
{
    public partial class BodyFatCalculator : Form
    {
        public BodyFatCalculator()
        {
            InitializeComponent();
            // Default to female.
            femaleRadioButton.Checked = true;
        }

        private void closeButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private double calculateBodyFat(bool isFemale, double weightFactor, double wristFactor,
            double waistFactor, double hipsFactor, double forearmFactor)
        {
            ...
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            double weightFactor = 0;
            double waistFactor = 0;
            double wristFactor = 0;
            double hipsFactor = 0;
            double forearmFactor = 0;
            double bodyFatPercentage = 0;
            bool isFemale = false;

            // See if female.
            if (femaleRadioButton.Checked)
                isFemale = true;

            // Attempt to convert TextBox values to doubles, marking unit labels red for failures.
            if (!double.TryParse(weightTextBox.Text, out weightFactor))
                weightUnitLabel.ForeColor = Color.Red;
            if (!double.TryParse(waistTextBox.Text, out waistFactor))
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (!double.TryParse(wristTextBox.Text, out wristFactor))
                    wristUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(hipsTextBox.Text, out hipsFactor))
                    hipsUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(forearmTextBox.Text, out forearmFactor))
                    forearmUnitLabel.ForeColor = Color.Red;
            }

            // Make sure all values are above zero.
            if (weightFactor <= 0)
                weightUnitLabel.ForeColor = Color.Red;
            if (waistFactor <= 0)
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (wristFactor <= 0)
                    wristUnitLabel.ForeColor = Color.Red;
                if (hipsFactor <= 0)
                    hipsUnitLabel.ForeColor = Color.Red;
            }
        }
    }
}
```

```

        if (forearmFactor <= 0)
            forearmUnitLabel.ForeColor = Color.Red;
    }

    // Check for error.
    bool isError = false;
    if (isFemale)
        isError = (weightUnitLabel.ForeColor == Color.Red || waistUnitLabel.ForeColor == Color.Red || wristUnitLabel.ForeColor == Color.Red
            || hipsUnitLabel.ForeColor == Color.Red || forearmUnitLabel.ForeColor == Color.Red);
    else
        isError = (weightUnitLabel.ForeColor == Color.Red || waistUnitLabel.ForeColor == Color.Red);

    if (!isError)
    {
        // Display the body fat Label.
        bodyFatLabel.Visible = true;
        bodyFatLabel.ForeColor = Color.Red;

        // No errors, so calculate and display body fat percentage, formatting
        to two decimal places.
        bodyFatPercentage = (calculateBodyFat(isFemale, weightFactor, wristFactor, waistFactor, hipsFactor, forearmFactor) * 100) / weightFactor;
        bodyFatLabel.Text = "Body Fat Percentage: " + bodyFatPercentage.ToString("#0.00") + "%";
    }
}
}

```

Save your changes , and click  to run the program. Try various kinds of entries, including illegal ones, such as not entering a forearm measurement for a female.

Now let's discuss how it works.

OBSERVE: BodyFatCalculator.cs

```
.
.
.
namespace BodyFatCalculator
{
    public partial class BodyFatCalculator : Form
    {
        public BodyFatCalculator()
        {
            InitializeComponent();
            // Default to female.
            femaleRadioButton.Checked = true;
        }

        private double calculateBodyFat(bool isFemale, double weightFactor, double wristFactor,
            double waistFactor, double hipsFactor, double forearmFactor)
        {
            ...
        }

        private void closeButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            double weightFactor = 0;
            double waistFactor = 0;
            double wristFactor = 0;
            double hipsFactor = 0;
            double forearmFactor = 0;
            double bodyFatPercentage = 0;
            bool isFemale = false;

            // See if female.
            if (femaleRadioButton.Checked)
                isFemale = true;

            // Attempt to convert TextBox values to doubles, marking unit labels red for
            // failures.
            if (!double.TryParse(weightTextBox.Text, out weightFactor))
                weightUnitLabel.ForeColor = Color.Red;
            if (!double.TryParse(waistTextBox.Text, out waistFactor))
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (!double.TryParse(wristTextBox.Text, out wristFactor))
                    wristUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(hipsTextBox.Text, out hipsFactor))
                    hipsUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(forearmTextBox.Text, out forearmFactor))
                    forearmUnitLabel.ForeColor = Color.Red;
            }

            // Make sure all values are above zero.
            if (weightFactor <= 0)
                weightUnitLabel.ForeColor = Color.Red;
            if (waistFactor <= 0)
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (wristFactor <= 0)
                    wristUnitLabel.ForeColor = Color.Red;
                if (hipsFactor <= 0)
```

```

        hipsUnitLabel.ForeColor = Color.Red;
        if (forearmFactor <= 0)
            forearmUnitLabel.ForeColor = Color.Red;
    }

    // Check for error.
    bool isError = false;
    if (isFemale)
        isError = (weightUnitLabel.ForeColor == Color.Red || waistUnitLabel.ForeColor == Color.Red || wristUnitLabel.ForeColor == Color.Red || hipsUnitLabel.ForeColor == Color.Red || forearmUnitLabel.ForeColor == Color.Red);
    else
        isError = (weightUnitLabel.ForeColor == Color.Red || waistUnitLabel.ForeColor == Color.Red);

    if (!isError)
    {
        // Display the body fat Label.
        bodyFatLabel.Visible = true;
        bodyFatLabel.ForeColor = Color.Red;

        // No errors, so calculate and display body fat percentage, formatting to two decimal places.
        bodyFatPercentage = (calculateBodyFat(isFemale, weightFactor, wristFactor, waistFactor, hipsFactor, forearmFactor) * 100) / weightFactor;
        bodyFatLabel.Text = "Body Fat Percentage: " + bodyFatPercentage.ToString("#0.00") + "%";
    }
}

```

We modified the **calculateButton_Click** event handler to try to convert each of the TextBox control values to double values. We change the Unit Label controls to **red** if any of the TextBox control values fail to convert to a valid double or were not above 0. Finally, we check that we don't have any errors, and if not, we go ahead and calculate the percentage of body fat.

Within the **calculateButton_Click** event handler, we use a few more relational and boolean operators. We use the **TryParse** method of *double* to attempt to convert each of the TextBox control values. We use the **!** operator to indicate we want to check if the **TryParse** method returned *false*. Literally, the **!** reverses the value, so you could read it as *if the result of TryParse is NOT true*.

After attempting to convert each TextBox control value, we then test to see that each value is greater than zero using the **<=** relational operator. We also test to see if any of the Unit Label controls have their ForeColor set to **Color.Red** using the boolean operator **||** and relational operator **==**. The *isError* boolean variable will be true if any of the Unit Label controls have their ForeColor set to **Color.Red**.

Read through the code again, looking for any other use of relational or boolean operators.

Before finishing this section, we do want to add a bit more code, just for fun, to support the display of a *ToolTip* to show the error message. A *ToolTip* is a popup that appears if the mouse pointer hovers over a control.

Modify the code as shown below.

BodyFatCalculator.cs

```
using ...

namespace BodyFatCalculator
{
    public partial class BodyFatCalculator : Form
    {
        .
        .
        .
        private void calculateButton_Click(object sender, EventArgs e)
        {
            double weightFactor = 0;
            double waistFactor = 0;
            double wristFactor = 0;
            double hipsFactor = 0;
            double forearmFactor = 0;
            double bodyFatPercentage = 0;
            bool isFemale = false;

            // See if female.
            if (femaleRadioButton.Checked)
                isFemale = true;

            // Set all unit labels to black.
            weightUnitLabel.ForeColor = Color.Black;
            waistUnitLabel.ForeColor = Color.Black;
            wristUnitLabel.ForeColor = Color.Black;
            hipsUnitLabel.ForeColor = Color.Black;
            forearmUnitLabel.ForeColor = Color.Black;

            // Attempt to convert TextBox values to doubles, marking unit labels red fo
r failures.
            if (!double.TryParse(weightTextBox.Text, out weightFactor))
                weightUnitLabel.ForeColor = Color.Red;
            if (!double.TryParse(waistTextBox.Text, out waistFactor))
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (!double.TryParse(wristTextBox.Text, out wristFactor))
                    wristUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(hipsTextBox.Text, out hipsFactor))
                    hipsUnitLabel.ForeColor = Color.Red;
                if (!double.TryParse(forearmTextBox.Text, out forearmFactor))
                    forearmUnitLabel.ForeColor = Color.Red;
            }

            // Make sure all values are above zero.
            if (weightFactor <= 0)
                weightUnitLabel.ForeColor = Color.Red;
            if (waistFactor <= 0)
                waistUnitLabel.ForeColor = Color.Red;
            if (isFemale)
            {
                if (wristFactor <= 0)
                    wristUnitLabel.ForeColor = Color.Red;
                if (hipsFactor <= 0)
                    hipsUnitLabel.ForeColor = Color.Red;
                if (forearmFactor <= 0)
                    forearmUnitLabel.ForeColor = Color.Red;
            }

            // See if there were any errors by checking Unit label Forecolor.
            // Weight
            if (weightUnitLabel.ForeColor == Color.Red)
                setToolTip(weightLabel, weightTextBox, weightUnitLabel, "Weight (pounds
) must be numeric and above 0.");
```

```

        else
            setToolTip(weightLabel, weightTextBox, weightUnitLabel, "");
        // Waist
        if (waistUnitLabel.ForeColor == Color.Red)
            setToolTip(waistLabel, waistTextBox, waistUnitLabel, "Waist (inches) must be numeric and above 0.");
        else
            setToolTip(waistLabel, waistTextBox, waistUnitLabel, "");
        if (isFemale)
        {
            // Wrist
            if (wristUnitLabel.ForeColor == Color.Red)
                setToolTip(wristLabel, wristTextBox, wristUnitLabel, "Wrist (inches) must be numeric and above 0.");
            else
                setToolTip(wristLabel, wristTextBox, wristUnitLabel, "");
            // Hips
            if (hipsUnitLabel.ForeColor == Color.Red)
                setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, "Hips (inches) must be numeric and above 0.");
            else
                setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, "");
            // Forearm
            if (forearmUnitLabel.ForeColor == Color.Red)
                setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, "Forearm (inches) must be numeric and above 0.");
            else
                setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, "");
        }

        .
        .
        .

        if (!isError)
        {
            // Display the body fat Label.
            bodyFatLabel.Visible = true;
            bodyFatLabel.ForeColor = Color.Red;

            // No errors, so calculate and display body fat percentage, formatting to two decimal places.
            bodyFatPercentage = (calculateBodyFat(isFemale, weightFactor, wristFactor, waistFactor, hipsFactor, forearmFactor) * 100) / weightFactor;
            bodyFatLabel.Text = "Body Fat Percentage: " + bodyFatPercentage.ToString("#0.00") + "%";
        }
        else
        {
            // There were errors.
            bodyFatLabel.Visible = true;
            bodyFatLabel.ForeColor = Color.Red;
            bodyFatLabel.Text = "Hover mouse over error to see error message";
        }
    }

    private double calculateBodyFat(bool isFemale, double weightFactor, double wristFactor,
        double waistFactor, double hipsFactor, double forearmFactor)
    {
        ...
    }

    private void setToolTip(Control labelControl, Control textBoxControl, Control labelUnitControl, string toolText)
    {
        // Create a tooltip object.
        ToolTip tp = new ToolTip();

```

```

        // Set tooltip for each control.
        tp.SetToolTip(labelControl, toolText);
        tp.SetToolTip(textBoxControl, toolText);
        tp.SetToolTip(labelUnitControl, toolText);
    }
}

```

 and
  it. It should look something like this:

Let's discuss how this code works.

OBSERVE: BodyFatCalculator.cs

```
.
.
.

private void calculateButton_Click(object sender, EventArgs e)
{
    ...

    // Set all unit labels to black.
    weightUnitLabel.ForeColor = Color.Black;
    waistUnitLabel.ForeColor = Color.Black;
    wristUnitLabel.ForeColor = Color.Black;
    hipsUnitLabel.ForeColor = Color.Black;
    forearmUnitLabel.ForeColor = Color.Black;

    ...

    // See if there were any errors by checking Unit label Forecolor.
    // Weight
    if (weightUnitLabel.ForeColor == Color.Red)
        setToolTip(weightLabel, weightTextBox, weightUnitLabel, "Weight (pounds
) must be numeric and above 0.");
    else
        setToolTip(weightLabel, weightTextBox, weightUnitLabel, "");
    ...

    if (!isError)
    {
        ...
    }
    else
    {
        // There were errors.
        bodyFatLabel.Visible = true;
        bodyFatLabel.ForeColor = Color.Red;
        bodyFatLabel.Text = "Hover mouse over error to see error message";
    }
}

private double calculateBodyFat(bool isFemale, double weightFactor, double wris
tFactor,
    double waistFactor, double hipsFactor, double forearmFactor)
{
    ...
}

private void setToolTip(Control labelControl, Control textBoxControl, Control l
abelUnitControl, string toolText)
{
    // Create a tooltip object.
    ToolTip tp = new ToolTip();

    // Set tooltip for each control.
    tp.SetToolTip(labelControl, toolText);
    tp.SetToolTip(textBoxControl, toolText);
    tp.SetToolTip(labelUnitControl, toolText);
}
}
```

These final additions to the code first make sure that the ForeColor of all Unit Labels is reset Color.Black. Then, we call the **setToolTip** method to set all three of the related controls to have a ToolTip if there's an error. And of course we created the **setToolTip** method.

Conditional Operator

The *Conditional Operator* in C#, also known as the *ternary operator* because it has three components, is fundamentally a shortcut, or compact, syntax, for the *if..else* construct. What does it look like?

Conditional Operator Syntax
<code>Condition ? First expression : Second expression</code>

If the **Condition** component is *true*, then the **First expression** is evaluated, and becomes the result. If the **Condition** is *false*, then the **Second expression** is evaluated, and becomes the result.

An *if..else* structure for the same logic would look like this:

Conditional Operator Syntax
<pre>if (Condition) First expression else Second expression</pre>

The Conditional Operator is often used to eliminate the more lengthy *if..else* syntax, especially when the result is to be used for output. In fact, we'll replace some of our code with this operator. Before moving on, scan through the code, and see if you can determine what *if..else* constructs might be replaced.

Modify the code as shown below.

BodyFatCalculator.cs

```
.
.
.
        // See if there were any errors by checking Unit label Forecolor.
        // Weight
        setToolTip(weightLabel, weightTextBox, weightUnitLabel, weightUnitLabel.ForeColor == Color.Red ? "Weight (pounds) must be numeric and above 0." : "");
        if (weightUnitLabel.ForeColor == Color.Red)
            setToolTip(weightLabel, weightTextBox, weightUnitLabel, "Weight (pounds) must be numeric and above 0.");
        else
            setToolTip(weightLabel, weightTextBox, weightUnitLabel, "");
        // Waist
        setToolTip(waistLabel, waistTextBox, waistUnitLabel, waistUnitLabel.ForeColor == Color.Red ? "Waist (inches) must be numeric and above 0." : "");
        if (waistUnitLabel.ForeColor == Color.Red)
            setToolTip(waistLabel, waistTextBox, waistUnitLabel, "Waist (inches) must be numeric and above 0.");
        else
            setToolTip(waistLabel, waistTextBox, waistUnitLabel, "");
        if (isFemale)
        {
            // Wrist
            setToolTip(wristLabel, wristTextBox, wristUnitLabel, wristUnitLabel.ForeColor == Color.Red ? "Wrist (inches) must be numeric and above 0." : "");
            if (wristUnitLabel.ForeColor == Color.Red)
                setToolTip(wristLabel, wristTextBox, wristUnitLabel, "Wrist (inches) must be numeric and above 0.");
            else
                setToolTip(wristLabel, wristTextBox, wristUnitLabel, "");
            // Hips
            setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, hipsUnitLabel.ForeColor == Color.Red ? "Hips (inches) must be numeric and above 0." : "");
            if (hipsUnitLabel.ForeColor == Color.Red)
                setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, "Hips (inches) must be numeric and above 0.");
            else
                setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, "");
            // Forearm
            setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, forearmUnitLabel.ForeColor == Color.Red ? "Forearm (inches) must be numeric and above 0." : "");
            if (forearmUnitLabel.ForeColor == Color.Red)
                setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, "Forearm (inches) must be numeric and above 0.");
            else
                setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, "");
        }
.
.
.
```

Save your changes , and click  to run the program. You should see identical results.

Let's discuss how this code works.

OBSERVE: BodyFatCalculator.cs

```
.  
.   
.   
    // See if there were any errors by checking Unit label Forecolor.  
    // Weight  
    setToolTip(weightLabel, weightTextBox, weightUnitLabel, weightUnitLabel.Fore  
eColor == Color.Red ? "Weight (pounds) must be numeric and above 0." : "");  
    setToolTip(waistLabel, waistTextBox, waistUnitLabel, waistUnitLabel.ForeCol  
or == Color.Red ? "Waist (inches) must be numeric and above 0." : "");  
    if (isFemale)  
    {  
        // Wrist  
        setToolTip(wristLabel, wristTextBox, wristUnitLabel, wristUnitLabel.For  
eColor == Color.Red ? "Wrist (inches) must be numeric and above 0." : "");  
        // Hips  
        setToolTip(hipsLabel, hipsTextBox, hipsUnitLabel, hipsUnitLabel.ForeCol  
or == Color.Red ? "Hips (inches) must be numeric and above 0." : "");  
        // Forearm  
        setToolTip(forearmLabel, forearmTextBox, forearmUnitLabel, forearmUnitL  
abel.ForeColor == Color.Red ? "Forearm (inches) must be numeric and above 0." : "");  
    }  
.   
.   
.   

```

This version certainly makes the section of code that set up the ToolTips much more compact! Rather than using the *if..else* syntax, we're able to just call the `setToolTip` method, but now the **toolText** parameter uses the Conditional Operator. We use `weightUnitLabel.ForeColor == Color.Red` as the **Condition** component. If the **Condition** is *true*, we set the **toolText** parameter to **"Weight (pounds) must be numeric and above 0."**, and if the **Condition** is *false*, we set the **toolText** parameter to **" "**. Very nice!

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Branching Statements: Conditional and Unconditional

In this lesson, we'll further explore the important concept of branching. We know that code is processed sequentially in a method, although depending on the code, the program may jump to another method in the same class, another method in another class, or even execute a C# or .Net-provided component or code. We've already created programs that use those kinds of branching, but let's take a more thorough look at this essential programming concept.

Unconditional Branching

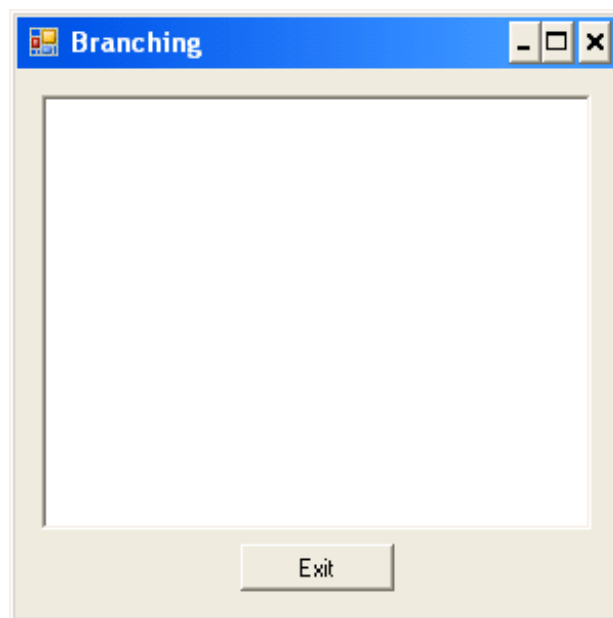
C# provides the following *Unconditional Branching* techniques:

- **methodName**: jumps to the named method.
- **return**: terminates execution of a method and returns control to calling method, and may return a value if provided.
- **break**: terminates the closest enclosing statement (*while*, *do*, *for*, *foreach*), passing control to the statement that follows the terminated enclosing statement.
- **goto**: transfers program control directly to a labeled statement or *switch case* statement.
- **continue**: passes control to the next iteration of the enclosing iteration statement (*while*, *do*, *for*, *foreach*).
- **throw**: signals the occurrence of an anomalous situation (or *exception*) and passes control to an exception-handling routine if available.

Some of these techniques and keywords you've seen and used before, such as methods, *return*, and *break*, but the rest are new. Let's take a look at each of these unconditional branching techniques a bit more. How? We'll create a new project, of course! However, some of the unconditional branching options available should only be used sparingly, and usually in only certain situations. We'll indicate those situations, and any issues to watch out for, as we work with each option. All program output will be sent to a Listbox. Unconditional branching statements are also frequently referred to as *jump* statements. We will not have a section about methods here; we'll devote an entire lesson to this important branching technique later.

Select **File | New | Project**, change the Name to **Branching**, and click **OK**. Find the entry for **Form1.cs** and change it to **Branching.cs**. Change the Form1 Form title bar's Text property in the Properties Window to **Branching**. Click .

Modify the form to look like this:



Change each control's Name property to match the Name Property column below, and Text property to match the Text Property column. Arrange the controls similar to the image. When you finish, click  to save your changes.

Object	Name Property	Text Property
--------	---------------	---------------

ListBox	outputListBox	
Button	exitButton	Exit

Add an event handler for the **exitButton** control by double-clicking on the Exit Button to automatically generate the default event handler code. Studio generates an appropriate name for the event handler that consists of the name of the control, an underscore separator, and the event name.

Modify **Branching.cs** as shown below. The event handler code was added previously.

Branching.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Branching
{
    public partial class Branching : Form
    {
        public Branching()
        {
            InitializeComponent();

            // Unconditional branching methods.
            // return
            doReturnNoValue();
            doReturnWithValue();

            // break
            doBreak();

            // goto
            doGotoWithLabel();
            doGotoWithSwitch();

            // continue
            doContinue();

            // throw
            doThrow();
        }

        private void doReturnNoValue()
        {
            // Get method name.
            outputListBox.Items.Add("> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
        }

        private bool doReturnWithValue()
        {
            bool result = false;

            // Get method name.
            outputListBox.Items.Add("> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

            return result;
        }

        private void doBreak()
        {
            // Get method name.
            outputListBox.Items.Add("> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
        }

        private void doGotoWithLabel()
        {
            // Get method name.
            outputListBox.Items.Add("> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
        }
    }
}
```

```

    }

    private void doGotoWithSwitch()
    {
        // Get method name.
        outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
    }

    private void doContinue()
    {
        // Get method name.
        outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
    }

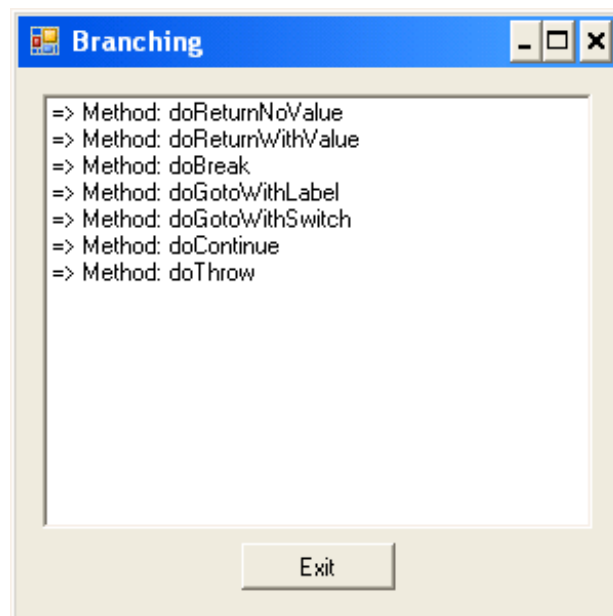
    private void doThrow()
    {
        // Get method name.
        outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);
    }

    private void exitButton_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
}

```

This first code is just a template to hold each section as we implement and discuss the various unconditional branching techniques. As you can see, we added calls to methods that will cover each of the techniques we will discuss. We did add one interesting line to each method: `System.Reflection.MethodInfo.GetCurrentMethod().Name` gets the name of the method we're in.

Click  and . You should see the following output:



Now, let's add and discuss each branching section.

Modify **Branching.cs** as shown below to add code to the **doReturnNoValue()** method. (We'll understand if you cut and paste—just for this lesson!—the lengthy comments.) Note that only the **doReturnNoValue()** method is shown—no other code should change.

Branching.cs

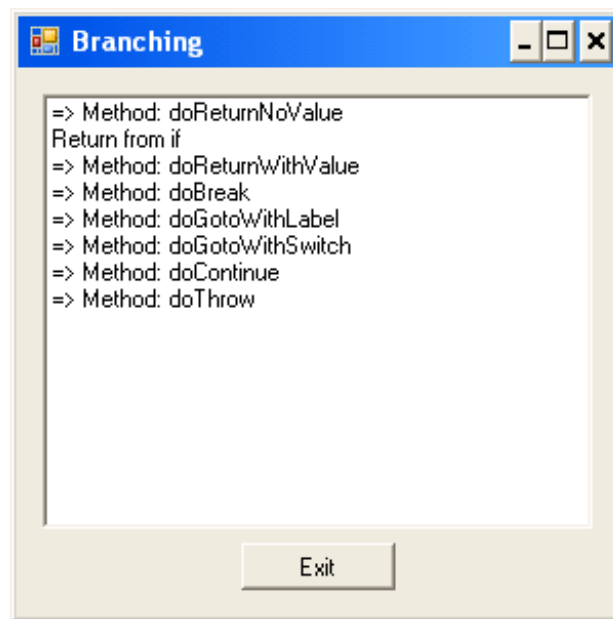
```
.
.
.
private void doReturnNoValue()
{
    // Get method name.
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

    // If your method has no return type, you normally would not use a return statement
    // as your program will exit normally. Good programming practice is to have a single exit
    // point from your code; however, if you need to immediately exit a method, especially
    // with deeply nested if statements, you might consider using a return.

    int testVariable = 3;
    int compareVariable = 4;

    if (compareVariable > testVariable)
    {
        if (testVariable < 10)
        {
            outputListBox.Items.Add("Return from if");
            return;
        }
        else
        {
            // Do some more checking.
            switch (testVariable)
            {
                case 1:
                    outputListBox.Items.Add("Return from switch case 1");
                    return;
                case 2:
                    outputListBox.Items.Add("Return from switch case 2");
                    return;
                default:
                    testVariable++;
                    break;
            }
        }
    }
    outputListBox.Items.Add("Normal method termination");
}
.
.
.
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.  
.   
.   
private void doReturnNoValue()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // If your method has no return type...  
  
    int testVariable = 3;  
    int compareVariable = 4;  
  
    if (compareVariable > testVariable)  
    {  
        if (testVariable < 10)  
        {  
            outputListBox.Items.Add("Return from if");  
            return;  
        }  
        else  
        {  
            // Do some more checking.  
            switch (testVariable)  
            {  
                case 1:  
                    outputListBox.Items.Add("Return from switch case 1");  
                    return;  
                case 2:  
                    outputListBox.Items.Add("Return from switch case 2");  
                    return;  
                default:  
                    testVariable++;  
                    break;  
            }  
        }  
    }  
    outputListBox.Items.Add("Normal method termination");  
}  
.   
.   
.
```

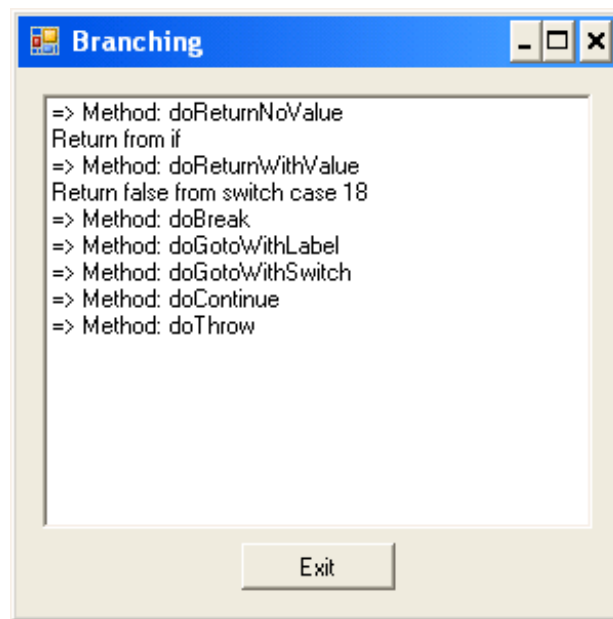
This method demonstrates how to use the **return** keyword from anywhere within a method, without returning a value. The **doReturnNoValue()** specifies a return type of **void**, so no value may be returned. This code illustrates a possible reason to use a **return** statement with deeply nested *if* statements, although the example isn't that deeply nested. Notice that within the **switch**'s **case** statements, the **return** statement will immediately exit the method, so you do not need a **break** statement.

Modify **Branching.cs** as shown below to add the code to the **doReturnWithValue()** method. Note that only this method is shown; no other code should change.

Branching.cs

```
.  
.   
.   
private bool doReturnWithValue()  
{  
    bool result = false;  
  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // A return statement is required when a method has a return type (bool in this example). You  
    // can have multiple return statements, with different values, although good programming practice  
    // is to have a single exit point from your code.  
  
    int myAge = 18;  
  
    if (myAge > 21)  
        result = true;  
    else if (myAge >= 18)  
    {  
        // Do some more checking.  
        switch (myAge)  
        {  
            case 18:  
                outputListBox.Items.Add("Return false from switch case 18");  
                return false;  
            case 19:  
                outputListBox.Items.Add("Return true from switch case 19");  
                return true;  
            case 20:  
                outputListBox.Items.Add("Return false from switch case 20");  
                return false;  
            default:  
                myAge++;  
                break;  
        }  
    }  
  
    outputListBox.Items.Add("Normal method termination, return value is " + result);  
    return result;  
}  
.   
.   
.
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.  
.   
.   
private bool doReturnWithValue()  
{  
    bool result = false;  
  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // A return statement is required when...  
  
    int myAge = 18;  
  
    if (myAge > 21)  
        result = true;  
    else if (myAge >= 18)  
    {  
        // Do some more checking.  
        switch (myAge)  
        {  
            case 18:  
                outputListBox.Items.Add("Return false from switch case 18");  
                return false;  
            case 19:  
                outputListBox.Items.Add("Return true from switch case 19");  
                return true;  
            case 20:  
                outputListBox.Items.Add("Return false from switch case 20");  
                return false;  
            default:  
                myAge++;  
                break;  
        }  
    }  
  
    outputListBox.Items.Add("Normal method termination, return value is " + result);  
    return result;  
}  
.   
.   
.
```

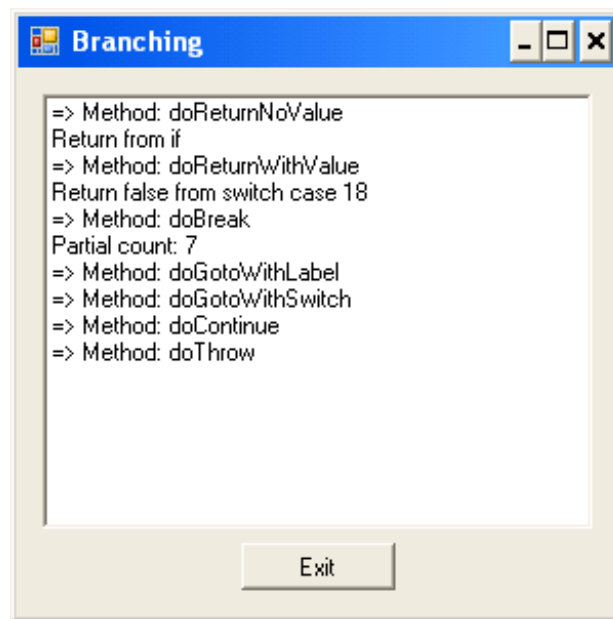
This method demonstrates how to use the **return** keyword from anywhere in a method, even when returning a value (in this case, we use the literal **true** or **false**, or the **result** variable, which is set to true or false depending on the value of **myAge**). The **doReturnWithValue()** specifies a return type of **bool**, so all **returns** must return a **bool** regardless of where the **return** is used.

Modify **Branching.cs** as shown below to add the code to the **doBreak()** method. Only this method is shown; no other code should change.

Branching.cs

```
.  
.br/>.br/>private void doBreak()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // A break statement exits an enclosing loop, such as a while, do, for, or foreach,  
    // or a switch statement.  
    // It must be used with one of these C# constructs. Other than with switch statements,  
    // excessive use of  
    // break statements within loops can lead to confusion.  
  
    int partialCount = 0;  
  
    // while (or do) - watch out for infinite loops.  
    while (true)  
    {  
        if (partialCount > 5)  
            break;  
        partialCount++;  
    }  
  
    // for (or foreach)  
    for (int i = 0; i < 10; i++)  
    {  
        if (i < 5)  
            break;  
        partialCount++;  
    }  
  
    // switch  
    switch (partialCount)  
    {  
        case 5:  
            partialCount *= 5;  
            break;  
        case 10:  
            partialCount *= 10;  
            break;  
        default:  
            partialCount++;  
            break;  
    }  
  
    outputListBox.Items.Add("Partial count: " + partialCount);  
}  
.br/>.br/.
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.  
.   
.   
private void doBreak()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // A break statement exits an enclosing loop...  
  
    int partialCount = 0;  
  
    // while (or do) - watch out for infinite loops.  
    while (true)  
    {  
        if (partialCount > 5)  
            break;  
        partialCount++;  
    }  
  
    // for (or foreach)  
    for (int i = 0; i < 10; i++)  
    {  
        if (i < 5)  
            break;  
        partialCount++;  
    }  
  
    // switch  
    switch (partialCount)  
    {  
        case 5:  
            partialCount *= 5;  
            break;  
        case 10:  
            partialCount *= 10;  
            break;  
        default:  
            partialCount++;  
            break;  
    }  
  
    outputListBox.Items.Add("Partial count: " + partialCount);  
}  
.   
.   
.
```

This method demonstrates the multiple ways you can use the **break** keyword, exiting looping structures or the **switch** structure. The **break** immediately exits the enclosing structure.

Modify **Branching.cs** as shown below to add the code to the **doGotoWithLabel()** method. Note that only this method is shown; no other code should change.

Branching.cs

```
.
.
.
private void doGotoWithLabel()
{
    // Get method name.
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

    // A goto statement transfers program control directly to another section of the code identified
    // by a label. Goto statements can transfer control out of a block, such as below where the code
    // transfers control out of an if statement. You can not transfer control into a block. Remember,
    // a block is indicated by curly braces. Even though in this example the labeling is pretty clear,
    // other mechanisms exist that would not require the use of goto and labels. Also, labels may
    // easily be far removed from any goto statements, thus obscuring the flow and logic of the code,
    // so good programming practice discourages the use of goto statements.

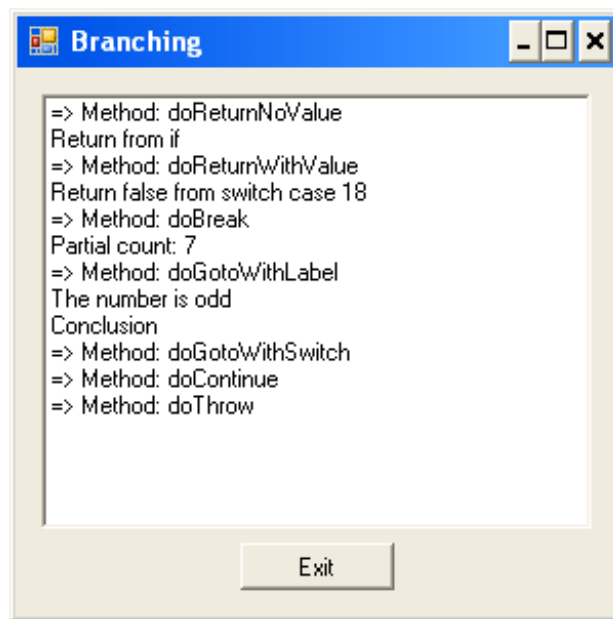
    int anyNumber = 3;

    if (anyNumber % 2 == 0)
        goto Even;
    else
        goto Odd;

    Odd:
        outputListBox.Items.Add("The number is odd");
        goto Finish;
    Even:
        outputListBox.Items.Add("The number is even");

    Finish:
        outputListBox.Items.Add("Conclusion");
}
.
.
.
```

Click  and . You should see the following output:



Let's discuss how this code works.

```
OBSERVE: Branching.cs

.
.
.
private void doGotoWithLabel()
{
    // Get method name.
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

    // A goto statement ...

    int anyNumber = 3;

    if (anyNumber % 2 == 0)
        goto Even;
    else
        goto Odd;

    Odd:
        outputListBox.Items.Add("The number is odd");
        goto Finish;
    Even:
        outputListBox.Items.Add("The number is even");

    Finish:
        outputListBox.Items.Add("Conclusion");
}
.
.
.
```

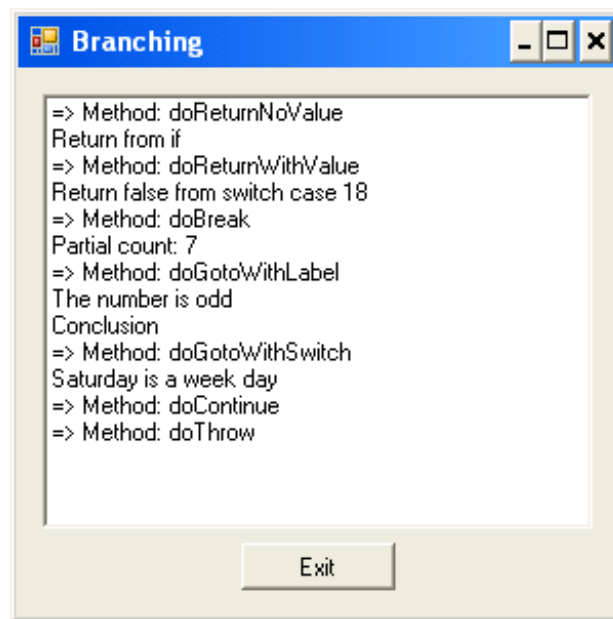
The **doGotoWithLabel** method demonstrates the use of the **goto** statement with labels: **Odd**, **Even**, and **Finish**. The **goto** statement immediately transfers control to the specified labels.

Modify **Branching.cs** as shown below to add the code to the **doGotoWithSwitch()** method. Note that only this method is shown. No other code was changed. Click the Save button  to save your changes.

Branching.cs

```
.  
.br/>.br/>private void doGotoWithSwitch()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // Same notes as in doGotoWithLabel about goto statements apply.  
  
    string dayOfWeek = "Saturday";  
    bool isWeekend = true;  
  
    switch (dayOfWeek)  
    {  
        case "Saturday":  
            isWeekend = true;  
            break;  
        case "Sunday":  
            goto case "Saturday";  
        case "Monday":  
            isWeekend = false;  
            break;  
        case "Tuesday":  
            goto case "Monday";  
        case "Wednesday":  
            goto case "Monday";  
        case "Thursday":  
            goto case "Monday";  
        case "Friday":  
            goto case "Monday";  
        default:  
            outputListBox.Items.Add(dayOfWeek + " is not a valid week day");  
            break;  
    }  
  
    outputListBox.Items.Add(dayOfWeek + (isWeekend?" is ":" is not") + "a week day");  
}  
.br/>.br/.
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.
.
.
private void doGotoWithSwitch()
{
    // Get method name.
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

    // Same notes as in doGotoWithLabel about goto statements apply.

    string dayOfWeek = "Saturday";
    bool isWeekend = false;

    switch (dayOfWeek)
    {
        case "Saturday":
            isWeekend = true;
            break;
        case "Sunday":
            goto case "Saturday";
        case "Monday":
            isWeekend = false;
            break;
        case "Tuesday":
            goto case "Monday";
        case "Wednesday":
            goto case "Monday";
        case "Thursday":
            goto case "Monday";
        case "Friday":
            goto case "Monday";
        default:
            outputListBox.Items.Add(dayOfWeek + " is not a valid week day");
            break;
    }

    outputListBox.Items.Add(dayOfWeek + (isWeekend?" is ":" is not") + "a week day");
}
.
.
.
```

The **doGotoWithSwitch** method demonstrates the use of the **goto** statement with **case** statements within a **switch** statement. The **goto** statement immediately transfers control to the specified **case** statement.

Tip

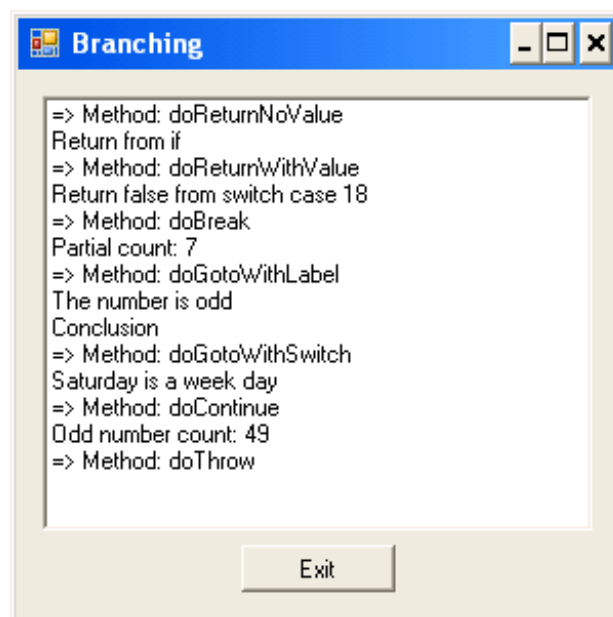
You might be curious why the parenthesis surround the ternary **? :** expression. The reason is that addition has a higher precedence than the ternary/conditional operator, so the code attempts to perform an addition of a *string* (**dayOfWeek**) and *bool* (**isWeekend**), which is invalid. Wrapping the ternary expression with parenthesis changes the precedence, allowing the ternary evaluation to take place, resulting in a *string* result.

Modify **Branching.cs** as shown below to add the code to the **doContinue()** method. Note that only this method is shown. No other code was changed. Click the Save button  to save your changes.

Branching.cs

```
.  
.br/>.br/>private void doContinue()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // The continue statement is used to skip to the next iteration of any enclosing iterating statements,  
    // such as while, do, for, or foreach. Execution is immediately transferred to the beginning of the body  
    // of that iteration statement. Good programming practices recommend using the continue sparingly to avoid  
    // obscuring the logic and flow of the source code.  
  
    int startNumber = 3;  
    int endNumber = 100;  
    int count = 0;  
  
    for (int i = startNumber; i <= endNumber; i++)  
    {  
        // Only count odd numbers  
        if (i % 2 == 0)  
            continue;  
        count++;  
    }  
  
    outputListBox.Items.Add("Odd number count: " + count);  
}  
.br/>.br/.
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.  
.   
.   
private void doContinue()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // The continue statement is used to skip to the next iteration of any enclosing iterating statements,  
    // such as while, do, for, or foreach. Execution is immediately transferred to the beginning of the body  
    // of that iteration statement. Good programming practices recommend using the continue sparingly to avoid  
    // obscuring the logic and flow of the source code.  
  
    int startNumber = 3;  
    int endNumber = 100;  
    int count = 0;  
  
    for (int i = startNumber; i <= endNumber; i++)  
    {  
        // Only count odd numbers  
        if (i % 2 == 0)  
            continue;  
        count++;  
    }  
  
    outputListBox.Items.Add("Odd number count: " + count);  
}  
.   
.   
.
```

The **doContinue** method demonstrates the use of the **continue** statement, which forces a skip to the next iteration of the enclosing iterating statement, in this case, a *for* loop, skipping the **count++** statement.

Note

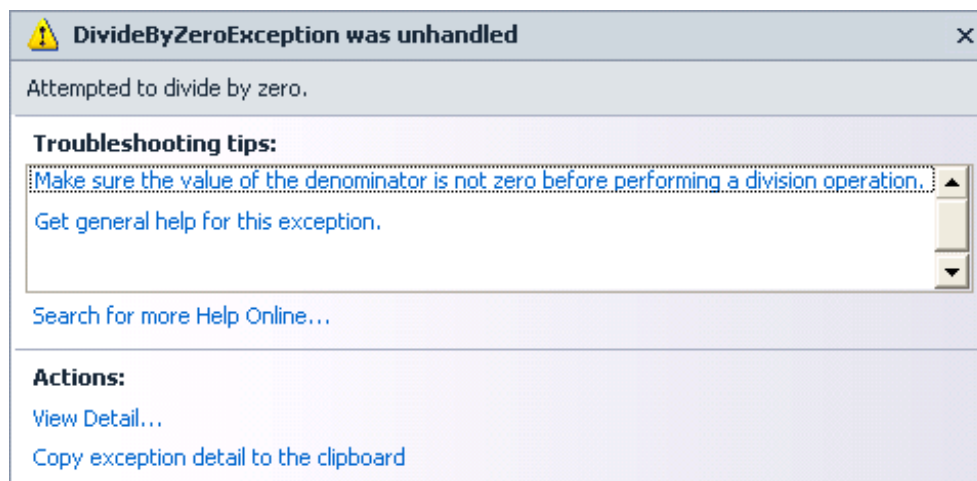
The continue statement is used to skip to the next iteration of any enclosing iterating statements, such as while, do, for, or foreach. Execution is immediately transferred to the beginning of the body of that iteration statement. Good programming practices recommend using the continue sparingly to avoid obscuring the logic and flow of the source code.

Modify **Branching.cs** as shown below to add the code to the **doThrow()** method. Note that only this method is shown. No other code was changed. Click the Save button  to save your changes.

Branching.cs

```
.  
.   
.   
private void doThrow()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // The throw statement is used to signal that an anomalous situation, also known as  
    an exception,  
    // has occurred. Exceptions are handled using try..catch blocks. If an exception oc  
    curs in a program,  
    // the system looks for a try..catch block that will handle the exception. These ex  
    ceptions are  
    // raised by C# even if you do not throw an exception. Throwing an exception is usu  
    ally done  
    // by the user with custom exceptions to use the try..catch block to handle unusua  
    l circumstances  
    // that might occur. This code demonstrates how to throw a standard exception, divi  
    de  
    // by zero, but even if you had not thrown the exception, it would have been raised  
    when  
    // the actual divide by zero was attempted. Without a try..catch block, the system  
    will  
    // stop on the exception.  
  
    int numerator = 10;  
    int denominator = 0;  
    int result = 0;  
  
    if (denominator == 0)  
        throw new DivideByZeroException();  
    else  
        result = numerator / denominator;  
}  
.   
.   
.
```

Click  and . You will see the following exception. Click the  to stop the program so we can correct the error after we discuss how the code works.



OBSERVE: Branching.cs

```
.  
.   
.   
private void doThrow()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // The throw statement...  
  
    int numerator = 10;  
    int denominator = 0;  
    int result = 0;  
  
    if (denominator == 0)  
        throw new DivideByZeroException();  
    else  
        result = numerator / denominator;  
}  
.   
.   
.
```

The **doThrow** method demonstrates how to throw an exception. Exceptions are situations that are unexpected, but would cause a serious problem if the exception is not correctly dealt with. Examples of exceptions include a database connection becomes invalid, access to the network is disabled, or an attempt to divide by 0, as in our example. Unhandled exceptions will cause the program to either become unstable, unusable, or to stop functioning. In **doThrow**, we detect if **denominator** is 0, and if so, we **throw** a **DivideByZeroException**, creating an instance of that exception during the **throw**. This situation is a bit contrived, as we have already detected that the **denominator** is 0, so we have other possible ways of dealing with it, such as telling the user that the **denominator** must not be 0.

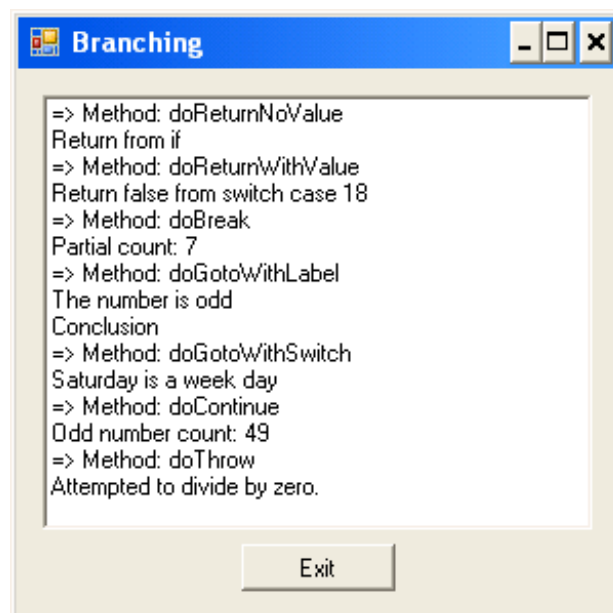
Since this code as written will raise an exception and freeze the code, let's add a *try..catch* block to allow the program to continue to function.

Modify **Branching.cs** as shown below to add the code to the **doContinue()** method to handle the **DivideByZeroException**. Note that only this method is shown. No other code was changed. Click the Save button  to save your changes.

Branching.cs

```
.  
.br/>.br/>private void doThrow()  
{  
    // Get method name.  
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);  
  
    // The throw statement...  
  
    int numerator = 10;  
    int denominator = 0;  
    int result = 0;  
  
    try  
    {  
        if (denominator == 0)  
            throw new DivideByZeroException();  
        else  
            result = numerator / denominator;  
    }  
    catch (Exception ex)  
    {  
        outputListBox.Items.Add(ex.Message);  
    }  
}
```

Click  and . You should see the following output:



Let's discuss how this code works.

OBSERVE: Branching.cs

```
.
.
.
private void doThrow()
{
    // Get method name.
    outputListBox.Items.Add("=> Method: " + System.Reflection.MethodInfo.GetCurrentMethod().Name);

    // The throw statement...

    int numerator = 10;
    int denominator = 0;
    int result = 0;

    try
    {
        if (denominator == 0)
            throw new DivideByZeroException();
        else
            result = numerator / denominator;
    }
    catch (Exception ex)
    {
        outputListBox.Items.Add("You cannot divide by 0");
    }
}
.
.
.
```

Now, when you run the program, the **doThrow** method will still **throw** the **DivideByZeroException** exception, but now the **try..catch** block will "catch" the error. We'll cover **try..catch** blocks and exceptions more later.

Note

As mentioned above, we actually do not need to throw the exception, as the attempt to divide by zero would have raised the same exception. Later, we'll introduce custom exceptions, so knowing how to throw an exception now will prepare you for that lesson.

Conditional Branching

Let's take a look at the C# list of **Conditional Branching** techniques:

- The *if*, *else*, and *else if* keywords
- The *switch* keyword

Sprinkled throughout our **unconditional branching**, you can find examples of all of these **conditional branching** techniques, so we do not need to provide more sample code for you to enter. We should reiterate a couple of points:

- *if*, *else*, *else if* statements require Boolean (*bool*) expressions
- *switch* statements only branch on *bool*, *char*, *string*, integral data types (*sbyte*, *short*, *int*, *long*, *byte*, *ushort*, *uint*, *ulong*), or enumerations (covered later, but essentially a named constant that should only be of type *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, or *ulong*).

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Iteration (Looping)

In this lesson, we will cover iterations, or looping—another topic we've sprinkled into our lessons so far. The iteration statements in C# are *for*, *while*, and *do..while*. There is also a *foreach* statement using the *in* keyword that is used to iterate over elements within an array or collection object. We'll discuss this iteration technique later, when we discuss arrays.

The For Loop

The *for* loop is used to repeat one or more statements until a specific expression evaluates to false. The statements within a *for* loop may not execute if the expression evaluates to false initially. The format and components (note that all of the components are optional) are:

For Loop Syntax:

```
for (declaration and initialization ; condition ; iteration )
```

The **declaration and initialization** component of the *for* loop may be used to declare one or more variables. Good programming practices would recommend using the **declaration and initialization** component to declare and initialize a variable that would be used to establish the iteration, as we'll see shortly. However, you can declare variables before starting a *for* loop if you want to access the value of such variables after the loop concludes. Variables declared within the **declaration and initialization** section are not available after the loop concludes. Note that we used the plural—variables. You may declare and initialize multiple variables, separated by commas. Here are a few examples of **declarations and initializations**:

- `for ( ; ; )`
- `int i;
for (i = 0 ; ;)`
- `for (int x = 0 ; ; )`
- `for (int x = 0, y = 0 ; ; )`

The **condition** component of the *for* loop may be used to evaluate an expression to determine the number of iterations of the loop. The **condition** component may use any combination of boolean, relational, and logical operators. Here are a few examples of **conditions**:

- `for (int i = 0 ; i < 10 ; )`
- `for (int x = 0, y = 0 ; x >= 5 && y < 15 ; )`

If you omit the **condition** component, you must provide an alternative method to exit the loop, or you will be create an *infinite loop*. Even with the **condition**, you must provide a way to exit the loop, typically by adjusting the variables of the **condition** component in the **iteration** component.

The **iteration** component of the *for* loop may be used to adjust one or more variables usually used to control the number of times loop is repeated. If you do not adjust the variables within the **iteration** section, you must provide a way to adjust those variables to prevent creating an infinite loop. Here are a few examples of **iterations**.

- `for (int i = 0 ; i < 10 ; i++)`
- `for (int x = 0, y = 0 ; x >= 5 && y < 15 ; x++, y++)`
- `for (int i = 0 ; i < 10 ; i += 5)`

The While Loop

The *while* loop is used to repeat one or more statements repeatedly until a specific expression evaluates to false. The statements within a *while* loop may never execute if the expression evaluates to false initially. The format of the *while* loop is:

While Loop Syntax

```
while (condition)
```

The **condition** expression of the while loop consists of an expression normally used to determine how many times to iterate through the loop. Within the loop block, variables contained within the **condition** expression are modified to ensure that the loop eventually concludes to prevent an *infinite loop* condition. Just as with the for loop, you may use any combination of boolean, relational, and logical operators. Here are a few examples:

- **while (notFound == true)**
- **while (!recordFound && !endOfRows)**

The Do..While Loop

The *do..while* loop (often referred to as just a *do* loop) repeats one or more statements until a specific **condition** evaluates to false. The statements in a *do..while* loop must execute at least once, because the evaluation expression follows the statements. Below is the format of the *do..while* loop:

Do..While Loop Syntax
do while (condition)

The structure of the *do..while* indicates that all statements following the **do** component will execute *before* the **condition** is evaluated. Whenever you need a loop that must execute at least once, use the *do..while* mechanism. For examples, see the while loop above.

For Loop Versus While Loop

So, when do you use a for loop, and when do you use a while or *do..while* loop? Generally, when you know the number of iterations, you use a for loop. For indefinite looping, or looping until a condition changes, use a while or *do..while* loop. So, with the for loop, you typically will see loops that involve counting, whereas with while or *do..while* loops, often the evaluation uses boolean expressions. Regardless, you can use any looping structure to accomplish the desired task.

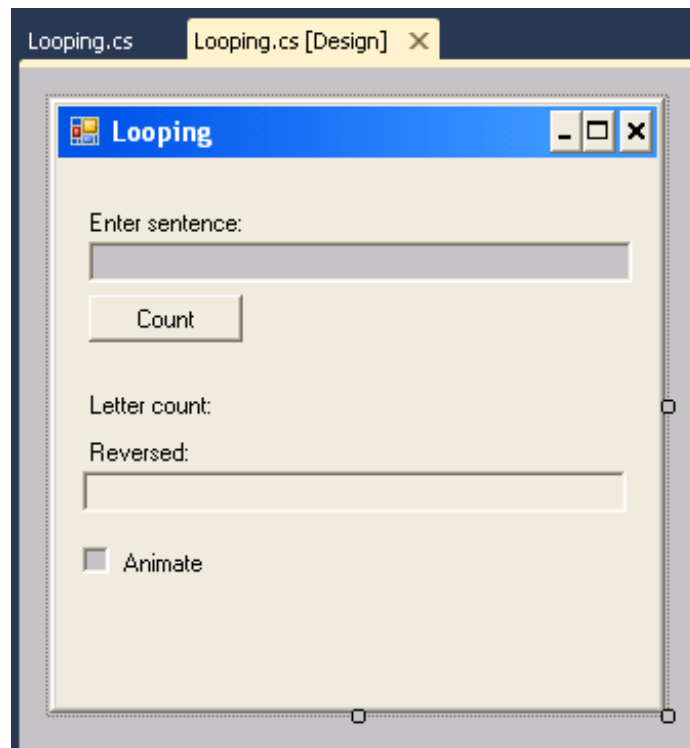
Putting Looping To Work

Let's create a project to experiment with these different kinds of loops.

Selecting **File | New | Project**. Change the project name to **Looping** and click **OK**.

Click the entry for **Form1.cs** and change it to **Looping.cs**. Change the form title bar's Text property in the Properties Window to **Looping**. Click  to save your changes.

Modify the form to look like this:



Change each control's Name property to match the Name Property column below, and Text property to match the Text Property column. Arrange the controls similar to the image. Also, set the ReadOnly property of the reversedTextBox control to **True**. When you finish, click  to save your changes.

Object	Name Property	Text Property
Label	sentenceLabel	Enter sentence:
TextBox	sentenceTextBox	
Button	countButton	Count
Label	letterCountLabel	Letter count:
Label	reversedLabel	Reversed:
TextBox	reversedTextBox	
CheckBox	animateCheckBox	Animate

Next, add event handlers for the **countButton** and **animateCheckBox** controls. You can either double-click on each control to automatically generate the default event handler code or, for each control, use the Properties Window,

click the control, click the **Events** icon , and locate the event we need to handle for the control. For the **countButton** control, we need to handle the **Click** event, and for the **animateCheckBox** control, we need to handle the **CheckedChanged** event. Once you've located the correct event, you could type in the name of the handler, but in this case, it's much easier to just double-click each event. Studio generates an appropriate name for the event handler that consists of the name of the control, an underscore separator, and the event name.

Note

In general, double-clicking a control will generate the typical event for this control, but you should still know how to find the other events for the control, and generate appropriate handlers for those events, as we'll have other non-default events we'll want to handle.

Modify the **Looping.cs** code as shown below. The event handler code was added previously.

Looping.cs

```
using ...

namespace Looping
{
    public partial class Looping : Form
    {
        public Looping()
        {
            InitializeComponent();
        }

        private void countButton_Click(object sender, EventArgs e)
        {
            int letterCount = 0;

            // Use for loop to find and count letters only.
            for (int i = 0; i < sentenceTextBox.Text.Length; i++)
            {
                // Determine if we have a character.
                if (char.IsLetter(sentenceTextBox.Text, i))
                    letterCount++;
            }

            letterCountLabel.Text = "Letter count: " + letterCount;

            // Use while loop to reverse.
            StringBuilder reverseSB = new StringBuilder(sentenceTextBox.Text.Length);

            int letterPosition = sentenceTextBox.Text.Length;
            while (letterPosition > 0)
            {
                // Add character, working from end of string backwards.
                reverseSB.Append(sentenceTextBox.Text.Substring(--letterPosition, 1));
            }

            reversedTextBox.Text = reverseSB.ToString();
        }

        private void animateCheckBox_CheckedChanged(object sender, EventArgs e)
        {
        }
    }
}
```



and to run the program. Enter a short sentence in the text box that includes some numbers and a known number of letters, and click **Count**. The program should display the letter count and your sentence, backwards.

Let's discuss how this code works (we'll show it in two parts).

OBSERVE: Looping.cs, Part 1

```
.
.
.
private void countButton_Click(object sender, EventArgs e)
{
    int letterCount = 0;

    // Use for loop to find and count letters only.
    for (int i = 0; i < sentenceTextBox.Text.Length; i++)
    {
        // Determine if we have a character.
        if (char.IsLetter(sentenceTextBox.Text, i))
            letterCount++;
    }

    letterCountLabel.Text = "Letter count: " + letterCount;
.
.
.
```

Our for loop declares and initializes a loop variable **i** (**int i = 0**), looping while the loop variable is less than the length of the string **i < sentenceTextBox.Text.Length**, and incrementing the loop variable for each iteration (**i++**). Notice that we use less than (**<**) rather than less than or equal to (**<=**). Why? Remember that for most objects, C# uses 0-based indexing, which means that the first element is at index, or position, 0. That also means that the final element in a 0-based object is going to be the length of the object minus 1. So, we loop while the loop counter is less than the length. So, this for loop counts the number of letters by incrementing **letterCount**, which we accomplish by calling the very suitable **IsLetter** method of **char**. This method has two parameters: the complete string to look at, and the position within that string to check.

What exactly is the flow of a for loop? It processes in the following order:

0. **declaration and initialization** 1. **condition** 2. code within the for loop block 3. **iteration** 4. goto Step 1

So, Step 0 only occurs once, and Step 2 is only reached if Step 1 is true. Of course, you can always exit the for loop using a break statement.

OBSERVE: Looping.cs, Part 2

```
.
.
.
    // Use while loop to reverse.
    StringBuilder reverseSB = new StringBuilder(sentenceTextBox.Text.Length);

    int letterPosition = sentenceTextBox.Text.Length;
    while (letterPosition > 0)
    {
        // Add character, working from end of string backwards.
        reverseSB.Append(sentenceTextBox.Text.Substring(--letterPosition, 1));
    }

    reversedTextBox.Text = reverseSB.ToString();
}
.
.
.
```

We next added a while loop to reverse the user-entered text. How this code works might not be obvious, but you might suspect something is strange when you see the **--letterPosition** code, which uses the pre-decrement operator. This while loop iterates as long as **letterPosition > 0**. We are effectively iterating backward through each element of the **sentenceTextBox.Text**, using the **Substring** method to extract that element. You should note that **letterPosition** is set to the length of **sentenceTextBox.Text**, and since this is an invalid position within the string, we use the pre-decrement operator to decrease the **letterPosition** variable by one before we attempt to extract the substring. We also are able to use **letterPosition > 0** because of the pre-decrement. Work through this loop with a short sentence to make sure you understand how it works. How? When you run the program, you can set a break point and evaluate

the value of **letterPosition** through each iteration, and see what first decrementing the position does for extracting the correct substring.

We also introduced an important class: **StringBuilder**. Whenever you're going to be creating a string result through concatenation (adding pieces of text to each other), consider using the **StringBuilder** class rather than a string data type. When you create a string, memory is allocated for the value of the string, and that memory is fixed, or what is often referred to as *immutable*. A simple string concatenation actually requires creating another temporary string in memory to hold the concatenated string result, reassigning the initial string to reference the new memory, and then deallocating the original memory. That's a lot of work, especially in a loop! Rather than concatenate with a string, you should use the **StringBuilder** class, which is optimized for such concatenation operations.

We've covered for and while loops; now let's add a do..while loop. For this loop, though, we'll have a bit more fun!

Add event handlers for the Looping Form **Paint** and **FormClosing** events; click on the title bar of the Looping Form in

the Visual Designer, click the  icon in the Properties Window, find the **Paint** event and double-click it. The Code Editor appears. Switch back to the Visual Designer, find the **FormClosing** event, and double-click it. Then, modify the **Looping.cs** code as shown below (again, our code is getting lengthy, so we'll just show the parts that we want to change).

Tip

Simply adding the event handler code will NOT connect the event handler method to the control: you must either double-click the control to generate a handler for the default event for the control, or use the Events icon in the Properties Window.

Looping.cs

```
using ...

namespace Looping
{
    public partial class Looping : Form
    {
        bool doAnimation = false;
        .
        .
        .

        private void Looping_Paint(object sender, PaintEventArgs e)
        {
            string drawText = sentenceTextBox.Text.Length > 0 ? sentenceTextBox.Text :
"Some text";
            // Determine our approximate stopping point.
            float maxWidth = e.Graphics.VisibleClipBounds.Width;
            // Set up starting x and y locations, noting that only the x is going to ch
ange.
            float x = animateCheckBox.Left;
            float y = animateCheckBox.Top + (animateCheckBox.Height * 2);
            // We'll need a previous x location so we can 'erase' the previous drawing.
            float previousX = x;

            // Loop until either we've reached our stopping point, or doAnimation is fa
lse.
            do
            {
                // Draw text at old location.
                e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(th
is.BackColor), previousX, y);
                // Make the system sleep for a short time so we can see the animation.
                System.Threading.Thread.Sleep(20);
                // Allow the program to process events so we can allow the user interfa
ce to be responsive after sleeping.
                Application.DoEvents();
                // Draw text at new location.
                e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(Co
lor.Red), x, y);
                // Save the current location.
                previousX = x;
                // Advance to next location.
                x++;
            } while (x < maxWidth && doAnimation);

            // Do some cleanup.
            e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(this.B
ackColor), previousX, y);
            animateCheckBox.Checked = false;
            doAnimation = false;
        }

        private void animateCheckBox_CheckedChanged(object sender, EventArgs e)
        {
            // Determine if we are to animate, then force paint event to occur using Re
fresh.
            doAnimation = animateCheckBox.Checked;
            this.Refresh();
        }

        private void Looping_FormClosing(object sender, FormClosingEventArgs e)
        {
            // We need to prevent, or cancel, closing, as code may still be in animatio
n do loop.
            if (doAnimation)
            {
```

```

        MessageBox.Show("You must stop any animation before trying to close the
program", this.Text, MessageBoxButtons.OK);
        e.Cancel = true;
    }
}
}
}

```



and to run the program. Enter a sentence and reverse it, and then check the Animate box and see what happens.

Let's discuss how this code works. First the Looping_Paint method:

OBSERVE: Looping.cs, part 1

```

private void Looping_Paint(object sender, PaintEventArgs e)
{
    string drawText = sentenceTextBox.Text.Length > 0 ? sentenceTextBox.Text :
"Some text";
    // Determine our approximate stopping point.
    float maxWidth = e.Graphics.VisibleClipBounds.Width;
    // Set up starting x and y locations, noting that only the x is going to ch
ange.
    float x = animateCheckBox.Left;
    float y = animateCheckBox.Top + (animateCheckBox.Height * 2);
    // We'll need a previous x location so we can 'erase' the previous drawing.
    float previousX = x;

    // Loop until either we've reached our stopping point, or doAnimation is fa
lse.
    do
    {
        // Draw text at old location.
        e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(th
is.BackColor), previousX, y);
        // Make the system sleep for a short time so we can see the animation.
        System.Threading.Thread.Sleep(20);
        // Allow the program to process events so we can allow the user interfa
ce to be responsive after sleeping.
        Application.DoEvents();
        // Draw text at new location.
        e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(Co
lor.Red), x, y);
        // Save the current location.
        previousX = x;
        // Advance to next location.
        x++;
    } while (x < maxWidth && doAnimation);

    // Do some cleanup.
    e.Graphics.DrawString(drawText, new Font("Arial", 8), new SolidBrush(this.B
ackColor), previousX, y);
    animateCheckBox.Checked = false;
    doAnimation = false;
}

private void animateCheckBox_CheckedChanged(object sender, EventArgs e)
{
    // Determine if we are to animate, then force paint event to occur using Re
fresh.
    doAnimation = animateCheckBox.Checked;
    this.Refresh();
}

```

As you can see, we've added code that uses the **DrawString** method to effectively make whatever text the user types in to appear to move, or animate, across the screen, by selecting the **animateCheckBox** control. If no text is entered, we use the conditional/ternary operator we've learned (**? :**) to use some default text. We use the **do..while** loop to iterate through all possible **x** positions across the screen, after setting a starting point below the **animateCheckBox**. We use a compound conditional statement, **x < maxWidth && doAnimation**, to make sure we stop trying to animate once we've reached the maximum **x** position, or if **doAnimation** becomes false.

What is the purpose of the **doAnimation** class variable? The method we've used to animate the text is not the best solution for this type of action, but we wanted to illustrate the **do..while** loop. A better solution would be a timer event, but that wouldn't require a loop. We had to use the **Sleep** method to slow down the drawing of our text; otherwise, the text would draw so quickly you wouldn't see the animation. Also, we had to use a **DoEvents** method to allow our code to process any events that might have been queued while the **Sleep** method was called. Within our **do..while** loop, we want to be able to uncheck the **animateCheckBox** to effectively stop the animation, which is accomplished by changing the **doAnimate** variable to **false**. We also added a call to the Looping Form **Refresh** method, which triggers the Paint event, thus calling the **Looping_Paint** method.

Now let's look at the **Looping_FormClosing** method:

OBSERVE: Looping.cs, part 3

```
private void Looping_FormClosing(object sender, FormClosingEventArgs e)
{
    // We need to prevent, or cancel, closing, as code may still be in animation do loop.
    if (doAnimation)
    {
        MessageBox.Show("You must stop any animation before trying to close the program", this.Text, MessageBoxButtons.OK);
        e.Cancel = true;
    }
}
```

The **Looping_FormClosing** method handles any attempt to close the Looping form. We didn't add a button to allow the

user to close the form, so if they want to close it, they must click the , which triggers the **FormClosing** event, thus calling this method, which is linked to handle this event. If they try to close the form while the text is animating, when the **do..while** code executes the **DoEvents** line, our code will suspend while the method handles the **FormClosing** event. Trying to exit while the **do..while** code is exiting could lead to the Looping form appearing to close, but actually just disappearing—and then the program returning to finish the **do..while** loop, then finally closing. For our code, this scenario isn't a big deal, but we'd rather introduce this concept of *event-driven* coding now, as we'll see more such scenarios in the future, and will need to learn how to properly handle *asynchronous* events (events received in an unspecified order). For now, we display a **MessageBox**, and cancel the closing of the form by setting the **Cancel** property to **true**.

So how exactly are we making the text appear to animate? Each iteration through the **do..while** loop, we first "erase" the existing text by drawing it in the same color as the background at its previous position (**previousX**), then we increment the **x** value and draw the text in red at the new position. When the loop is concluded, we again draw the text at the **previousX** position to erase it one last time. That's it!

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Input and Output Statements

Programming and logic are wonderful things, but they don't exist in a vacuum. They need to get information from, and return it to, their creators. In this lesson, we'll learn about how to perform input and output. We've already seen output by writing to the Console, but now we'll expand our arsenal to include writing output to the screen, and to a file, as well as reading input from the screen and a file. We'll also learn the difference between a Windows application and a console application, and how they differ when it comes to input and output. Let's get started!

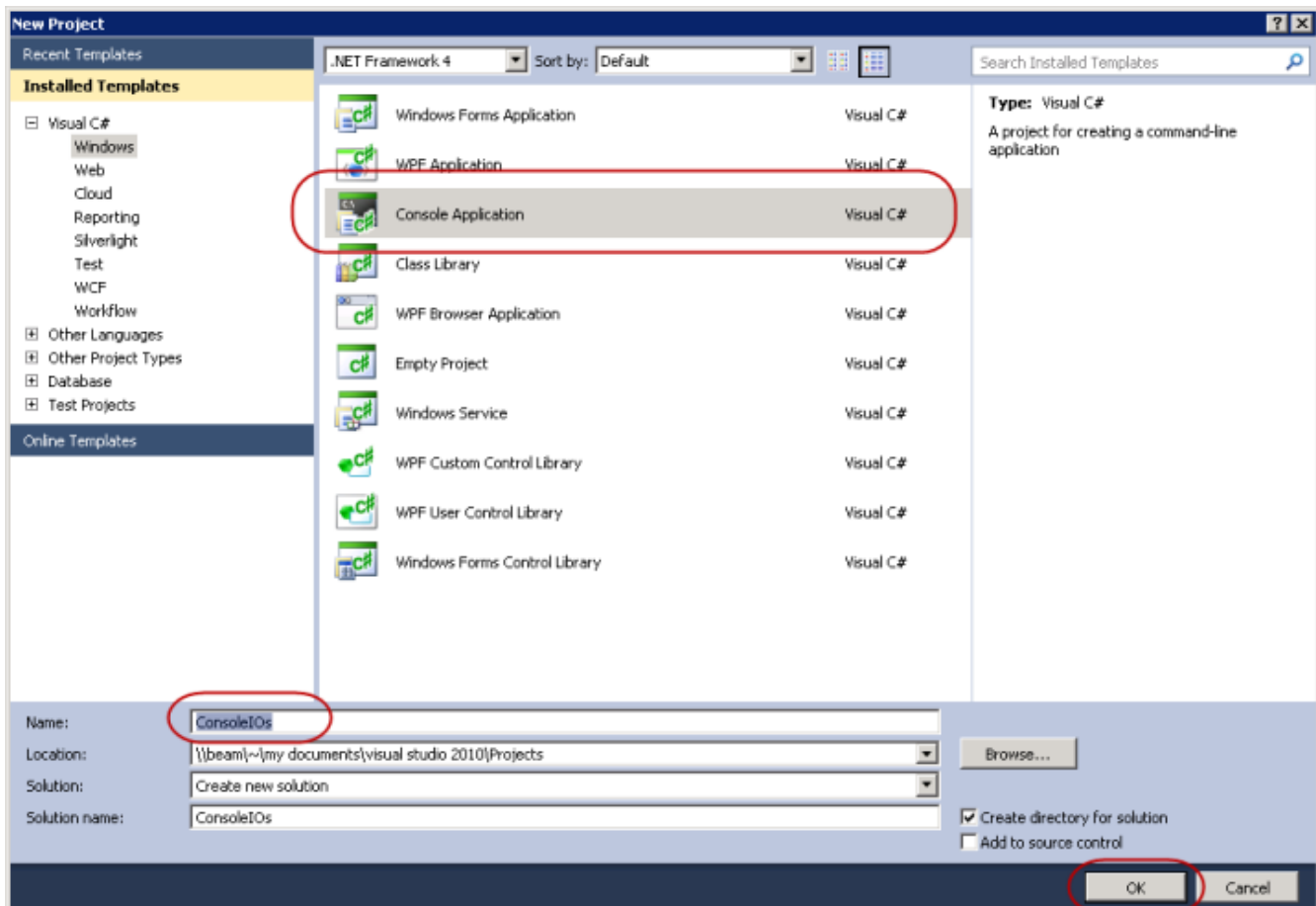
Windows GUI Application Versus Console Application

So far, for every application we've created, we've selected the default *Windows Forms Application*. You probably noticed that there are quite a few other types of projects we could create, such as a *Console Application*. A Console Application runs as a *command-line* application, without the graphical windows we've become so used to. A Console Application can have a GUI (graphical user interface), but it isn't constructed using the Microsoft Windows GUI components. Many older applications are console GUI applications. Also, numerous command-line/console applications exist on whatever operating system you're using, adding functionality that does not require a GUI for user interaction. Depending on your operating system, you may have heard of a command prompt, DOS prompt, MS-DOS prompt, or command-line window. Fortunately for us, when you create a Console Application and execute it, Studio opens a command-line window, or console window, and executes the application. Because a command-line application executes then ends without a GUI, we will add functionality to the end of our Console Application to allow it to pause before the console window closes.

Let's create our first Console Application before we move on to input and output statements.

Simple Console Input and Output

Select **File | New | Project**. In the center section of the New Project dialog box, select **Console Application**. Change the Name of the project to **ConsoleIOs**. All other dropdowns and check boxes take their default settings:



Click **OK**. After you've created the new Console Application project, you'll notice a few differences: the code in the Visual Editor is different, including the using statements, there is no Form class (Form1.cs), and the default class for the C# source code is **Program.cs**. Let's look at the default code.

OBSERVE: ConsoleIO.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleIO
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

It turns out that our Windows Forms Application also has a Program.cs file, but we've never discussed this file yet. Both a Console Application and a Windows Forms Application must have what is known as an *entry point*: the code that is first called when the application starts. For both types of applications, the code in a **Main** method is first called. In a Windows Forms Application, we've always skipped the **Main** code and gone right to the Form, as that is where we needed to add our controls and code to get the application going. With a Console Application, we will work directly within **Main**.

A Console Application can have optional command-line parameters passed to it. These command-line parameters would be available to our code in the **args** parameter of **Main**. Also, you'll notice a new keyword: **static**. Using the **static** keyword means that there is only one, and in our case, that makes sense, as we only want one entry point into our program. We'll discuss **static** more in later lessons, as for now we're mainly interested in input and output. Let's add some output to our program.

Modify **Console.cs** as shown below.

ConsoleIO.cs

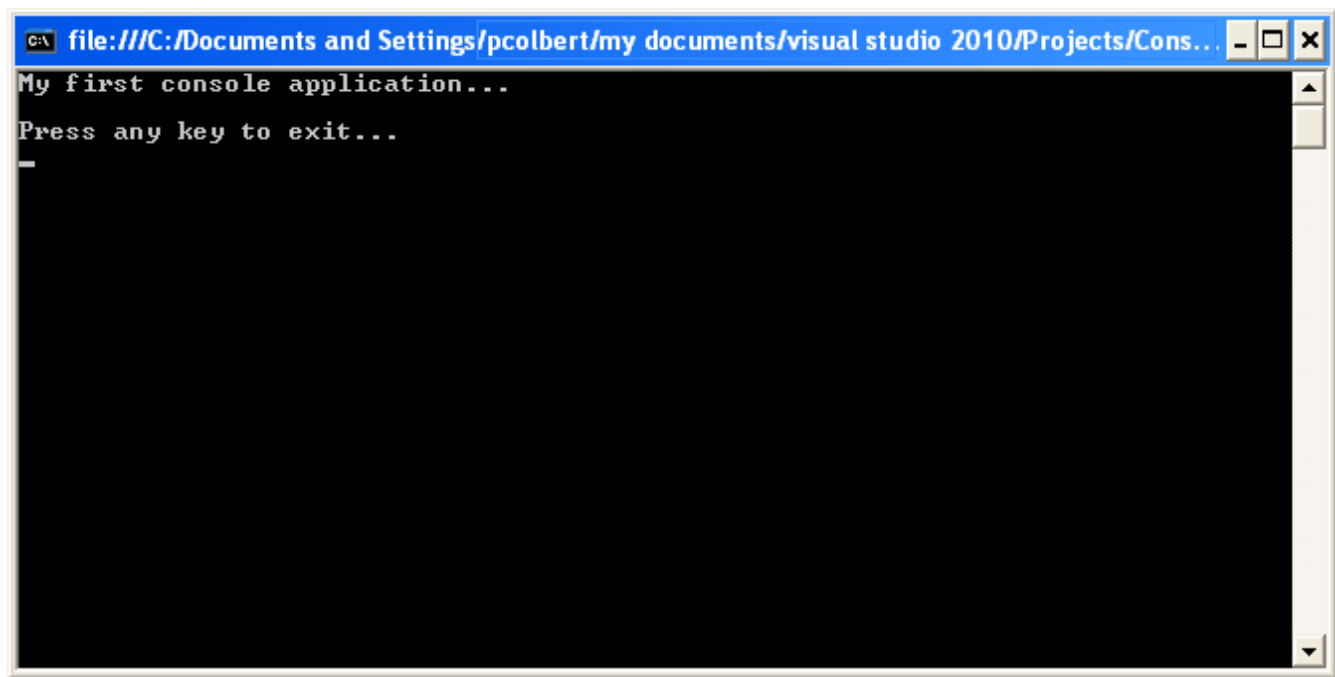
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleIO
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("My first console application!");

            // Keep the console open in debug mode.
            Console.WriteLine("\nPress any key to exit.");
            Console.ReadKey();
        }
    }
}
```



and  to run the program



You've seen the `Console.WriteLine()` method many times before, but we did add a line of code that is actually our first input statement: `Console.ReadKey()`. This method waits until any key is pressed. Once a key is pressed, the **`ReadKey()`** method allows the program to continue, effectively creating the pause in the program we mentioned. You can try to click on the console window to see if it will let the program close. It won't.

Let's introduce some other ways we can get keyboard input.

For console input, there are three basic routines you can use to read data directly from the keyboard:

- `Console.ReadKey`: Immediately returns when any key is pressed.
- `Console.Read`: Only returns after the Enter key is pressed, only returning the last key pressed before Enter key.
- `Console.ReadLine`: Only returns after the Enter key is pressed, but will return all keys pressed before Enter key.

Modify **`Console.cs`** as shown below.

ConsoleIO.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleIO
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("My first console application!");

            bool quitLoop = false;
            bool interceptKey = true;

            // Use ReadKey.
            Console.WriteLine("\nPress any key; then, press Enter to exit.\n");
            do
            {
                Console.Write("> ");
                ConsoleKeyInfo keyInfo = Console.ReadKey(interceptKey);
                if (keyInfo.Key == ConsoleKey.Enter)
                {
                    quitLoop = true;
                    Console.WriteLine("You pressed: " + keyInfo.Key);
                } while (!quitLoop);

                // Use ReadLine.
                Console.WriteLine("\nType in any characters; press Enter to exit.\n");
                do
                {
                    Console.Write("> ");
                    string enteredLine = Console.ReadLine();
                    if (enteredLine.Length == 0)
                        break;
                    Console.WriteLine("You typed {0} character" + (enteredLine.Length > 1 ?
"s." : "."), enteredLine.Length);
                } while (true);

                // Keep the console open in debug mode.
                Console.WriteLine("\nPress any key to exit.");
                Console.ReadKey();
            }
        }
    }
}
```



and  to run the program and follow the onscreen instructions.

Let's discuss how this code works.

OBSERVE: ConsoleIO.cs

```
.  
.   
.   
  
    // Use ReadKey.  
    Console.WriteLine("\nPress any key; then, press Enter to exit.\n");  
    do  
    {  
        Console.Write("> ");  
        ConsoleKeyInfo keyInfo = Console.ReadKey(interceptKey);  
        if (keyInfo.Key == ConsoleKey.Enter)  
            quitLoop = true;  
        Console.WriteLine("You pressed: " + keyInfo.Key);  
    } while (!quitLoop);  
  
.   
.   
. 
```

We used the **Console.ReadKey()** and **ReadLine()** methods to process user input. The **ReadKey** uses a do..while loop to continue to take user input from the keyboard until the user presses **Enter** on a blank line (which will ultimately set **quitLoop** to true, providing a means to exit the loop). We store information about the key pressed in **keyInfo**. You'll notice that we pass **interceptKey** (with a value of true) to **Console.ReadKey** to suppress the display of the key pressed by the user. We then extract the key pressed and output it using **keyInfo.Key**, comparing the key pressed to **ConsoleKey.Enter** to see if we need to exit.

OBSERVE: ConsoleIO.cs

```
.  
.   
.   
  
    // Use ReadLine.  
    Console.WriteLine("\nType in any characters; press Enter to exit.\n");  
    do  
    {  
        Console.Write("> ");  
        string enteredLine = Console.ReadLine();  
        if (enteredLine.Length == 0)  
            break;  
        Console.WriteLine("You typed {0} character" + (enteredLine.Length > 1 ?  
"s" : ""), enteredLine.Length);  
    } while (true);  
  
.   
.   
. 
```

For **ReadLine()**, we again use a do..while loop, but this time we use a conditional statement of **true**, which means that the do..while loop will loop forever unless we use another mechanism to exit. In this case, we use a **break** statement when the user presses **Enter** on a blank line. The **Console.ReadLine()** method accepts multiple characters until **Enter** is pressed. We store the entered characters in **enteredLine**. We check the length of **enteredLine** and if no characters were entered, we use **break** to exit the loop. We output the number of typed characters using formatted output (**{0}**), and the conditional/ternary operator (**? :**) to determine if we need to add an "s" (if more than one character was entered).

You may notice that we used both output versions of **Console: Write()** and **WriteLine()**. **WriteLine** automatically includes a carriage return when it writes, whereas a **Write** does not, allowing the prompt for the **Console.ReadLine()** (and **Console.ReadKey()**) to follow after the prompt (**>**).

So, that's simple input and output from a console application. What about from a GUI? Let's make a new project to explore input and out from a Windows Forms Application.

GUI Input and Output

We've already seen a number of ways to output information to a user using a variety of controls: Labels, TextBoxes, ListBoxes, drawing directly to a form, etc. We've also used quite a few controls to get input from a user. What we haven't done yet is to create a new Form with our Windows Forms Application that might serve as a *dialog box*. What is

a dialog box? A dialog box is a window that appears in front of all other windows. As discussed earlier, they can be of two types: *modal* dialog boxes must be dismissed before control can be passed back to the calling program; *modeless* dialog boxes do not require being dismissed, allowing the user to interact with the calling program while the dialog box remains visible. For the project for this lesson, we want a *modal dialog box* to force the user to either enter the desired information, or to dismiss/cancel the dialog box, before they continue working.

Dialog boxes also have a few unique GUI properties: they typically cannot be resized (exceptions are allowed), they may not have the usual Windows controls (minimize, maximize, close, or the system icon) at their top left and top right, and do not appear up in the Windows Taskbar. Basically, the intent of a dialog box is to alert the user of an immediate need for input, or to convey some type of information, before the program can continue. As the developer, you do have the freedom to create dialog boxes as you desire depending on the user interaction you want or need to make your program work. Microsoft does have a number of articles that constitute a *style guide* or *user experience* guidelines for dialog boxes that you may want to research, but in general you can observe the behavior of existing Windows software for examples of what you can do.

Let's create a **Dialog Box Form** to request user input.

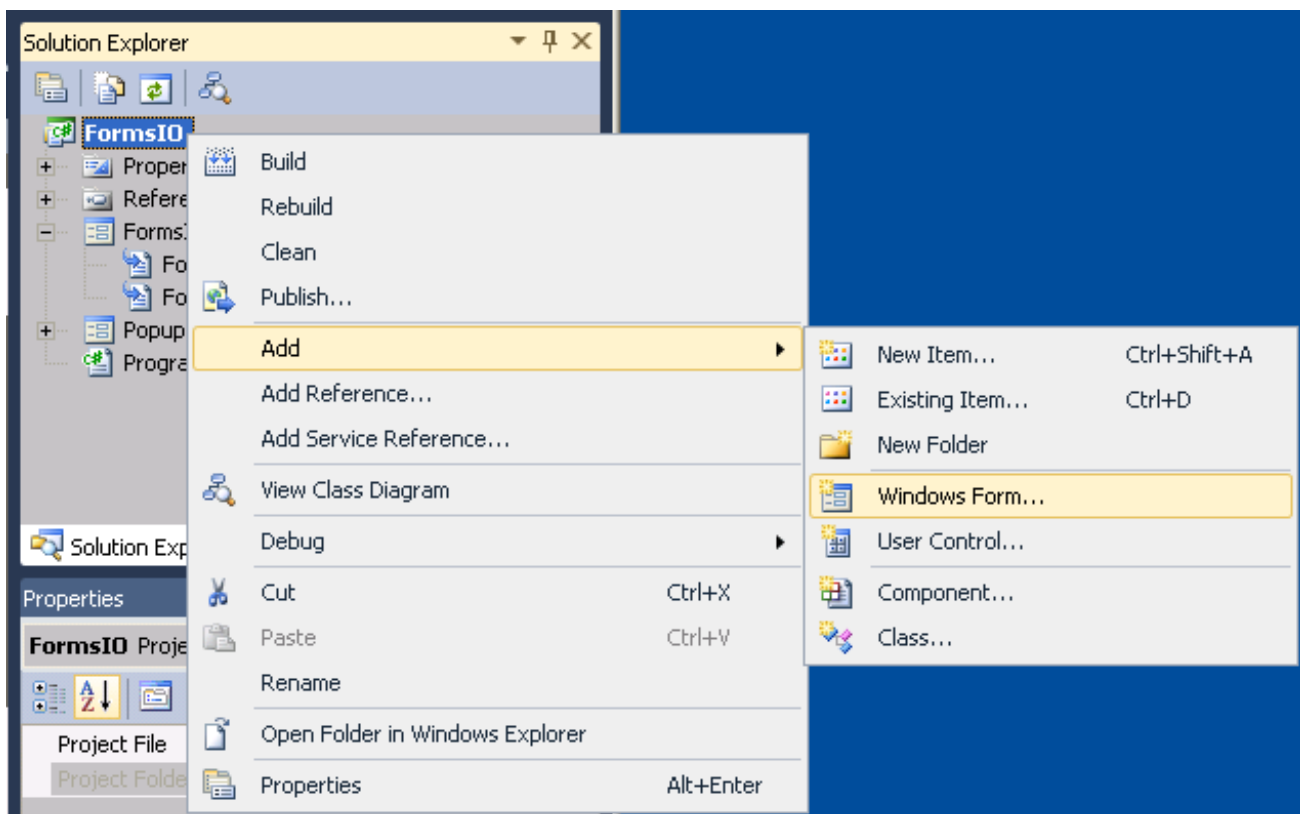
Select **File | New | Project**. Change the name to **FormsIO**, and make sure **Windows** is selected under the C# tree item, and **Windows Forms Application** is selected in the center section (this will have changed after our last project). All other dropdowns and check boxes take their default settings. Click **OK**.

Rename the file Form1.cs to **FormsIO.cs**. Change the form's title bar Text property in the Properties Window from Form1 to **Forms IO**. Click  to save your changes.

Adding a Label control and a Button control. Change the Label control Name property to **hiLabel** and Text property to **Hi**. Change the Button control Name property to **dialogButton** and Text property to **Dialog**. Double-click the **dialogButton** control to add an event handler for the button. Click  to save your changes.

Next, we'll add a new Form to our project that will become our popup dialog box.

In the Solution Explorer, right-click the FormsIO project name, and select **Add | Windows Form...**:

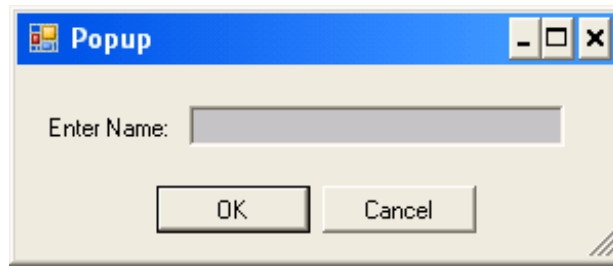


In the Add New Item - FormsIO dialog box, make sure that **Windows Form** is selected. Change the Name field to **Popup.cs**, and then click **Add**.

Now that we've added the new Popup Form, let's add a few controls, and change the size of the form.

Modify the Popup Form by adding the controls listed below. Change each control's Name property to match the Name Property column, and each Text property to match the Text Property column. Arrange the controls and resize the form

as shown:



Object	Name Property	Text Property
Label	nameLabel	Enter Name:
TextBox	nameTextBox	
Button	okButton	OK
Button	cancelButton	Cancel

Click  to save your changes.

We also need to add the Button event handlers, and change some of the Form properties to turn the Form into the type of dialog box we want (and explore more of the Form properties along the way).

Double-click each Button to generate an event handler. Modify the Popup Form properties in the Properties Window using the list below. Remember to click on the Popup Form in the Visual Designer to select it. Change each property to the specified value. When you finish, click  to save your changes.

Property	Value
AcceptButton	okButton
CancelButton	cancelButton
ControlBox	False
FormBorderStyle	FixedDialog
ShowInTaskbar	False
StartPosition	CenterParent

Note When you try to set the **AcceptButton** and **CancelButton** properties, if you do not find the **Value** listed in the table, you need add the event handler for the Button.

So what exactly do each of these properties do? Rather than list out what each property does, let's *learn by doing*.

In the Properties Window, click on each of the Popup Form properties listed above, then look below the Properties Window to find the **Properties Help Window** that describes each property. For example, for the **AcceptButton** property, you should see:

The accept button of the form. If this is set, the button is 'clicked' whenever the user presses the 'ENTER' key.

Now, let's go back to the **FormsIO** Form and call the Popup dialog box Form.

Select the **FormsIO.cs** Code Editor tab, and modify the code as shown below.

FormsIO.cs

```
using ...

namespace FormsIO
{
    public partial class FormsIO : Form
    {
        public FormsIO()
        {
            InitializeComponent();
        }

        private void dialogButton_Click(object sender, EventArgs e)
        {
            Popup myPopup = new Popup();
            myPopup.ShowDialog();
        }
    }
}
```



and to run the program. Click **Dialog** to see the dialog box we created. You can't really do much of anything with it yet, so click **OK** or **Cancel**, and then close the

Let's discuss how this code works.

OBSERVE: FormsIO.cs

```
.
.
.

    private void dialogButton_Click(object sender, EventArgs e)
    {
        Popup myPopup = new Popup();
        myPopup.ShowDialog();
    }

.
.
.
```

Now that we've created the **Popup** Form, we can use that Form as a data type, declaring the variable **myPopup**, and creating an instance of the **Popup** Form. We then call the **ShowDialog()** method of the **myPopup** Form, which will display our new dialog box.

Next, we need to modify the **Popup** Form to act like a dialog box and return the button selected entered by the user.

Switch to the **Popup.cs [Design]** tab. Click the **okButton** and change the DialogResult property to **OK**. Click the **cancelButton** and change the DialogResult property to **Cancel**. Then, modify the code in **Popup.cs** as shown below.

Popup.cs

```
using ...

namespace FormsIO
{
    public partial class Popup : Form
    {
        private string _userName = "";

        public string UserName
        {
            get { return _userName; }
            set { _userName = value; }
        }

        public Popup()
        {
            InitializeComponent();
        }

        private void okButton_Click(object sender, EventArgs e)
        {
            _userName = nameTextBox.Text;
            this.Close();
        }
    }
}
```



to save your changes. You can still run it without error, but it won't look any different yet. Let's discuss the new code.

OBSERVE: Popup.cs

```
.
.
.
    private string _userName = "";

    public string UserName
    {
        get { return _userName; }
        set { _userName = value; }
    }
.
.
.
    private void okButton_Click(object sender, EventArgs e)
    {
        _userName = nameTextBox.Text;
        this.Close();
    }
}
```

We added a private variable **_userName** to hold whatever the user might enter. We also added accessors for this variable. If the user clicks the **okButton**, we will set the **_userName** variable and **Close()** the Form. (We don't need to handle the Cancel event; the **CancelButton** took care of that for us.

Finally, we need to modify the **FormsIO** Form to be able to determine which button was clicked, and retrieve the information entered.

Select the **FormsIO.cs** Form Code Editor, and modify the code as shown below.

FormsIO.cs

```
using ...

namespace FormsIO
{
    public partial class FormsIO : Form
    {
        public FormsIO()
        {
            InitializeComponent();
        }

        private void dialogButton_Click(object sender, EventArgs e)
        {
            Popup myPopup = new Popup();

            myPopup.ShowDialog();
            if (myPopup.ShowDialog() == DialogResult.OK)
            {
                hiLabel.Text = "Hi, " + myPopup.UserName;
            }
            else
            {
                hiLabel.Text = "So don't tell me your name.";
            }
        }
    }
}
```



and to run the program. Click **Dialog**, enter your name, and click **OK**, and "Hi, your name" should appear in the form. Click **Dialog** and click **Cancel**, and "So don't tell me your name." should appear in the form.

Let's discuss how this code works.

OBSERVE: FormsIO.cs

```
.
.
.
    private void dialogButton_Click(object sender, EventArgs e)
    {
        Popup myPopup = new Popup();

        if (myPopup.ShowDialog() == DialogResult.OK)
        {
            hiLabel.Text = "Hi, " + myPopup.UserName;
        }
        else
        {
            hiLabel.Text = "So don't tell me your name.";
        }
    }
.
.
.
```

We've modified the FormsIO code to check if the `myPopup.ShowDialog()` method returns a value of `DialogResult.OK`, indicating the user clicked on the `okButton` on the `Popup` Form. If the `okButton` was clicked, then we can go ahead and access the data the user entered (if any) by calling the `myPopup.UserName` method. If not (they clicked the `cancelButton`), then we'll just have to let them know.

So, that's how to add your own custom dialog box!

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click

Hand it in at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

File Input and Output

A major use of programs is accessing files: seeing if a file exists, creating and deleting files, reading from and writing to files. Previously, we've used the `System.IO.FileInfo` class to retrieve information about a file (it will also work for a directory). We will need to use this object again to see if a file exists. We'll add a few new classes to enable us to work with files.

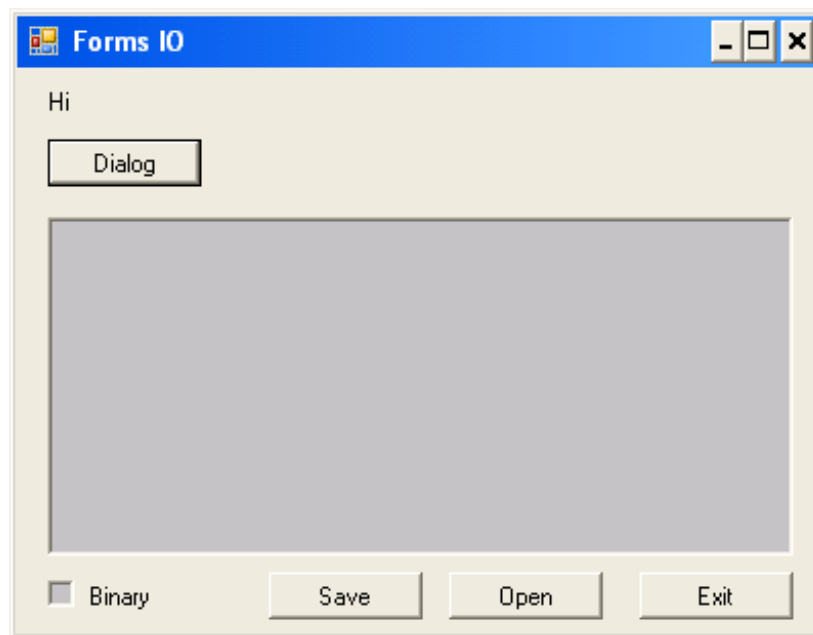
Files that we humans can read or write are called *text* (*plain text*, *ASCII*) files; or *binary* files. Files that can only be read by a computer are called *binary* files. Files that contain formatting information, like word-processing documents, typically are binary. All files, actually, are binary files, as all data stored on a computer can be reduced to the binary zeros and ones the computer reads, and we can read text files with the same C# classes that handle binary data. However, we have other classes we can use with text files. Let's explore each by extending our **FormsIO** project.

Note

You might wonder if we'll return to a Console Application for files. We won't, because file handling works the same whether you have a **Console Application** or a **Windows Forms Application**. Of course, you can't use some of the nicer Windows GUI controls with a Console Application, so we'll stick with a Windows Forms Application for this project.

Using File Object

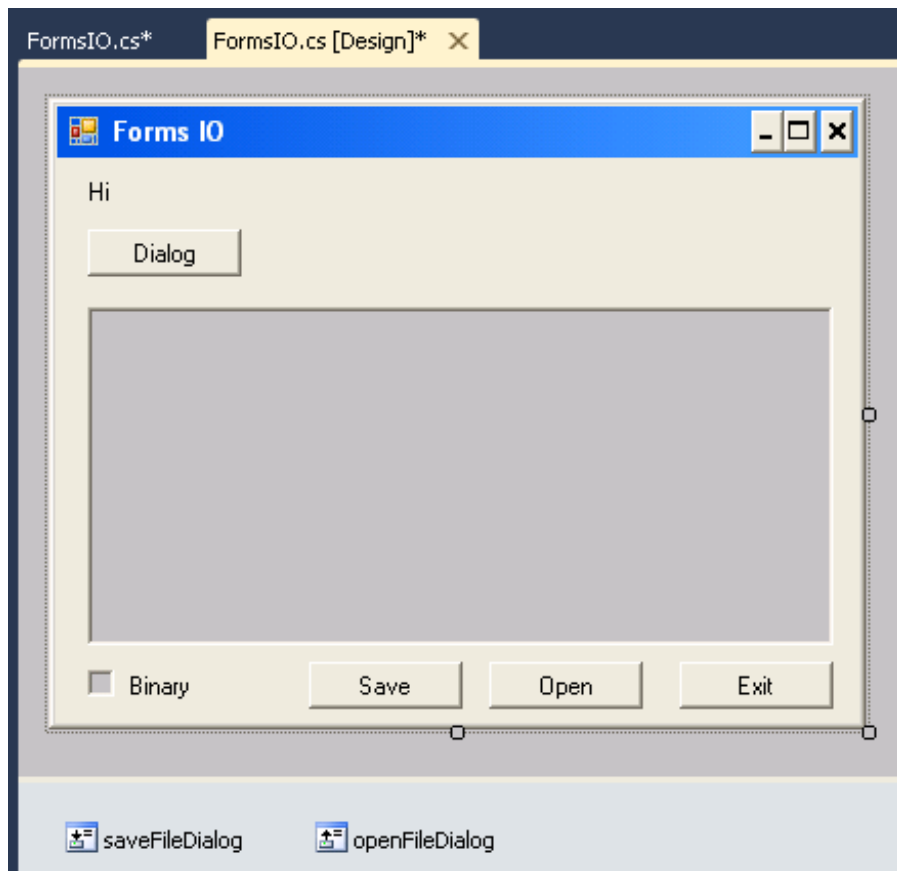
Modify the **FormsIO** form as shown below by adding the controls listed. Change each control Name property to match the Name Property column, and each Text property to match the Text Property column. Set the Multiline property of the `contentTextBox` control to **True** so you can expand it vertically. Double-click on each of the three buttons to add event handlers.



Object	Name Property	Text Property
TextBox	contentTextBox	
CheckBox	binaryCheckBox	Binary
Button	saveButton	Save
Button	openButton	Open
Button	exitButton	Exit

Next, let's add dialog box controls from the Toolbox that will assist us in saving and opening files. Note that you can drag these anywhere on the form, and they'll appear under the form.

From the **Dialogs** section of the Toolbox, drag a **SaveFileDialog** onto the **FormsIO** form. Change the Name property to **saveFileDialog**. Drag an **OpenFileDialog** onto the form. Change the Name property to **openFileDialog**:



Next, we'll add code to bring up the dialog controls when the user clicks on the buttons, code to exit the application, and a couple of C# constants we'll use with the dialogs.

Select the **FormsIO.cs** Form Code Editor tab, and modify the code as shown:

FormsIO.cs

```
using ...

namespace FormsIO
{
    public partial class FormsIO : Form
    {
        private const string TextFilters = "txt files (*.txt)|*.txt";
        private const string BinaryFilters = "All files (*.*)|*.*";

        public FormsIO()
        {
            InitializeComponent();
        }

        private void dialogButton_Click(object sender, EventArgs e)
        {
            Popup myPopup = new Popup();

            if (myPopup.ShowDialog() == DialogResult.OK)
            {
                hiLabel.Text = "Hi, " + myPopup.UserName;
            }
            else
            {
                hiLabel.Text = "So don't tell me your name.";
            }
        }

        private void saveButton_Click(object sender, EventArgs e)
        {
            saveFileDialog.InitialDirectory = Application.StartupPath;
            saveFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

            saveFileDialog.FilterIndex = 1;
            saveFileDialog.RestoreDirectory = true;
            saveFileDialog.FileName = "";
            saveFileDialog.Title = "Save File";

            if (saveFileDialog.ShowDialog() == DialogResult.OK)
            {
                // Save the file.
            }
        }

        private void openButton_Click(object sender, EventArgs e)
        {
            openFileDialog.InitialDirectory = Application.StartupPath;
            openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

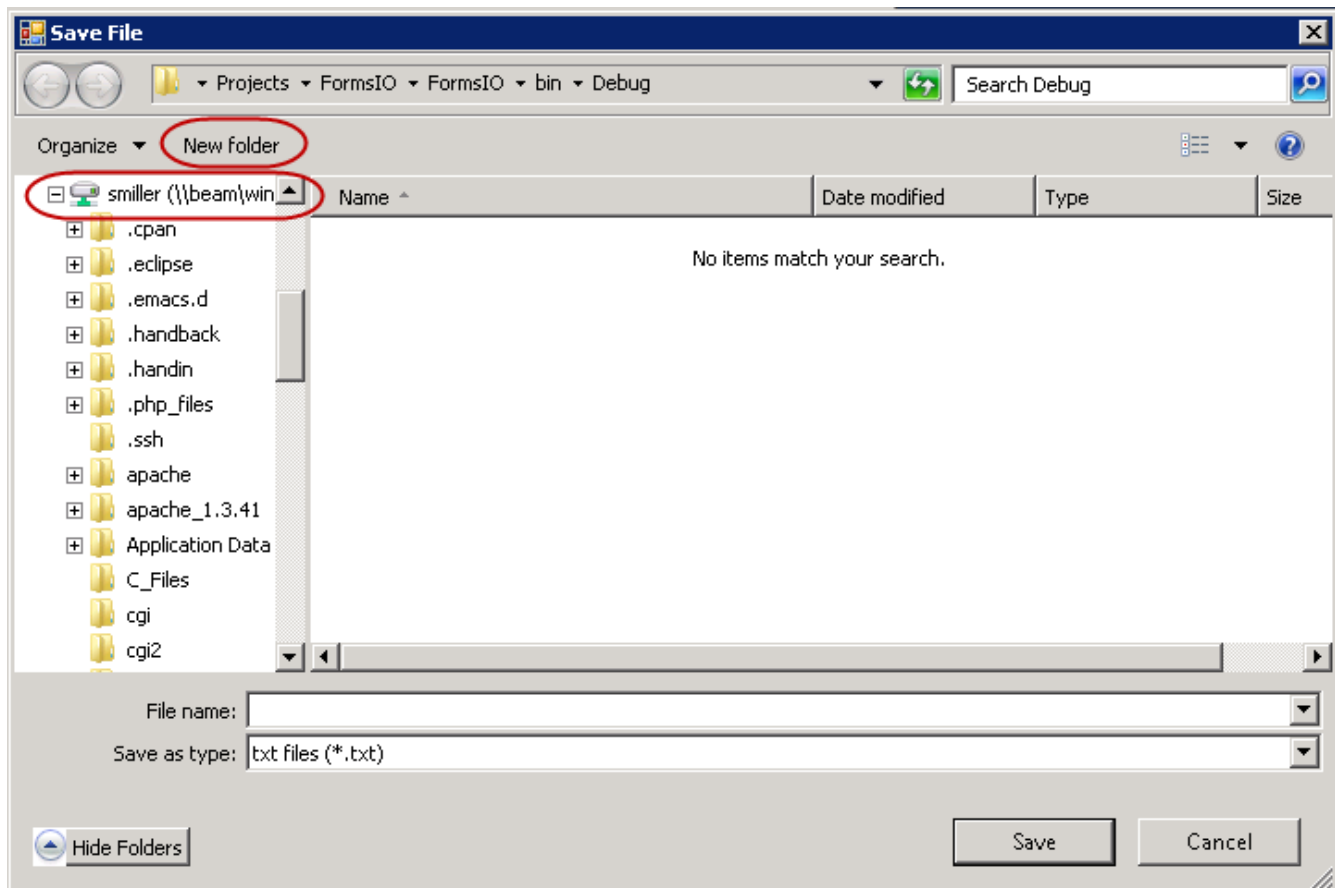
            openFileDialog.FilterIndex = 1;
            openFileDialog.RestoreDirectory = true;
            openFileDialog.FileName = "";
            openFileDialog.Title = "Open File";
            openFileDialog.Multiselect = false;

            if (openFileDialog.ShowDialog() == DialogResult.OK)
            {
                // Open the file.
            }
        }

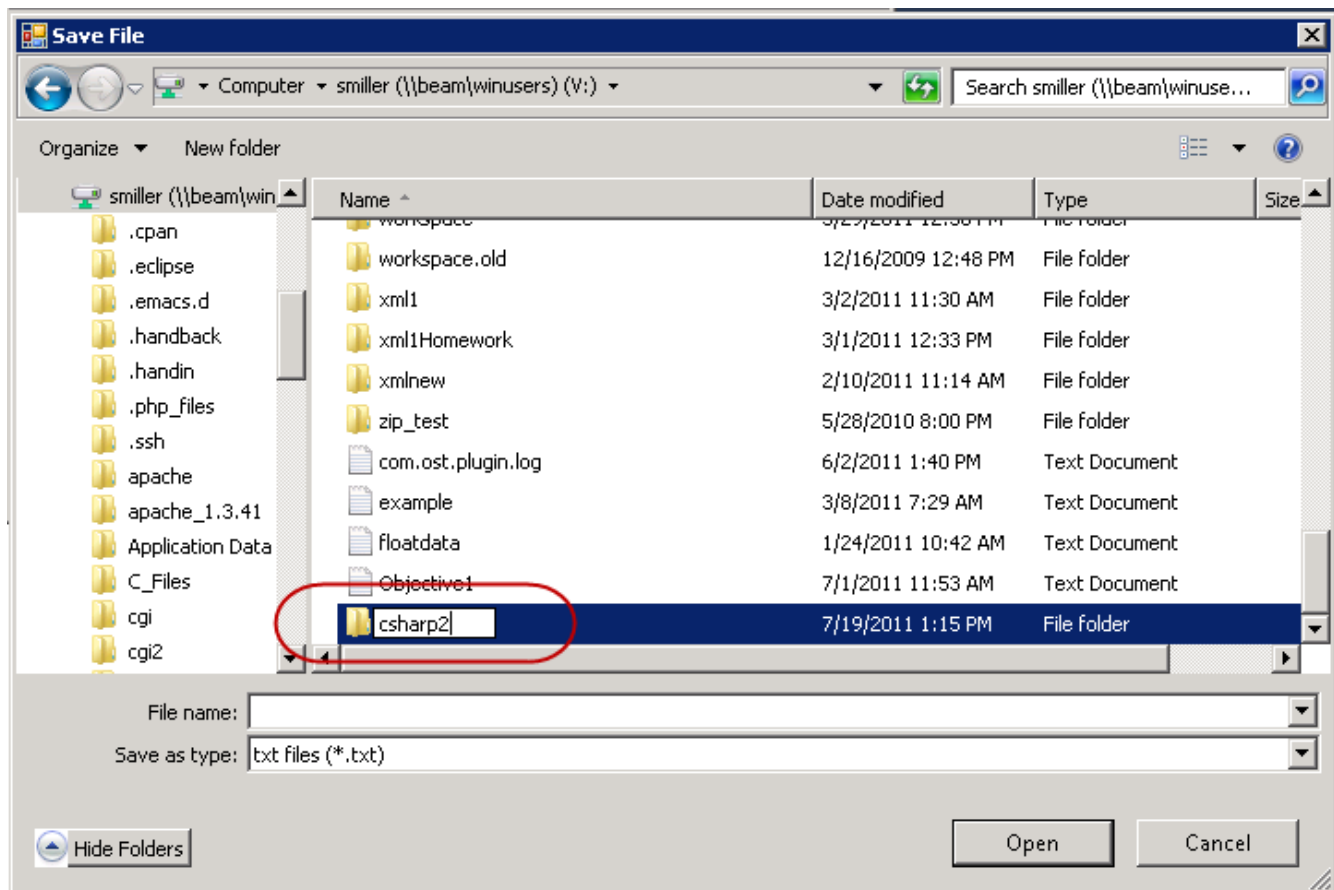
        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

```
}  
}  
}
```

 and  to run the program. We need to create a folder for the files we'll be using, so click the **Save** button on the form. In the Save File dialog that appears, select the folder with your user name (in place of "smiller" in the screen shot) under My Computer, and then click **New folder**:



Enter the folder name **csharp2** and press **Enter**:



We don't have code in place yet to save a file, but you can see that C# saves us a lot of work in providing the dialog for saving, and we now have a working folder for the rest of the lesson. So click **Cancel**, and then **Exit**. Let's discuss the new code.

OBSERVE: FormsIO.cs

```

.
.
.
public partial class FormsIO : Form
{
    private const string TextFilters = "txt files (*.txt)|*.txt";
    private const string BinaryFilters = "All files (*.*)|*.*";
.
.
.

    private void saveButton_Click(object sender, EventArgs e)
    {
        saveFileDialog.InitialDirectory = Application.StartupPath;
        saveFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

        saveFileDialog.FilterIndex = 1;
        saveFileDialog.RestoreDirectory = true;
        saveFileDialog.FileName = "";
        saveFileDialog.Title = "Save File";

        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            // Save the file.

        }
    }

    private void openButton_Click(object sender, EventArgs e)
    {
        openFileDialog.InitialDirectory = Application.StartupPath;
        openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

        openFileDialog.FilterIndex = 1;
        openFileDialog.RestoreDirectory = true;
        openFileDialog.FileName = "";
        openFileDialog.Title = "Open File";
        openFileDialog.Multiselect = false;

        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
            // Open the file.

        }
    }
.
.
.

```

The added code implements the setup and display of the **saveFileDialog** and **openFileDialog** dialog boxes. We use the **Application.StartupPath** variable that contains the full application path as the default folder for the dialog boxes. We also use, again, the ternary/conditional operator to set the dialog filters. The dialog filters determine what is available in the Files To List combo box for each of the dialog boxes. We also test to make sure that we only process either dialog if the **Open** or **Save** button is clicked, which is the equivalent of an OK button, by testing the result of calling the **ShowDialog()** method against **DialogResult.OK**. Now let's add the code to open or save the text within the **contentTextBox**.

Select the **FormsIO.cs** form Code Editor tab, and modify the code as shown below (we'll again just show the relevant areas, as our code is getting large).

FormsIO.cs

```

using ...
using System.IO
.
.
.

    private void saveButton_Click(object sender, EventArgs e)
    {
        saveFileDialog.InitialDirectory = Application.StartupPath;
        saveFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

        saveFileDialog.FilterIndex = 1;
        saveFileDialog.RestoreDirectory = true;
        saveFileDialog.FileName = "";
        saveFileDialog.Title = "Save File";

        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            // Save the file.
            if (saveFileDialog.FileName.Length > 0)
            {
                if (!binaryCheckBox.Checked)
                {
                    File.WriteAllText(saveFileDialog.FileName, contentTextBox.Text)
;

                }
                else
                {
                    UnicodeEncoding encoding = new UnicodeEncoding();
                    File.WriteAllBytes(saveFileDialog.FileName, encoding.GetBytes(c
ontentTextBox.Text));
                }
                MessageBox.Show(Path.GetFileName(saveFileDialog.FileName) + " saved
", this.Text,
                                MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
        }

        private void openButton_Click(object sender, EventArgs e)
        {
            openFileDialog.InitialDirectory = Application.StartupPath;
            openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

            openFileDialog.FilterIndex = 1;
            openFileDialog.RestoreDirectory = true;
            openFileDialog.FileName = "";
            openFileDialog.Title = "Open File";
            openFileDialog.Multiselect = false;

            if (openFileDialog.ShowDialog() == DialogResult.OK)
            {
                // Open the file.
                if (openFileDialog.FileName.Length > 0)
                {
                    if (!binaryCheckBox.Checked)
                    {
                        contentTextBox.Text = File.ReadAllText(openFileDialog.FileName)
;

                    }
                    else
                    {
                        UnicodeEncoding encoding = new UnicodeEncoding();
                        contentTextBox.Text = encoding.GetString(File.ReadAllBytes(open
FileDialog.FileName));
                    }
                    MessageBox.Show(Path.GetFileName(openFileDialog.FileName) + " load

```

```

d", this.Text,
                                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
.
.
.

```

 and  to run the program. Enter some text in the contentTextBox and click **Save**; in the Save File dialog that appears, navigate to the **csharp2** folder we created (under **Computer | username**); enter the file name **MyTextFile.txt**, and click **Save**. You'll see a confirmation that the file was saved. Now, enter different text, check the **Binary** box, and click **Save** again; in the Save File dialog, again navigate to the **csharp2** folder, enter the file name **MyBinaryFile.bin**, and click **Save**. Again you'll see the confirmation. You can use the **Open** button now to open either of the files—note that when **Binary** is checked, you see both files, and when it isn't checked, you only see the .txt file.

Let's discuss how this code works.

OBSERVE: FormsIO.cs

```

using...
using System.IO;
.
.
.

private void saveButton_Click(object sender, EventArgs e)
{
    saveFileDialog.InitialDirectory = Application.StartupPath;
    saveFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilter
s;

    saveFileDialog.FilterIndex = 1;
    saveFileDialog.RestoreDirectory = true;
    saveFileDialog.FileName = "";
    saveFileDialog.Title = "Save File";

    if (saveFileDialog.ShowDialog() == DialogResult.OK)
    {
        // Save the file.
        if (saveFileDialog.FileName.Length > 0)
        {
            if (!binaryCheckBox.Checked)
            {
                File.WriteAllText(saveFileDialog.FileName, contentTextBox.Text)
;
            }
            else
            {
                UnicodeEncoding encoding = new UnicodeEncoding();
                File.WriteAllBytes(saveFileDialog.FileName, encoding.GetBytes(c
ontentTextBox.Text));
            }
            MessageBox.Show(Path.GetFileName(saveFileDialog.FileName) + " save
d", this.Text,
                                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
.
.
.

```

The code is similar for both dialogs. First, We make sure a file was specified by checking that the length of **FileName** is greater than zero. If **Binary** is *not* checked, we save the contents of the **contentTextBox** to the file using the **File.WriteAllText** method. If **Binary** is checked, we use create an instance of the **UnicodeEncoding** class and use **File.WriteAllBytes** to write the contents of the TextBox as binary data. We use the **MessageBox** object to notify the

user that the file was either saved or loaded. We also use the **Path.GetFileName** method to extract just the file name component of the **FileName** dialog box property (which contains the complete path and filename).

Try changing the notification so that it gives the full path and filename.

Cool stuff! Start thinking about other ways you could use this kind of functionality.

Before moving on to the next topic, we want to point you to a fantastic web page resource by Microsoft that lists links to C# tips, solutions, and examples to most common input/output (I/O) tasks: [Common I/O Tasks](#). Review the list of topics to get an idea of the types of issues included.

Using Streams

Although using the File object works well, the general concept of reading and writing data can apply to many other scenarios beyond just a file; for example, to and from a network, to and from an object, and to and from a database. Potentially, you could envision other File classes for each of these data sources, but they all share common functionality related to data input and output. To encompass that commonality, programming languages evolved to include the concept of a *stream*. Using streams, we can simplify our coding to use similar methods even though the data sources differ. Let's rewrite our previous code using streams.

Select the **FormsIO** form Code Editor tab, and modify the `openButton_Click()` method as shown below.

FormsIO.cs: openButton_Click method

```
private void openButton_Click(object sender, EventArgs e)
{
    openFileDialog.InitialDirectory = Application.StartupPath;
    openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilters;
    openFileDialog.FilterIndex = 1;
    openFileDialog.RestoreDirectory = true;
    openFileDialog.FileName = "";
    openFileDialog.Title = "Open File";
    openFileDialog.Multiselect = false;

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        // Open the file.
        if (openFileDialog.FileName.Length > 0)
        {
            if (!binaryCheckBox.Checked)
            {
contentTextBox.Text = File.ReadAllText(openFileDialog.FileName);
                // Create a stream connected to a file.
                FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);

                // Create a text stream reader connected to the file stream.
                StreamReader streamReader = new StreamReader(fileStream);
                // Read the data to the end of the stream.
                contentTextBox.Text = streamReader.ReadToEnd();
                // Close the text stream reader.
                streamReader.Close();
                // Close the file stream.
                fileStream.Close();
            }
            else
            {
UnicodeEncoding encoding = new UnicodeEncoding();
contentTextBox.Text = encoding.GetString(File.ReadAllBytes(openFileDialog.FileName));
                // Create a stream connected to a file.
                FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);

                // Create a binary stream reader connected to the file stream.
                BinaryReader binaryReader = new BinaryReader(fileStream);
                // Get the length of the file in bytes.
                long fileLength = new FileInfo(openFileDialog.FileName).Length;
                // Create a byte array to hold the byte data.
                byte[] byteArray = new byte[fileLength];
                // Read the specified amount of data from the binary stream.
                byteArray = binaryReader.ReadBytes((int)fileLength);
                // Prepare an encoding object for translating the data read from the file.
                UTF8Encoding encoding = new UTF8Encoding();
                // Convert the byte data into a format we can read.
                contentTextBox.Text = encoding.GetString(byteArray);
                // Close the binary stream reader.
                binaryReader.Close();
                // Close the file stream.
                fileStream.Close();
            }
            MessageBox.Show(Path.GetFileName(openFileDialog.FileName) + " loaded", this
                .Text,
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
```

 and  to run the program. It should work exactly as before.

Let's discuss the changes.

OBSERVE: FormsIO.cs openButton_Click method

```
.
.
.
        if (!binaryCheckBox.Checked)
        {
            // Create a stream connected to a file.
            FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);

            // Create a text stream reader connected to the file stream.
            StreamReader streamReader = new StreamReader(fileStream);
            // Read the data to the end of the stream.
            contentTextBox.Text = streamReader.ReadToEnd();
            // Close the text stream reader.
            streamReader.Close();
            // Close the file stream.
            fileStream.Close();
        }
        else
        {
            // Create a stream connected to a file.
            FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);

            // Create a binary stream reader connected to the file stream.
            BinaryReader binaryReader = new BinaryReader(fileStream);
            // Get the length of the file in bytes.
            long fileLength = new FileInfo(openFileDialog.FileName).Length;
            // Create a byte array to hold the data.
            byte[] byteArray = new byte[fileLength];
            // Read the specified amount of data from the binary stream.
            byteArray = binaryReader.ReadBytes((int)fileLength);
            // Prepare an encoding object for translating the data read from the file.
            UTF8Encoding encoding = new UTF8Encoding();

            // Convert the byte data into a format we can read.
            contentTextBox.Text = encoding.GetString(byteArray);
            // Close the binary stream reader.
            binaryReader.Close();
            // Close the file stream.
            fileStream.Close();
        }
.
.
.
```

Don't be overwhelmed! Yes, we've added quite a few changes to the **openButton_Click** method (we also added copious single-line comments to help!), but we'll break this up into small chunks to explain them all! The general steps are:

1. Create a stream connected to a source.
2. Create a stream reader object.
3. Connect stream reader to our stream source.
4. Read as much of the data from the stream we want.
5. For binary: determine how to display data as necessary.
6. Close stream reader connection.
7. Close stream source connection.

Now, line by line in the code:

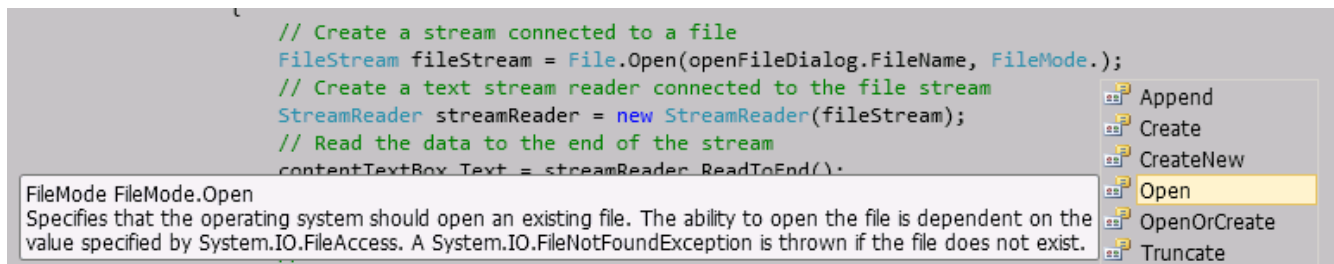
OBSERVE:

```
// Create a stream connected to a file.
FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);
```

For both branches of the *if* statement, we use the same code (above), which creates a **FileStream** object variable of type **FileStream**, using the **Open** method of the **File** object. Notice that when we use the **Open** method, we have to specify the access mode. For our needs, we use the **FileMode.Open** mode, assuming the file should already exist because we're using the **OpenFileDialog** object.

Note

Remember, you can use Intellisense to help you in learning more about objects and types in Studio by typing the object type or variable name, then the period, and waiting for Intellisense to give you a dropdown list of the options. You can use the arrow keys to move through the options, and pause again, waiting for Intellisense to give you an additional popup of information about the highlighted option. Below are the different **FileMode** access modes, with the **Open** mode highlighted.

**OBSERVE:**

```
// Create a text stream reader connected to the file stream.
StreamReader streamReader = new StreamReader(fileStream);
// Read the data to the end of the stream.
contentTextBox.Text = streamReader.ReadToEnd();
```

Next for text files, we create a **stream reader** object to enable reading data from a stream, and connect our stream reader to the stream source (a file, in our example). We use a **StreamReader** object for reading our stream data as text. Then, we read the data using the **ReadToEnd** method of the **streamReader** object.

OBSERVE:

```
// Create a binary stream reader connected to the file stream.
BinaryReader binaryReader = new BinaryReader(fileStream);
// Get the length of the file in bytes.
long fileLength = new FileInfo(openFileDialog.FileName).Length;
// Create a byte array to hold the byte data.
byte[] byteArray = new byte[fileLength];
// Read the specified amount of data from the binary stream.
byteArray = binaryReader.ReadBytes((int)fileLength);
```

For binary data, we use a **BinaryReader** object. The method differs for reading text and binary data, as binary data by definition may not be printable, but the non-printable nature is not the only difference: text data, as read, has a meaning, such as strings, but binary data is typically read as bytes, and can be interpreted as string, numerical data, or images, encrypted information, etc. So, for binary, we need to read the data in as *byte* data. We store the data in an object called an array (we'll discuss arrays later) using the variable **byteArray**. For now, just understand that the **byteArray** contains the characters you typed in the **TextBox**, translated one by one to computer language; for example, if you typed "now is the time," the **byteArray** would contain a table of decimal values for n, o, w, space, i, s, space, etc. We need to know how many of these bytes to read, and to size the array appropriately, so we use the **Length** property of the **FileInfo** object to set the variable **fileLength**. We use the **ReadBytes** method of the **binaryReader** object to read the data.

Note

You may note that we used **(int)** to convert the **fileLength** variable from a long to an int. Why? The **FileInfo Length** property returns a long, but the **BinaryReader ReadBytes** method requires an int. The fact that the **FileInfo Length** method returns a long, and the **BinaryReader ReadBytes** method uses an int, indicates that the **ReadBytes** method is not capable of reading very large data streams. Reading a large amount of data is a trade-off between efficiency and performance. The more data you can read at once is general more efficient, but very large amounts might cause your program to slow down during the read process. You'll find that frequently reading from a stream is an iterative process.

OBSERVE:

```
// Prepare an encoding object for translating the data read from the file.  
UTF8Encoding encoding = new UTF8Encoding();
```

We next see that we create another **encoding** object, but rather than Unicode, this time we create one of type **UTF8Encoding**. Actually, UTF8 is Unicode, but a specific version that allows for backward compatibility with ASCII. (In a later lesson, we'll get more into character sets and encoding.) We use the UTF8 **encoding** object to convert the individual bytes in the **byteArray** into *string* data we can read.

OBSERVE:

```
// Close the text stream reader.  
streamReader.Close();  
// Close the file stream.  
fileStream.Close();
```

OBSERVE:

```
// Close the binary stream reader.  
binaryReader.Close();  
// Close the file stream.  
fileStream.Close();
```

Finally, for both types of file, we close out our stream readers, **streamReader** and **binaryReader**, and stream objects **fileStream**, by calling the appropriate **Close()** method. When opening stream objects and readers, always be careful to close them when you finish using them; typically, you should close them as soon as you can. Opening streams can place locks on the stream sources, and you want to make sure you don't tie up resources. In fact, closing these objects is so important that C# includes a syntax, **using**, that ensures that limited-resource objects you create have limited scope. Once these objects go out of scope, their **Close()** methods are automatically called. Let's rewrite our code to take advantage of this syntax. We'll also use the **FileStream** object inline, which will negate the need for declaring the **fileStream** variable, and eliminate the **byteArray** variable.

Modify the **FormsIO.cs openButton_Click** method as shown below.

FormsIO.cs: openButton_Click method

```

private void openButton_Click(object sender, EventArgs e)
{
    openFileDialog.InitialDirectory = Application.StartupPath;
    openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilters;
    openFileDialog.FilterIndex = 1;
    openFileDialog.RestoreDirectory = true;
    openFileDialog.FileName = "";
    openFileDialog.Title = "Open File";
    openFileDialog.Multiselect = false;

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        // Open the file.
        if (openFileDialog.FileName.Length > 0)
        {
            if (!binaryCheckBox.Checked)
            {
// Create a stream connected to a file.
FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);
n);

                // Create a text stream reader connected to the file stream.
StreamReader streamReader = new StreamReader(fileStream);
                using (StreamReader streamReader = new StreamReader(File.Open(openFileDialog.FileName, FileMode.Open)))
                {
                    // Read the data to the end of the stream.
                    contentTextBox.Text = streamReader.ReadToEnd();
                }
// Close the text stream reader.
streamReader.Close();
// Close the file stream.
fileStream.Close();
            }
            else
            {
// Create a stream connected to a file.
FileStream fileStream = File.Open(openFileDialog.FileName, FileMode.Open);
n);

                // Create a binary stream reader connected to the file stream.
                using (BinaryReader binaryReader = new BinaryReader(File.Open(openFileDialog.FileName, FileMode.Open)))
                {
                    // Get the length of the file in bytes.
                    long fileLength = new FileInfo(openFileDialog.FileName).Length;
// Create a byte array to hold the byte data.
byte[] byteArray = new byte[fileLength];
// Read the specified amount of data from the binary stream.
byteArray = binaryReader.ReadBytes((int)fileLength);
// Prepare an encoding object for translating the data read from the
file.

                    UTF8Encoding encoding = new UTF8Encoding();
                    // Convert the byte data into a format we can read.
contentTextBox.Text = encoding.GetString(byteArray);
                    contentTextBox.Text = encoding.GetString(binaryReader.ReadBytes((int)fileLength));
                }
// Close the binary stream reader.
binaryReader.Close();
// Close the file stream.
fileStream.Close();
            }
            MessageBox.Show(Path.GetFileName(openFileDialog.FileName) + " loaded", this
                .Text,
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}

```

```
}
```



and to run the program. It should work just like before.

Let's discuss the new code.

OBSERVE: FormsIO.cs openButton_Click method

```
private void openButton_Click(object sender, EventArgs e)
{
    openFileDialog.InitialDirectory = Application.StartupPath;
    openFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilters;
    openFileDialog.FilterIndex = 1;
    openFileDialog.RestoreDirectory = true;
    openFileDialog.FileName = "";
    openFileDialog.Title = "Open File";
    openFileDialog.Multiselect = false;

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        // Open the file.
        if (openFileDialog.FileName.Length > 0)
        {
            if (!binaryCheckBox.Checked)
            {
                // Create a text stream reader connected to the file stream.
                using (StreamReader streamReader = new StreamReader(File.Open(openFileDialog.FileName, FileMode.Open)))
                {
                    // Read the data to the end of the stream.
                    contentTextBox.Text = streamReader.ReadToEnd();
                }
            }
            else
            {
                // Create a binary stream reader connected to the file stream.
                using (BinaryReader binaryReader = new BinaryReader(File.Open(openFileDialog.FileName, FileMode.Open)))
                {
                    // Get the length of the file in bytes.
                    long fileLength = new FileInfo(openFileDialog.FileName).Length;
                    // Prepare an encoding object for translating the data read from the file.
                    UTF8Encoding encoding = new UTF8Encoding();
                    // Convert the byte data into a format we can read.
                    contentTextBox.Text = encoding.GetString(binaryReader.ReadBytes((int)fileLength));
                }
            }
            MessageBox.Show(Path.GetFileName(openFileDialog.FileName) + " loaded", this.Text,
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
```

The **using** syntax allows us to remove the Close() method calls. We **refactored** some of our code to make it more concise and eliminate unneeded variables. *Refactoring* code is a learned process and is very dependent on the programmer's preferences. While concise code is often preferred, you may sometimes trade the conciseness for *readability*. Also, you might think that because the variables aren't declared they don't exist, but each former call that had a variable still has the need to store the results of the method calls somewhere, and C# creates and disposes of temporary variables for the brief time they're needed. If you don't need to reuse the information in a variable, a common practice is to not declare them, but the need for storage for the data is still present.

To complete this section, let's make the necessary changes to save our data using streams.

Modify the **FormsIO saveButton_Click** method as shown below.

FormsIO.cs: saveButton_Click method

```
private void saveButton_Click(object sender, EventArgs e)
{
    saveFileDialog.InitialDirectory = Application.StartupPath;
    saveFileDialog.Filter = binaryCheckBox.Checked ? BinaryFilters : TextFilters;
    saveFileDialog.FilterIndex = 1;
    saveFileDialog.RestoreDirectory = true;
    saveFileDialog.FileName = "";
    saveFileDialog.Title = "Save File";

    if (saveFileDialog.ShowDialog() == DialogResult.OK)
    {
        // Save the file.
        if (saveFileDialog.FileName.Length > 0)
        {
            if (!binaryCheckBox.Checked)
            {
File.WriteAllText(saveFileDialog.FileName, contentTextBox.Text);
                using (StreamWriter streamWriter = new StreamWriter(File.Open(saveFileDialog.FileName, FileMode.OpenOrCreate)))
                {
                    streamWriter.Write(contentTextBox.Text);
                }
            }
            else
            {
                UnicodeEncoding encoding = new UnicodeEncoding();
File.WriteAllBytes(saveFileDialog.FileName, encoding.GetBytes(contentTextBox.Text));
                using (BinaryWriter binaryWriter = new BinaryWriter(File.Open(saveFileDialog.FileName, FileMode.OpenOrCreate)))
                {
                    binaryWriter.Write(encoding.GetBytes(contentTextBox.Text));
                }
            }
            MessageBox.Show(Path.GetFileName(saveFileDialog.FileName) + " saved", this.Text,
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
}
```



and  to run the program. Again, it should work just like before.

Let's discuss the new code.

OBSERVE: FormsIO.cs saveButton_Click method

```
.
.
.
        if (!binaryCheckBox.Checked)
        {
            using (StreamWriter streamWriter = new StreamWriter(File.Open(saveFileDialog.FileName, FileMode.OpenOrCreate)))
            {
                streamWriter.Write(contentTextBox.Text);
            }
        }
        else
        {
            UnicodeEncoding encoding = new UnicodeEncoding();
            using (BinaryWriter binaryWriter = new BinaryWriter(File.Open(saveFileDialog.FileName, FileMode.OpenOrCreate)))
            {
                binaryWriter.Write(encoding.GetBytes(contentTextBox.Text));
            }
        }
.
.
.
```

Using streams to write data is very similar whether we're writing text or binary, although you do have to use different objects: **StreamWriter** for text, and **BinaryWriter** for binary. As with reading, we're again taking advantage of the **using** syntax. We use the **Write** method of either object to save the contents, although for binary we are encoding the data again in the UTF8 format. You might also note that we're using the **OpenOrCreate FileMode** mode, which opens the file if it already exists, or creates it if it doesn't.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **Hand it in** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Models, Classes, and Objects

In this lesson, we'll learn more about using C# as an object-oriented programming (OOP) language to model the real world, to represent that model using classes, to create instances of those classes by creating objects, and to control the interaction with our object model via a specific programming interface. We will create a virtual aquarium program as we work through these concepts. Let's get started!

Object-Oriented Programming Constructs

Just as with other OOP languages, C# includes programming constructs that aid in creating software that represents objects in the real world. Let's discuss these topics on our way to creating our virtual aquarium!

Models: A World View

The word *model* has many definitions related to OOP. Initially, we may consider the typical definition of model as a representation of something in the real world. Just as an architect might construct a small scale model to represent a final, real-world construction, computer programmers also use the idea of a model to represent what is to be coded. The closer the programming model resembles the real world, the easier it is to understand the implementation.

To help us understand the concept, let's consider an aquarium. An aquarium may consist of a tank, the water inside the tank, and fish. Other components might be rocks, pebbles, and other decoration items, as well as an air pump to oxygenate the water. Because we intend to use this model to create a *virtual* aquarium, all we need are the tank and the fish to populate it.

When developing software, we take the idea of a model even further. Software architects use a variety of representational models to assist in the development process. For example, the objects that are involved are typically modeled using Unified Modeling Language (UML) to construct a UML Model Diagram, also known as an object model, or class diagram. We'll be using UML later. We'll introduce other models as the lessons progress; you should note that the word "model" does not have to be part of the description, but that something is being represented by the "model."

Classes: A Blueprint and More

Along with a model, an architect composes a set of *blueprints* to further represent and define what is to be constructed. Software development also includes blueprints, but we call them *classes*. A C# class includes properties related to the object represented, as well as methods to set and retrieve the values of these properties, and other interactions related to the usage of the class as a virtual object.

Now, let's explore these concepts by creating our virtual aquarium!

Example: The Alphabet Aquarium

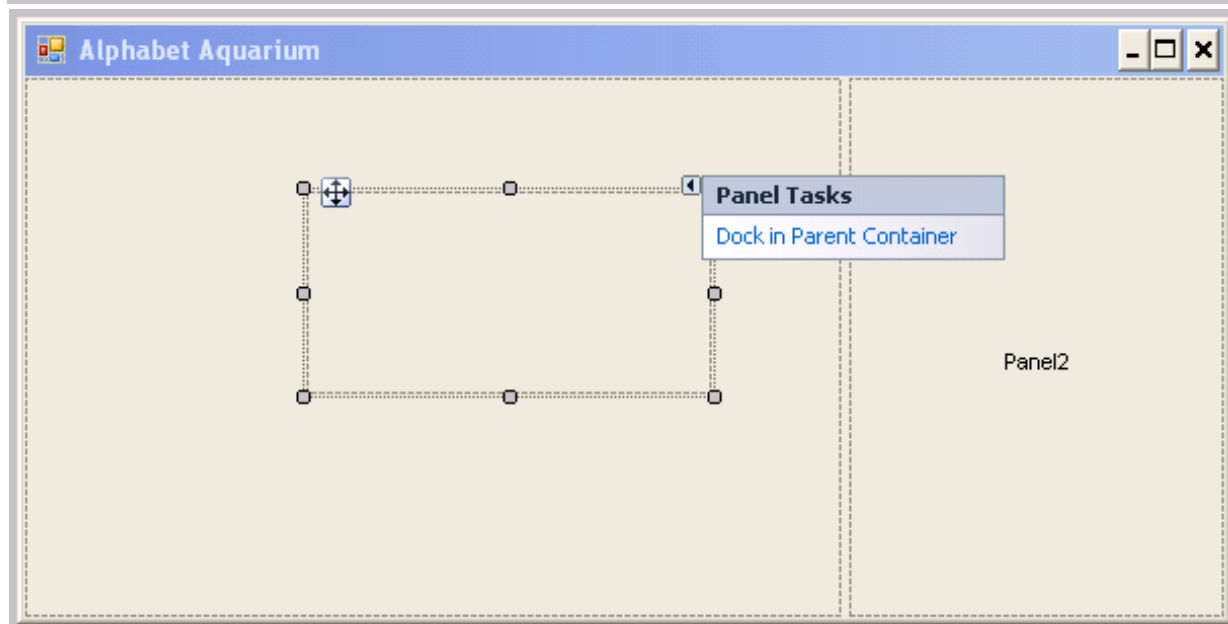
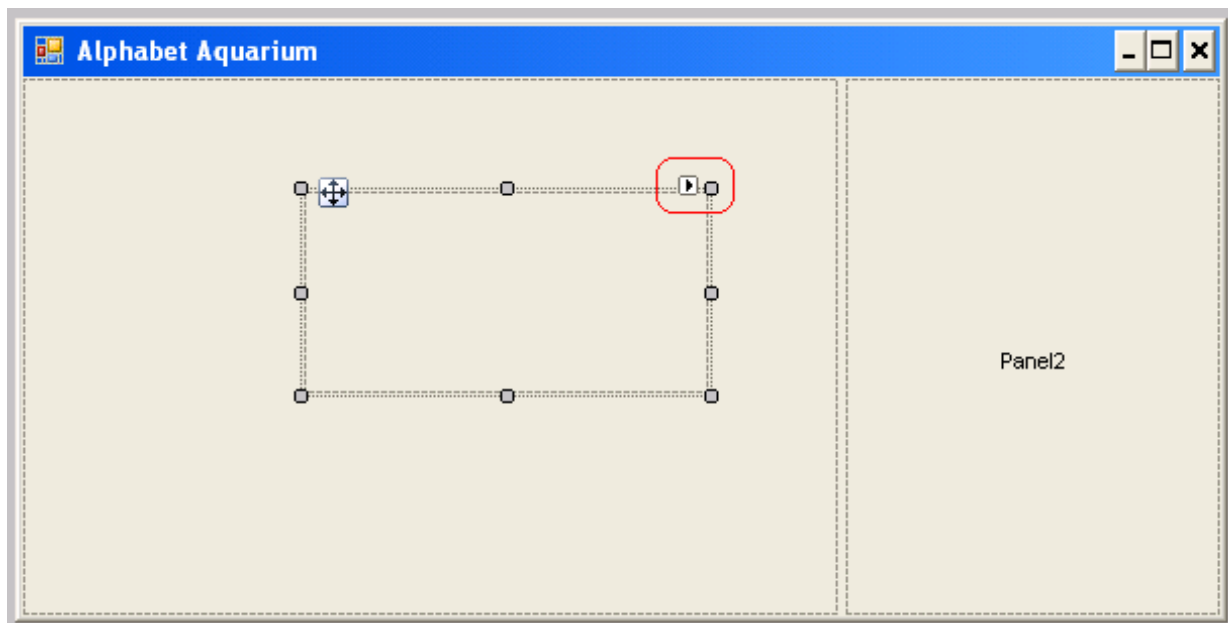
Create a new Visual Studio Windows Forms Application named **AlphabetAquarium**.

Change the name **Form1.cs** to **AlphabetAquarium.cs**. Change the Form 1 Form title bar's Text property to **Alphabet Aquarium**. Click  to save your changes.

We need to add controls to our Form; for this lesson we'll introduce two new controls: *SplitContainer* and *Panel*. We'll use the SplitContainer control to split the Form into two horizontal columns. The left column will be our aquarium, and the right column will be for our other controls. We'll use the Panel control as our actual aquarium. Using the Panel control will make it easier for us to set the color (to represent the water), and to add the fish.

Make the Form about twice as wide by clicking and dragging the right edge to the right. (Or click on the Form, and in the Properties Window change the Width property to **600**. If you have your Property Window sorted by Category, you will find the **Width** property under the **Size** category.)

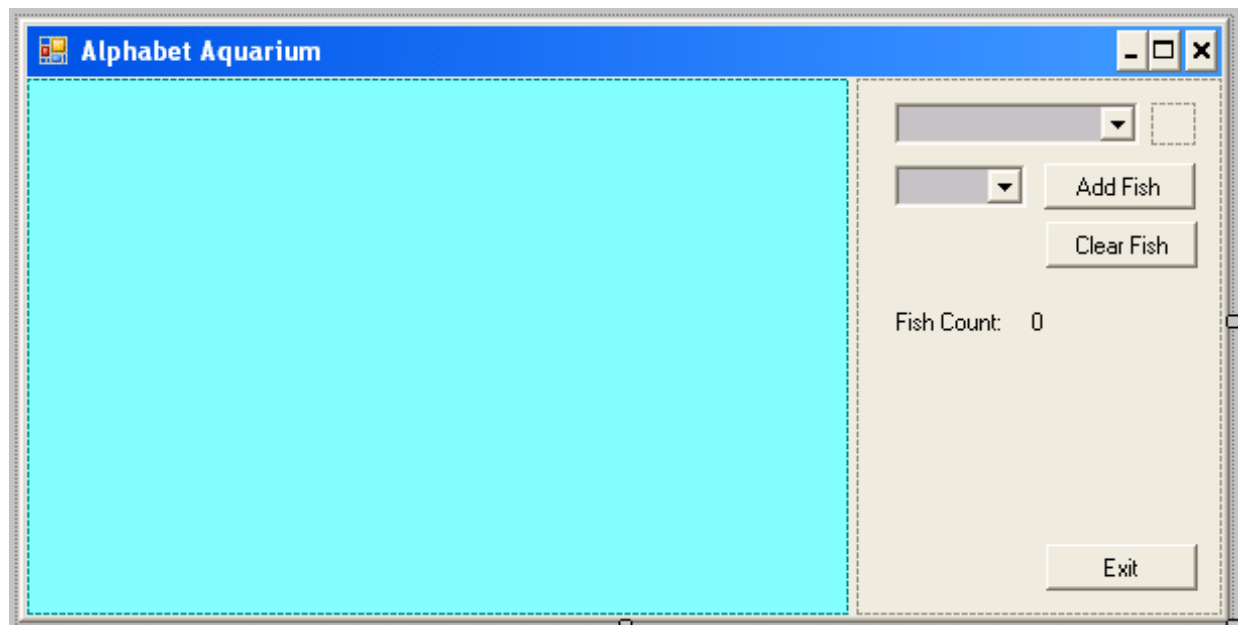
Drag a **SplitContainer** control onto the Form. You do not need to rename the SplitContainer control. Resize the panels within the SplitContainer by clicking and dragging the center line between Panel1 and Panel2, making Panel2 about 1/3 the size of Panel1. Drag a **Panel** control from the Toolbox onto Panel1. Click the small black right arrow at the top right of the Panel to bring up the Panel Tasks popup menu, and select **Dock in Parent Container** to have the Panel fill the entire Panel1 column of the SplitContainer control:



Change the Panel control's Name property to **fishTankPanel**, and the BackColor property to **Cyan** (under the **Web** tab of the Color Dialog box that appears).

Modify Panel2 by adding the controls listed in the first table below. Change each control's Name property to match the Name Property column, and each Text property to match the Text Property column. Arrange the controls as shown.

Object	Name Property	Text Property
ComboBox	colorsComboBox	
PictureBox	colorPictureBox	
ComboBox	fishComboBox	
Button	addFishButton	Add Fish
Button	clearFishButton	Clear Fish
Label	fishLabel	Fish Count:
Label	fishCountLabel	0
Button	exitButton	Exit



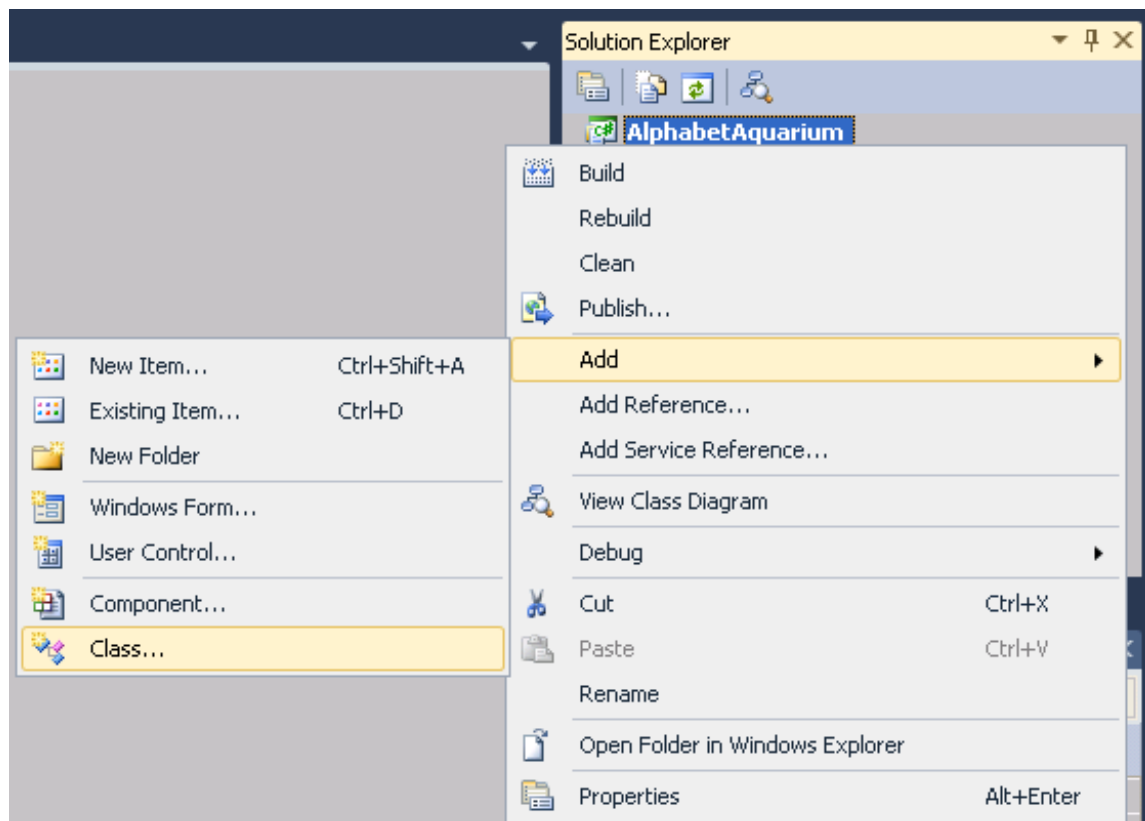
Further modify the ComboBox controls as shown:

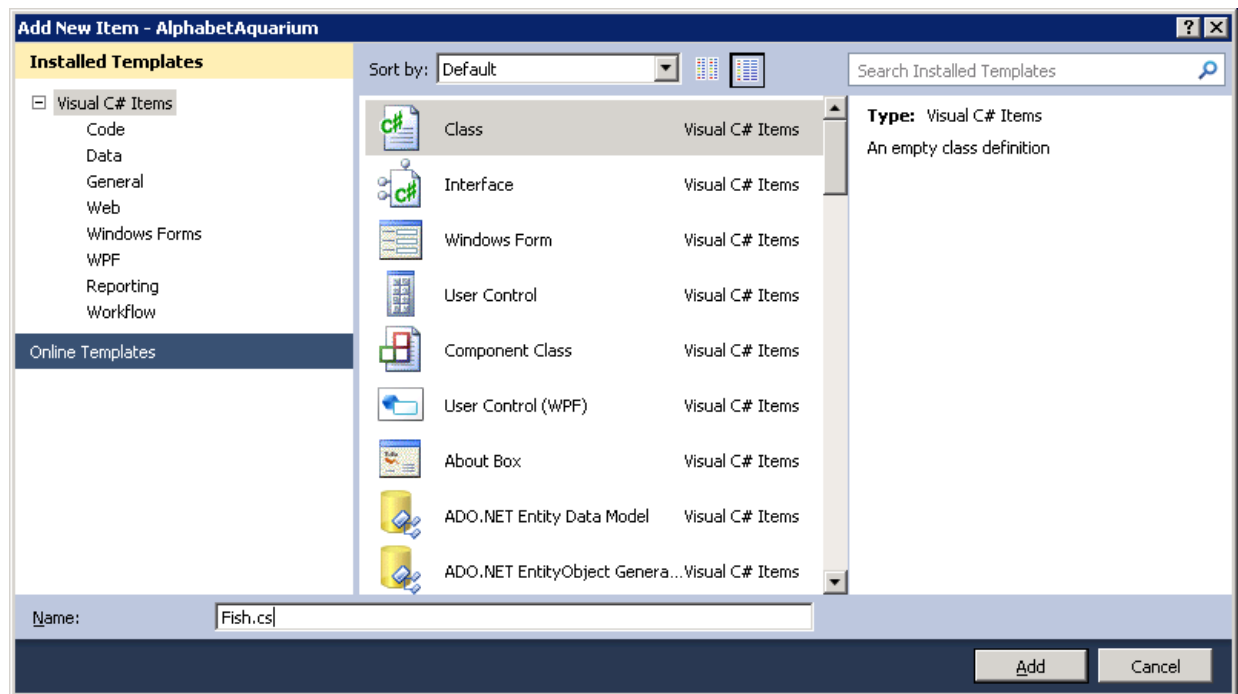
Object	Property	Value
colorsComboBox	DropDownStyle	DropDownList
fishComboBox	DropDownStyle	DropDownList

When you finish, click  to save your changes.

Now that we have the interface created, we need to add the appropriate C# code. Earlier, we introduced the *model components* for this project: a fish tank, and the fish. So, let's create these two classes.

Right-click on the **AlphabetAquarium** project item in the Solution Explorer, select **Add**, and then select **Class...** In the Add New Item dialog box, change the Name from Class1.cs to **Fish.cs**, then click **Add**.





In the Code Editor for Fish.cs, modify the code as shown.

Note

Note that we have not made the Fish class public as we have in past lessons. By default, when you create a new class, Studio omits the public access modifier. When a class is declared without an access modifier, the access level is internal. Internal access level indicates that the class can only be accessed from within the same assembly, but not from other assemblies. Public access would allow access to the class from other assemblies. Unless you are creating code that may be called from other assemblies, omitting the class access modifier is acceptable. If you decide later that you need to access the class from another assembly, you can always add back the public access modifier. You might also note that many developers and development groups prefer that you always indicate the access modifier to prevent confusion. For additional information, refer to the following web pages: [Access Modifiers](#) and [Assemblies and the Global Assembly Cache](#).

Fish.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;

namespace AlphabetAquarium
{
    class Fish
    {
        private Color _fishColor;
        public Color FishColor
        {
            get { return _fishColor; }
            set { _fishColor = value; }
        }

        private int _xPosition;
        public int XPosition
        {
            get { return _xPosition; }
            set { _xPosition = value; }
        }

        private int _yPosition;
        public int YPosition
        {
            get { return _yPosition; }
            set { _yPosition = value; }
        }

        private string _fishLetter;
        public string FishLetter
        {
            get { return _fishLetter; }
            set { _fishLetter = value; }
        }

        public Fish(string fishLetter, int xPosition, int yPosition, Color fishC
olor)
        {
            // If no letter specified, use "X."
            if (fishLetter.Length == 0)
                fishLetter = "X";
            _fishLetter = fishLetter;

            // Ensure the position is >= 0.
            if (xPosition < 0)
                xPosition = 0;
            _xPosition = xPosition;

            if (yPosition < 0)
                yPosition = 0;
            _yPosition = yPosition;

            // Set the fish color.
            _fishColor = fishColor;
        }
    }
}
```

Click  to save your changes. Let's discuss this code.

OBSERVE: Fish.cs

```

.
.
.
using System.Drawing;

namespace AlphabetAquarium
{
    class Fish
    {
        private Color _fishColor;
        public Color FishColor
        {
            get { return _fishColor; }
            set { _fishColor = value; }
        }

        private int _xPosition;
        public int XPosition
        {
            get { return _xPosition; }
            set { _xPosition = value; }
        }

        private int _yPosition;
        public int YPosition
        {
            get { return _yPosition; }
            set { _yPosition = value; }
        }

        private string _fishLetter;
        public string FishLetter
        {
            get { return _fishLetter; }
            set { _fishLetter = value; }
        }

        public Fish(string fishLetter, int xPosition, int yPosition, Color fishC
olor)
        {
            // If no letter specified, use "X."
            if (fishLetter.Length == 0)
                fishLetter = "X";
            _fishLetter = fishLetter;

            // Ensure the position is >= 0.
            if (xPosition < 0)
                xPosition = 0;
            _xPosition = xPosition;

            if (yPosition < 0)
                yPosition = 0;
            _yPosition = yPosition;

            // Set the fish color.
            _fishColor = fishColor;
        }
    }
}

```

The **Fish** class embodies the **properties** and methods (just the **Fish()** constructor for now) we need to create and track a fish in our virtual aquarium. Essentially, we include the color of the fish (**_fishColor**), the letter representing the fish (**_fishLetter**), and the position of the fish (**_xPosition**, **_yPosition**). Each of these properties, as class variables, is accessed using an **accessor**, and implemented using the appropriate **get** and **set** statements. The **Fish()** constructor includes parameters to initialize the properties of the fish, code

that tests the values of the **parameters** and sets them to default values if necessary, and the code to set the class variables from the **Fish() parameters**. You'll also note that we used a using statement to include the System.Drawing namespace for access to the Color type.

Let's now add a class to represent our fish tank.

Right-click on the **AlphabetAquarium** project item in the Solution Explorer, select **Add**, and then select **Class...** In the Add New Item dialog box, change the Name from Class1.cs to **FishTank.cs**, then click **Add**. In the FishTank.cs Code Editor, modify the code as shown:

```
FishTank.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AlphabetAquarium
{
    class FishTank
    {
        // Use a List collection to hold the fish.
        private List<Fish> _fishTank = new List<Fish>();

        public int CountFish()
        {
            return _fishTank.Count;
        }

        public Fish GetFish(int position)
        {
            return _fishTank[position];
        }

        public void AddFish(Fish fish)
        {
            _fishTank.Add(fish);
        }

        public void ClearFish()
        {
            _fishTank.Clear();
        }
    }
}
```

Click  to save your changes. We can't run it yet, but let's discuss this code.

OBSERVE: FishTank.cs

```
.
.
.
namespace AlphabetAquarium
{
    class FishTank
    {
        // Use a List collection to hold the fish.
        private List<Fish> _fishTank = new List<Fish>();

        public int CountFish()
        {
            return _fishTank.Count;
        }

        public Fish GetFish(int position)
        {
            return _fishTank[position];
        }

        public void AddFish(Fish fish)
        {
            _fishTank.Add(fish);
        }

        public void ClearFish()
        {
            _fishTank.Clear();
        }
    }
}
```

This code uses a C# collection **List** object, and the correct C# syntax appropriate to a collection (we'll cover these collection objects more completely later). We create a **List** object variable, **_fishTank**, using the **new** operator, that will hold each instance of the Fish class we created previously, effectively representing our aquarium. We use **List** methods to add a fish to our tank, remove all fish from our tank, return the number of fish in our tank, and to return a reference to specific fish instance.

Finally, we need to add the code to use the Fish and FishTank classes, enable all of the event handlers for our controls, and add methods and code to initialize and draw our fish.

Let's start by adding the event handlers for our ComboBox and Button controls. However, in the past, we typically double-clicked a control to generate the code for an event handler. This method works great when the event you want handled is the *default* event that's generated when you double-click a control, but sometimes that isn't the event you want handled. So, for the events we need for this application, we'll use the

Events lightning bolt icon  of the Properties Window.

Tip

Remember that the Properties Window, which includes the Events icon, is sensitive to what you have selected, so make sure you select the correct control or Form in creating event handlers. You can create an event handler by double-clicking on the event. The name of the method to handle the event will automatically populate the ComboBox next to the event, and you will be taken to the Code Editor, with the cursor positioned within the new event handler. If you accidentally click the wrong event, or decide later you do not want that event handler, you cannot simply delete the event code in the Code Editor. You must return to the Events Property page, and delete the event handler method name in the ComboBox next to the event. If you have not modified the event handler code in the Code Editor, then deleting the event handler method name in the Events Property page will also delete the event code from the Code Editor.

Select the **AlphabetAquarium.cs [Design]** Form Editor, and for each control or Form listed in the table below, select the **Events** icon in the **Properties Window** and create the appropriate Event handler by double-clicking the event. You will see the generated code in the Code Editor after each event handler is generated, so you'll need to return to the **AlphabetAquarium.cs [Design]** Form Editor to select the next

control. When you finish, click  to save your changes.

Control/Form	Event	Event Handler
AlphabetAquarium Form	Load	AlphabetAquarium_Load
colorsComboBox	SelectedIndexChanged	colorsComboBox_SelectedIndexChanged
addFishButton	Click	addFishButton_Click
clearFishButton	Click	clearFishButton_Click
exitButton	Click	exitButton_Click
fishTankPanel (the left panel)	Paint	fishTankPanel_Paint

Tip

If you need to select a control but can't find it, you can always select the control from the ComboBox at the top of the Properties Window.

The Code Editor should now have all of the event handlers that we'll need for this project, but before we add the code, let's discuss how the program is going to work.

Control/Form	Event	Discussion
AlphabetAquarium Form	Load	Populate the colorsComboBox with all available "named" colors, and select Black as the default. Populate the fishComboBox with all of the letters of the alphabet, and select the first entry in the ComboBox.
colorsComboBox	SelectedIndexChanged	Change the color of the colorPictureBox to match the selected color.
addFishButton	Click	Add the selected "fish" letter using the selected color to a fish tank Collection, generating a random location on the fishTankPanel. Generate an event to make the fishTankPanel update the display of our fish from the fish tank Collection.
clearFishButton	Click	Clear the fish tank Collection. Generate an event to make the fishTankPanel update the display of our fish from the fish tank Collection.
exitButton	Click	Close the application.
fishTankPanel	Paint	For each fish stored in our fish tank Collection, get the letter, color, and location, and draw the "fish" on the fishTankPanel.

As you can see, we've tried to apply our "model" concept as we created our classes to represent our aquarium, and as we created events to handle the interaction with our fish and our fish tank. Using concepts that model the real world as closely as possible makes understanding our code easier. So, let's add code to populate our ComboBox controls, display the selected color for each "fish" we add, set a class variable to store our added fish, and exit the application.

Modify the **AquariumAlphabet.cs** code as shown below.

AquariumAlphabet.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace AlphabetAquarium
{
    public partial class AlphabetAquarium : Form
    {
        private FishTank _fishTank = new FishTank();

        public AlphabetAquarium()
        {
            InitializeComponent();

            private void AlphabetAquarium_Load(object sender, EventArgs e)
            {
                // Populate "colors" ComboBox, select "Black" as default.
                colorsComboBox.Items.Add("Black");
                colorsComboBox.Items.Add("Red");
                colorsComboBox.Items.Add("Green");
                colorsComboBox.Items.Add("Blue");
                colorsComboBox.SelectedIndex = colorsComboBox.FindString("Black");
                // Populate the "fish" ComboBox, select "A" as default
                fishComboBox.Items.Add("A");
                fishComboBox.Items.Add("B");
                fishComboBox.Items.Add("C");
                fishComboBox.Items.Add("D");
                fishComboBox.SelectedIndex = fishComboBox.FindString("A");
            }

            private void colorsComboBox_SelectedIndexChanged(object sender, EventArgs e)
            {
                // Change the color of the colorPictureBox to match the selected color.
                colorPictureBox.BackColor = Color.FromName(colorsComboBox.SelectedItem.ToString());
            }

            private void addFishButton_Click(object sender, EventArgs e)
            {
            }

            private void fishTankPanel_Paint(object sender, PaintEventArgs e)
            {
            }

            private void clearFishButton_Click(object sender, EventArgs e)
            {
                _fishTank.ClearFish();
                fishTankPanel.Invalidate();
            }

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



, and  to run the program. Select a different color and note that the color picture box changes. Any new fish added will have this color.

Let's discuss how it works.

OBSERVE: AquariumAlphabet.cs

```
.
.
.

namespace AlphabetAquarium
{
    public partial class AlphabetAquarium : Form
    {
        private FishTank _fishTank = new FishTank();

        public AlphabetAquarium()
        {
            InitializeComponent();

            private void AlphabetAquarium_Load(object sender, EventArgs e)
            {
                // Populate "colors" ComboBox, select "Black" as default.
                colorsComboBox.Items.Add("Black");
                colorsComboBox.Items.Add("Red");
                colorsComboBox.Items.Add("Green");
                colorsComboBox.Items.Add("Blue");
                colorsComboBox.SelectedIndex = colorsComboBox.FindString("Black");
                // Populate the "fish" ComboBox, select "A" as default
                fishComboBox.Items.Add("A");
                fishComboBox.Items.Add("B");
                fishComboBox.Items.Add("C");
                fishComboBox.Items.Add("D");
                fishComboBox.SelectedIndex = fishComboBox.FindString("A");
            }

            private void colorsComboBox_SelectedIndexChanged(object sender, EventArgs e)
            {
                // Change the color of the colorPictureBox to match the selected color.
                colorPictureBox.BackColor = Color.FromName(colorsComboBox.SelectedItem.ToString());
            }
        }
    }
}
```

With this code, we use the ComboBox **Add()** method of the **Items** collection that holds the ComboBox data to populate both the **colorsComboBox** and **fishComboBox**. We then set the **SelectedIndex** property of the ComboBox by locating an appropriate default value for each ComboBox using the **FindString** method. As a color is selected in the **colorsComboBox**, the **colorsComboBox_SelectedIndexChanged** event is called, or "raised," and we use the **Color.FromName()** method to select the Color based on the name of the color, setting the **BackColor** of the **colorPictureBox** control.

We've also added a **_fishTank** class variable of type **FishTank** that will hold our "fish." The **_fishTank** instance class variable of the **FishTank** class gives us access to all of the methods of this class, and represents our aquarium in our model.

OBSERVE: Clearing the Fish

```
.  
.br/>.br/>private void clearFishButton_Click(object sender, EventArgs e)  
{  
    _fishTank.ClearFish();  
    fishTankPanel.Invalidate();  
}  
  
.br/>.br/>.
```

When the clearFishButton is clicked, the **clearFishButton_Click** event is raised, and we call the **ClearFish()** method. We then call the **Invalidate()** method of the **fishTankPanel** control. The Invalidate() method forces the fishTankPanel to redraw itself. When a control is drawn, the Paint event is called (**fishTankPanel_Paint** in our code), so let's proceed to implement the remaining code to complete our aquarium.

Modify the **AquariumAlphabet.cs** code as shown below.

AquariumAlphabet.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace AlphabetAquarium
{
    public partial class AlphabetAquarium : Form
    {
        private FishTank _fishTank = new FishTank();

        public AlphabetAquarium()
        {
            InitializeComponent();
        }

        private void AlphabetAquarium_Load(object sender, EventArgs e)
        {
            // Populate "colors" ComboBox, select "Black" as default.
            colorsComboBox.Items.Add("Black");
            colorsComboBox.Items.Add("Red");
            colorsComboBox.Items.Add("Green");
            colorsComboBox.Items.Add("Blue");
            colorsComboBox.SelectedIndex = colorsComboBox.FindString("Black");
            // Populate the "fish" ComboBox, select "A" as default.
            fishComboBox.Items.Add("A");
            fishComboBox.Items.Add("B");
            fishComboBox.Items.Add("C");
            fishComboBox.Items.Add("D");
            fishComboBox.SelectedIndex = fishComboBox.FindString("A");
        }

        private void colorsComboBox_SelectedIndexChanged(object sender, EventArgs e)
        {
            // Change the color of the colorPictureBox to match the selected color.
            colorPictureBox.BackColor = Color.FromName(colorsComboBox.SelectedItem.ToString());
        }

        private void addFishButton_Click(object sender, EventArgs e)
        {
            // Use the boundaries of the fishTankPanel to limit our random x, y location.
            Rectangle fishTankRect = fishTankPanel.Bounds;
            Random random = new Random();
            int x = random.Next(10, fishTankRect.Width - 10);
            int y = random.Next(10, fishTankRect.Height - 10);
            // Create a new Fish object, and add to our fish tank.
            Fish fish = new Fish(fishComboBox.SelectedItem.ToString(), x, y, Color.FromName(colorsComboBox.SelectedItem.ToString()));
            _fishTank.AddFish(fish);
            fishTankPanel.Invalidate();
        }

        private void fishTankPanel_Paint(object sender, PaintEventArgs e)
        {
            // Loop through each fish in our fish tank, and draw them.
            for (int i = 0; i < _fishTank.CountFish(); i++)
            {
                Fish fish = _fishTank.GetFish(i);
            }
        }
    }
}
```

```

        e.Graphics.DrawString(fish.FishLetter, new Font("Arial", 10),
            new SolidBrush(fish.FishColor), new Point(fish.XPosition, fi
sh.YPosition));
    }
    fishCountLabel.Text = _fishTank.CountFish().ToString();
}

private void clearFishButton_Click(object sender, EventArgs e)
{
    _fishTank.ClearFish();
    fishTankPanel.Invalidate();
}

private void exitButton_Click(object sender, EventArgs e)
{
    this.Close();
}
}
}

```

 and  to run the program. Now you can add fish of different colors and have them appear in random locations in the fish tank.

Let's discuss how this code works.

OBSERVE: AquariumAlphabet.cs

```

.
.
.

private void addFishButton_Click(object sender, EventArgs e)
{
    // Use the boundaries of the fishTankPanel to limit our random x, y
location.
    Rectangle fishTankRect = fishTankPanel.Bounds;
    Random random = new Random();
    int x = random.Next(10, fishTankRect.Width - 10);
    int y = random.Next(10, fishTankRect.Height - 10);
    // Create a new Fish object, and add to our fish tank.
    Fish fish = new Fish(fishComboBox.SelectedItem.ToString(), x, y, Col
or.FromName(colorsComboBox.SelectedItem.ToString()));
    _fishTank.AddFish(fish);
    fishTankPanel.Invalidate();
}

private void fishTankPanel_Paint(object sender, PaintEventArgs e)
{
    // Loop through each fish in our fish tank, and draw them.
    for (int i = 0; i < _fishTank.CountFish(); i++)
    {
        Fish fish = _fishTank.GetFish(i);
        e.Graphics.DrawString(fish.FishLetter, new Font("Arial", 10),
            new SolidBrush(fish.FishColor), new Point(fish.XPosition, fi
sh.YPosition));
    }
    fishCountLabel.Text = _fishTank.CountFish().ToString();
}

.
.
.

```

This code completes everything we need to fully implement our virtual alphabet "fish" aquarium! We added code to the **addFishButton_Click** event handler to generate a random location using the **Random** class. To make sure our random location falls within the limits of the **fishTankPanel**, we call the **Bounds** method of the **fishTankPanel** to return a **Rectangle** object, setting the **fishTankRect** variable. We use this variable later when we generate the actual random number for our location. We next create a **Fish** instance, **fish**,

passing in the necessary parameters to the **Fish** constructor, which includes our fish letter, the location of the fish, and the Color. We add the new **fish** to our **_fishTank**, and then force the redraw, or repaint of the **fishTankPanel** by calling the **Invalidate()** method.

The **fishTankPanel_Paint** method is the event handler that's raised whenever the **fishTankPanel** control needs to be redrawn, which we force when we call **Invalidate()** method. We've added a **for** loop to the **fishTankPanel_Paint** method, looping through the **_fishTank** object that contains all of our "fish." We limit the **for** loop using the **_fishTank.CountFish()** method, extracting each **fish** by calling the **_fishTank.GetFish** method, passing the loop variable **i**. We use the **DrawString** method of the **Graphics** object to draw our fish letter, passing in the necessary parameters. We use the **FishLetter**, **FishColor**, **XPosition** and **YPosition** methods of the **fish** object to extract the fish information. We also create a **Font**, **SolidBrush**, and **Point** object "on-the-fly" (without setting a variable) to complete the parameters needed by **DrawString**. Then, finally, we call the **CountFish()** method of the **_fishTank** to set the fish count in our GUI.

Classes: A New Data Type

When you create a class, you're creating a new data type. You can see this fact with the declaration of our **_fishTank** class variable: **private FishTank _fishTank**. Why is this fact important, besides the obvious about how you use a class? Earlier, we discussed the concept of our programming code representing, or modeling, the real world, and as you continue to learn more C#, and just as importantly, object-oriented programming concepts, you'll need to remember the flexibility and versatility of being able to create your own data types to find solutions.

Objects: A Multiplicity of Instances

Again related to our modeling the real world, even though we have a class blueprint, we also take advantage of the ability to create an instance of a class, creating a class instance object. For our aquarium, each time a "fish" was added, we created an instance of our **Fish** class, filling our **FishTank** object with multiple instances of **Fish**, just like you'd have in the real world.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Defining Classes and Class Relationships

In this lesson, we'll discuss a conceptual model versus the actual design elements required by using an OOP language like C# to implement the final product. We'll use Unified Modeling Language (UML) tools built into Visual Studio to assist with the conceptualization and implementation of the model, designing our C# classes during the modeling process. We'll further discuss the concept of scope and visibility in the context of our model and classes, the relationship between our classes, and the operations between those classes as objects.

Conceptual Versus Design

In creating a C# application, we've considered a conceptual model of the components of the application as they might exist as objects in the real world. As we move from our conceptual model to designing the application using Visual Studio and C#, often the real-world model and the design implementation diverge. Why? There may be a number of reasons for the differences, such as:

- **Functionality:** A full model is not needed to represent the objects when actually coded.
- **Feature deficit:** The OOP features of the language do not provide the ability to create a one to one representation of the model in code.
- **Performance:** A modified version of the model is more efficiently implemented using certain features of the OOP language.
- **Other Constraints:** Another reason exists to adapt the model during implementation.

Regardless, we will always try to have our model initially represent real-world objects as closely as possible. Modifications that deviate from this initial model should be documented as you work through the design implementation. So, let's create a new application that uses letters as stamps to create a stamp collection. Before we begin coding, let's see if we can design a "handwritten" model by determining the objects we need and how they relate to each other.

Defining a Class

What objects would compose a real-world stamp album? Primarily, there would be two objects: stamps and albums. In the real world, we might collect more than one stamp of the same kind, and we may need more than one album, as an album has a finite number of stamps it can contain. If we wanted to create an application that exactly duplicated what we might contain in the real world, then we would have to support the concept of a single stamp, with different properties, and multiple albums, each holding an instance of a stamp. Such an application would effectively serve as a catalog of the real stamps and albums we owned. But, we could also create a virtual stamp collection. Why virtual? Well, a virtual album has unlimited space, so we'd only need a single album. Also, our stamp collection may not be intended to represent every duplicate stamp, but only each unique stamp. Can you see how quickly our design can deviate from the objects in the real world? The final design depends on what we want the application to do, and how we might adapt the application to take advantage of a virtual representation versus a real-world application.

For our stamp collection, let's model the real world, making our application a catalog of our entire stamp collection. That means that rather than have, say, an "A" stamp, and a count of how many "A" stamps we have and treating them as if they were all identical, we look at each "A" stamp as an individual stamp, with properties that relate to each stamp, like the date it was released, when we acquired it, and so on. In the real world, a single album only holds so many stamps, so we'll have to include an album identifier, or number, and a way to represent a collection of stamps for each album, and a way to represent a collection of albums.

So, properties of our stamp class are:

- Letter (a different letter represents a different stamp)
- Stamp release date
- Stamp acquired date
- Includes postmark

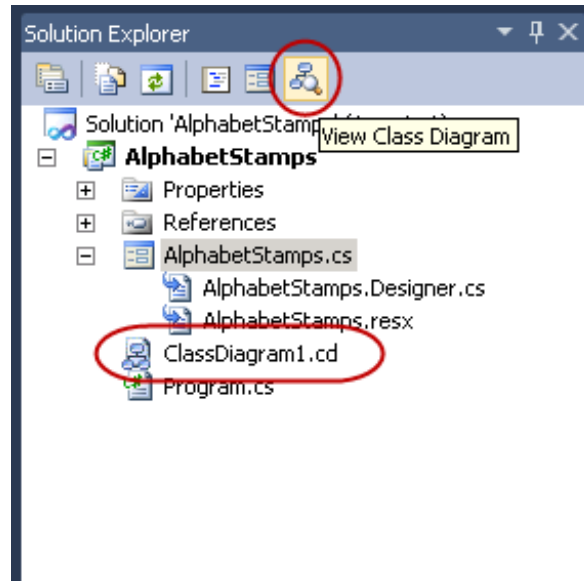
Properties of the album class are:

- Album number
- Stamps
- Stamp count

With these two class definitions, let's use Visual Studio to create a class diagram of our two classes using the Visual Class Diagram tool. This tool helps us to visually create the classes, adding class variables (fields), properties, accessors (get and set methods), methods, and constructors, adding the appropriate code to our classes. Not only does the tool create a UML class diagram, it helps us to visualize the relationships between classes that we'll explore in the next section.

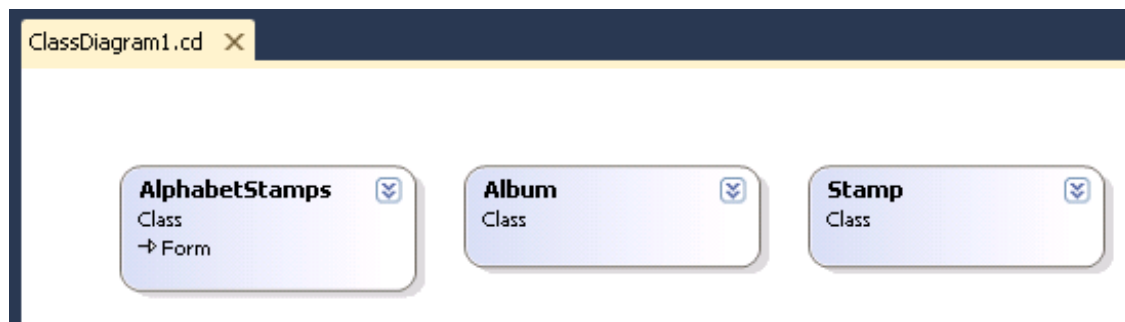
Create a new Visual Studio Windows Forms Application named **AlphabetStamps**. Click the entry for **Form1.cs** and replace it with **AlphabetStamps.cs**. Click the Form1 Form titlebar, and change its Text property from Form1 to **Alphabet Stamps**.

Click the View Class Diagram icon  in the Solution Explorer.

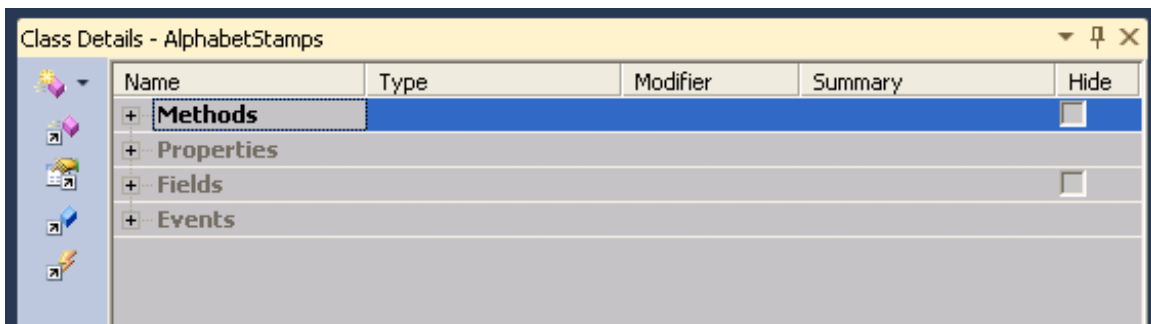


Click  to save your changes. You'll notice that the class diagram for the project already contains an item, known as a class icon. The class icon containing **AlphabetStamps** is the main class we work with when we create a Windows Form application, and is the class named Form1.cs by default, which we changed to AlphabetStamps.cs. Now, we'll add some classes of our own.

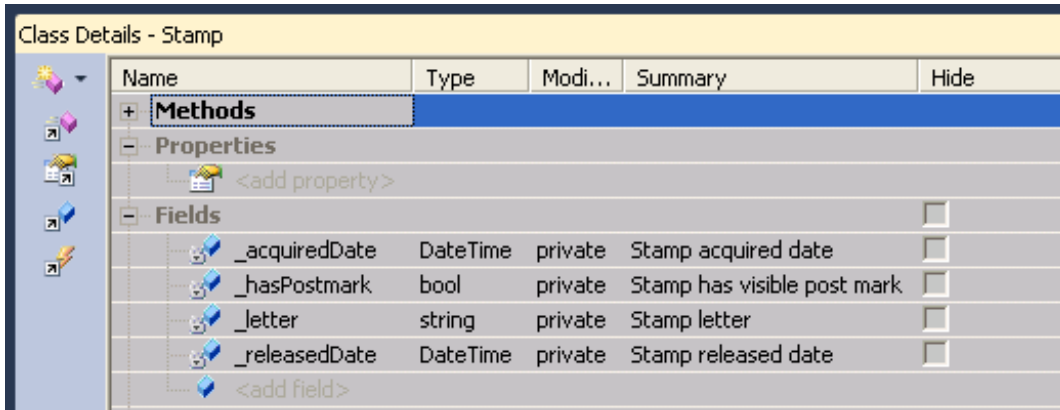
Right-click in the white space of the Visual Class Designer, and in the popup menu that appears, select **Add | Class**. In the New Class dialog box that appears, change the Class name: value to **Stamp**, then click **OK**. Add a second class named **Album**.



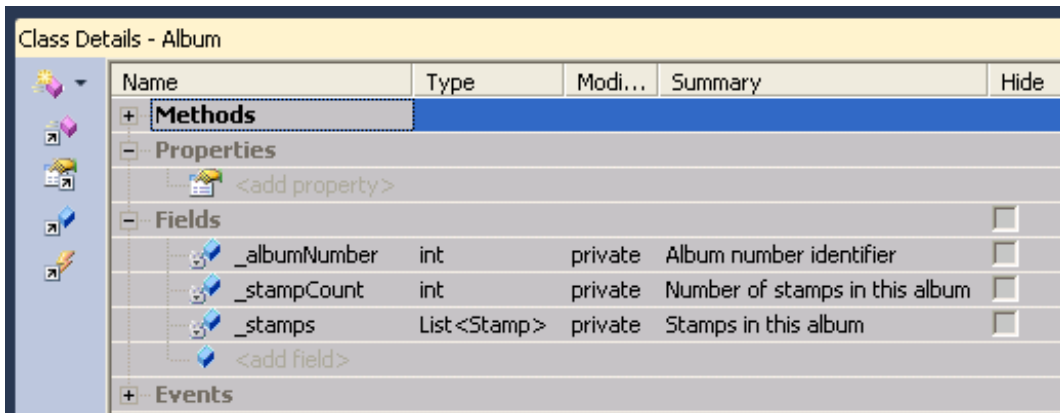
Let's add our class properties to both of the classes using the Class Details. The Class Details window includes both *Properties* and *Fields*. What's the difference? Fields are simply *private* class variables, whereas Properties include accessors (get and/or set methods). We could add all of these properties in the **Properties** section, but the default C# code that is generated isn't as optimal as if we add them as **Fields** and then Refactor them.



Select the **Stamp** class in the Visual Class Designer, then add each field using the Class Details as shown below.



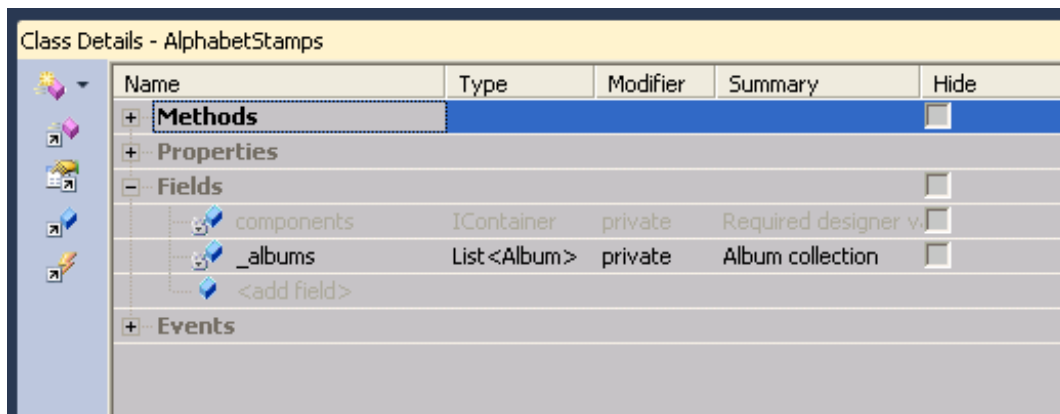
Click  to save the modified class diagram and Stamp class. Then, select the **Album** class in the Visual Class Designer, and add the fields as shown below.



Click  to save the modified class diagram and Album class. You probably recognize the **List<Stamp>** syntax as a .NET *Collection* like we've used before. We'll use this collection to hold each **Stamp** object that a specific album contains.

We now have a stamp class we can use to create an instance of each stamp object. We also have an album class we can use to hold each instance of a stamp object for that album. What other objects might we need? In the real world, we would have our albums stacked nicely on a shelf, kept in a safe, or hidden somewhere safe from our siblings who can't wait to draw a funny mustache on a very expensive stamp. In our virtual stamp collection, we need another collection to represent the concept of storing our albums. We could create another class, perhaps called **Albums**, but for our current implementation, all we really need is another **List** collection. We'll add this collection to our AlphabetStamps class.

Select the **AlphabetStamps** class in the Visual Class Designer, then add a new private field named **_albums** of type **List<Album>**, as shown below.



Click  to save the modified class diagram and AlphabetStamps class. Using tools like the Visual Class Designer helps you to visualize your classes and objects as you attempt to model the real world. (Of course, you have the freedom to design your model however you like, even if it doesn't represent the real world, although approximating the real world helps you and others to more easily understand your code and the classes and objects you design.) The Visual Class Designer is creating your underlying code, so let's take a look at the two new classes, and the changes we made to the AlphabetStamps Form class.

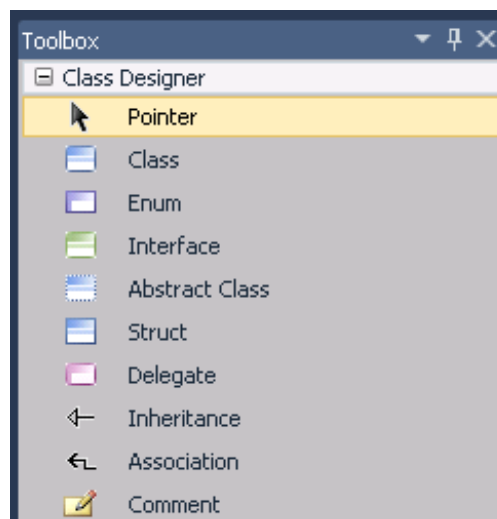
Double-click the **Stamp.cs** entry in the Solution Explorer, and examine the generated code in the Visual Code Editor. Repeat this process for the **Album.cs** entry, and for the **AlphabetStamp.cs** entry (for this entry, you'll need to right-click it and select **View Code**, as it also has a design component, the Form).

As you open and examine the generated code, you'll notice the three-slash (///) character blocks. These sections are part of Studio's ability to generate documentation for your source code. We'll discuss this later in more detail.

Before we add the "sets" and "puts" (accessors) using refactoring, let's discuss class relationships.

Defining Relationships

When you click on the Visual Class Designer (the **ClassDiagram1.cd** tab), the Studio Toolbox displays tools you can use when visually creating classes:



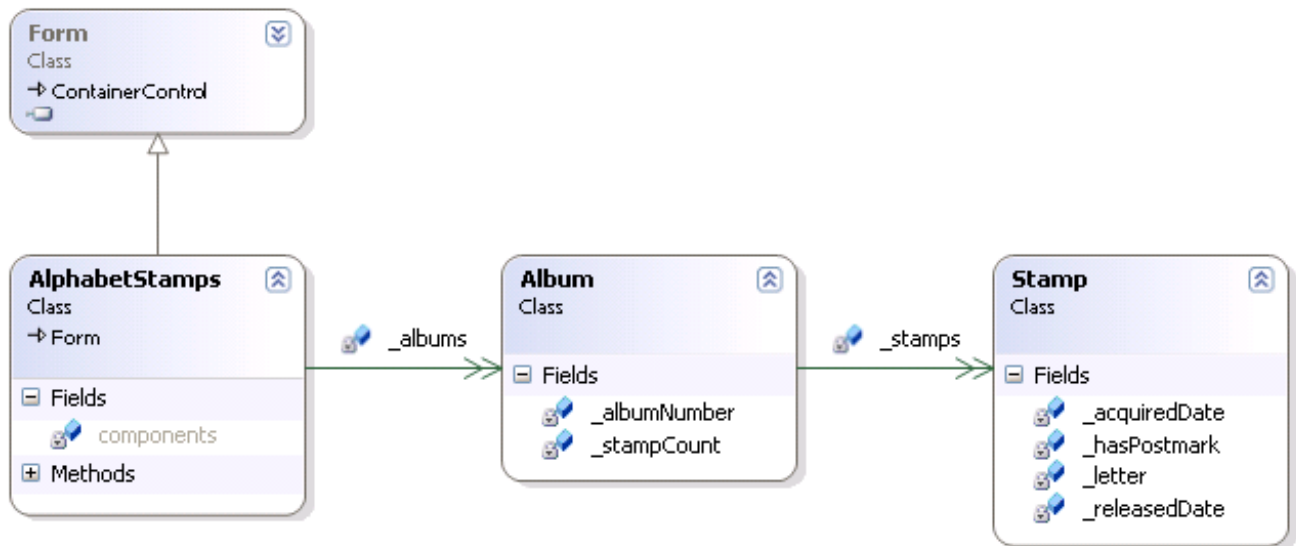
As with other Toolbox tools, you can drag and drop certain tools onto the Designer, but a couple of these tools, when selected, allow you to create *relationships* between classes and class fields. What do we mean by relationships? Consider objects in the real world, and how we define them. For example, all trucks are vehicles, but not all vehicles are trucks. Some might be cars, motorcycles, or something else entirely. When we think of objects, often it's helpful to think in terms of one object being of a type of another object, an *is-a* relationship: a truck is-a vehicle. Another relationship is often described as a *has-a* relationship: a car has-a steering wheel. In our current project, we have one is-a relationship: our AlphabetStamps class is-a Form. We also have two has-a relationships: our Album class has stamps, and our Albums collection has individual albums.

We use different kinds of visual cues to show these relationships in the Designer. An is-a relationship (also called inheritance), is indicated by a line with a closed arrow next to the parent class. A has-a relationship (also called association), is indicated by a line with an open arrow. Associations indicate that the datatype of a field is of a class

we've defined. Associations that involve collections are called collection associations, and use two open arrows. Short descriptive information is also typically displayed next to the association line to help describe the relationship.

Our Class Diagram does not currently contain any visual associations, so let's display the associations that we've already created. Although the Class Designer visual display will change to display these associations, no underlying code changes will occur. In future lessons, we'll further explore using the Visual Class Designer to create these associations using Toolbox Class Designer items.

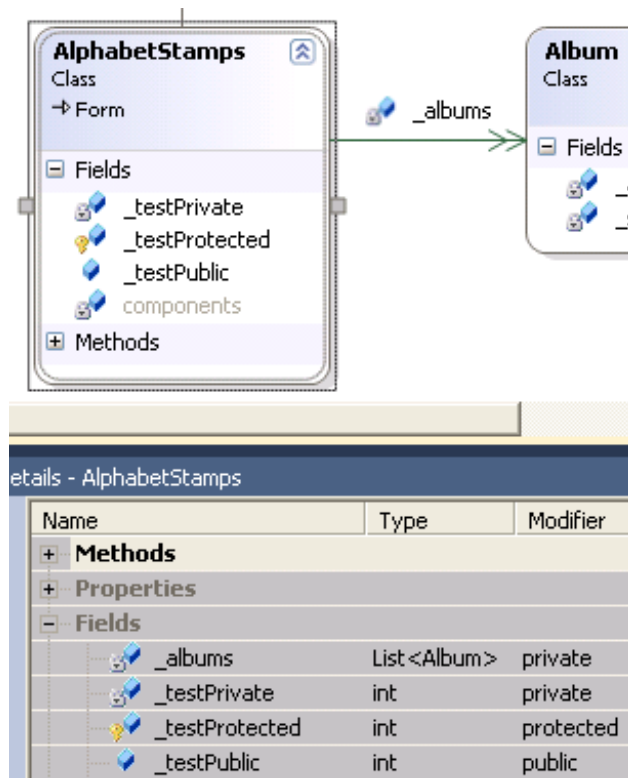
Right-click on the **AlphabetStamps** class and select **Show Base Class**. Next, right-click on the **_albums** field and select **Show As Collection Association**. Then, right-click on the **_stamps** field **Album** class and select **Show As Collection Association**. Your design view should look like this:



As you can see, adding the inheritance and association lines enhances the visual representation of the class diagram. Let's discuss a few other issues related to class diagrams.

Visibility and Cardinality

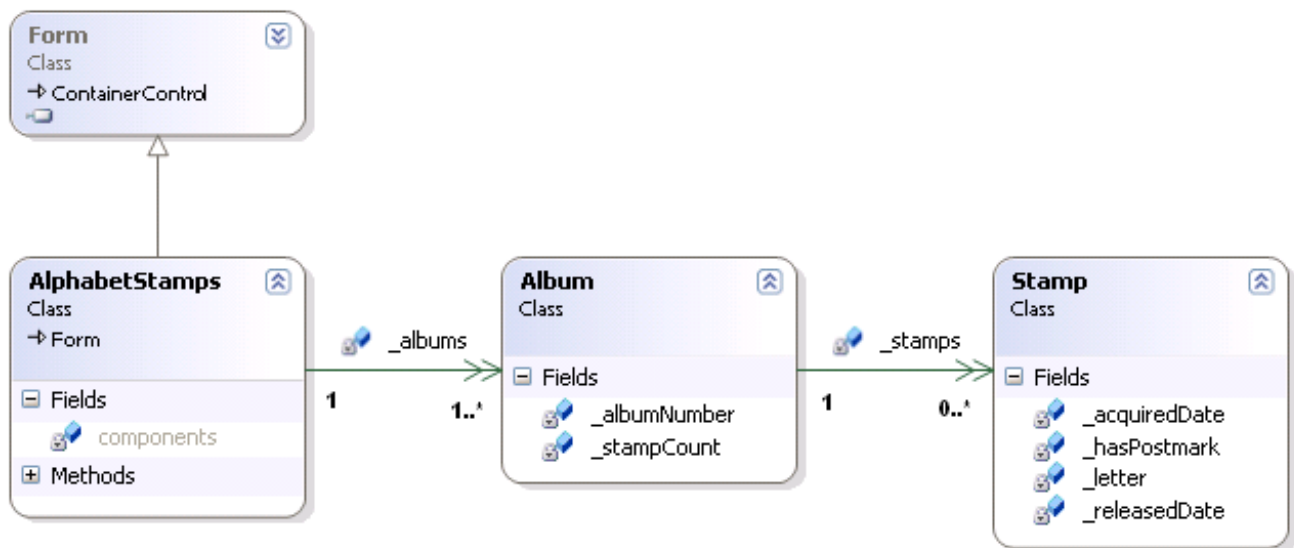
An important feature of a UML class diagram is the ability to easily determine and recognize the *visibility* of elements of a class, such as fields and methods, by using access modifiers. Typically, in a standard UML class diagram, public access is visually illustrated using a plus symbol (+), private access using a minus symbol (-), and protected access using the pound or hash mark (#). In the Visual Class Designer, visibility is portrayed by the symbols to the left of each element of the class. For protected, a small gold key would be shown. For private, a small padlock, and for public, there would be no extra symbol. Below is an image of three class variables we've added to the **AlphabetStamps** class for illustration purposes only to show each of these three scenarios, and the accompanying symbols. In the image, you'll notice we also included the Class Details that includes the access modifiers. Do not add these fields to your class.



Another very important concept that a UML class diagram should portray is *cardinality*, also known as *multiplicity*. This information indicates the relationship between classes, showing the number of instances of one class when linked to a single instance of another class. For example, in our model, one album will have multiple stamps (or none), and one stamp will belong to only a single album. The Visual Class Designer currently has no mechanism for depicting this relationship. The typical symbols used to depict cardinality in a UML class diagram are:

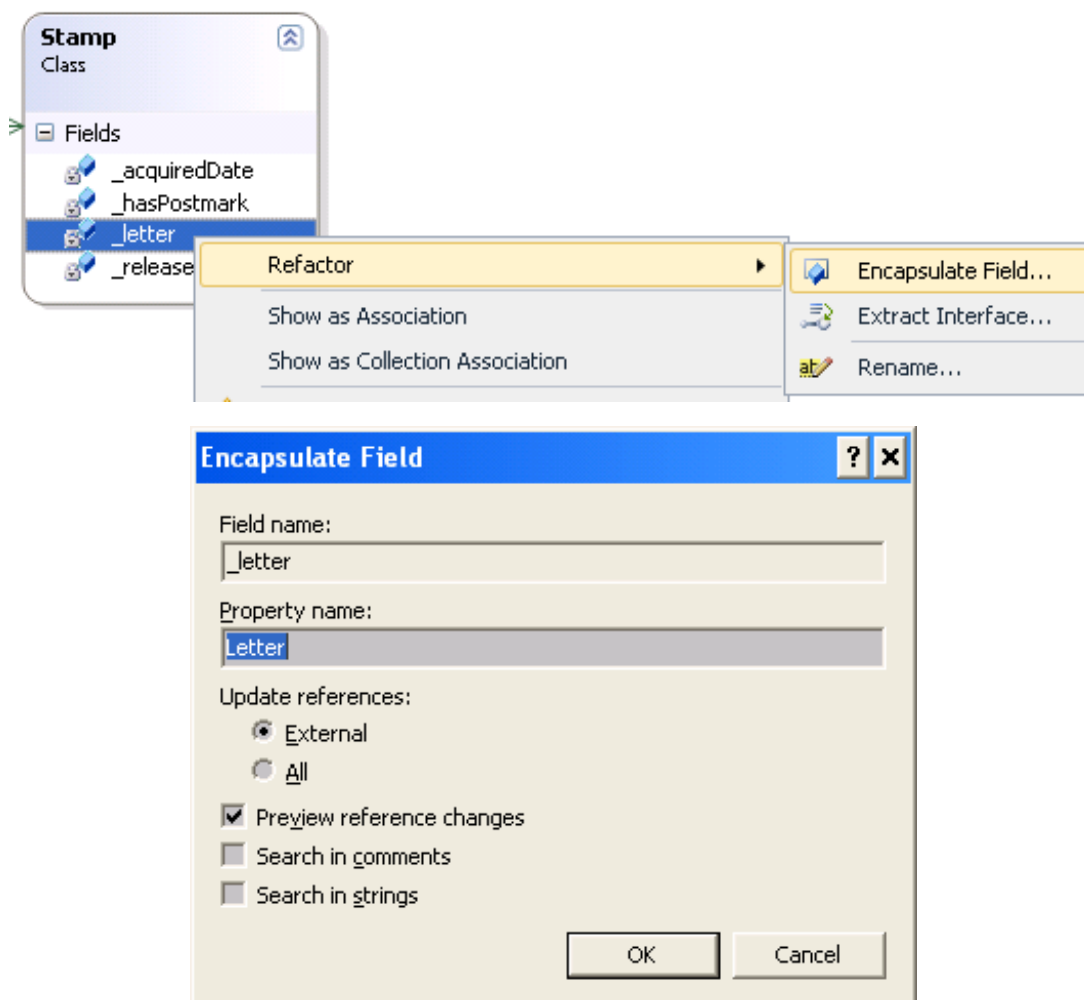
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only n (where n > 1)
0..n	Zero to n (where n > 1)
1..n	One to n (where n > 1)

So what would our class diagram look like if we could add cardinality? The image below shows the cardinality relationship between the stamp and album class, and the relationship between the albums collection and the album class. Note that an album does not require a stamp to be an album, expressed using the 0..* indicator, but that an albums collection requires at least one album, expressed using the 1..* indicator.



To go through all of the features UML can represent could easily require a course of its own, but for now, we've shown a few of the fundamentals provided by Visual Class Designer. We've also illustrated how the Visual Class Designer aids in producing your code. What about the refactoring we'd mentioned earlier? Let's refactor one class variable (or field), turning the field into a property by adding the accessors for the field.

Right-click on the **Stamp** class **_letter** field, and select **Refactor | Encapsulate Field**. In the Encapsulate Field dialog box, accept all of the defaults and click **OK**. In the next dialog that appears, click **Apply**. Next, double-click on the newly added **Letter** property to open the Visual Code Editor for the Stamp.cs class.



When you view the accessors in the Visual Code Editor, you should see the code highlighted below has been added to allow access to the **_letter** field. Using refactoring is a great way to convert fields into properties!

OBSERVE: Stamp.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AlphabetStamps
{
    public class Stamp
    {
        /// <summary>
        /// Stamp letter
        /// </summary>
        private string _letter;

        public string Letter
        {
            get { return _letter; }
            set { _letter = value; }
        }
        /// <summary>
        /// Stamp released date
        /// </summary>
        private DateTime _releasedDate;
        /// <summary>
        /// Stamp acquired date
        /// </summary>
        private DateTime _acquiredDate;
        /// <summary>
        /// Stamp has visible post mark
        /// </summary>
        private bool _hasPostmark;
    }
}
```

Congratulations! You've completed this lesson on defining classes and class relationships. You should have a better understanding of UML class diagrams and using the Visual Class Designer to create your classes!

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select the **Quiz(zes)** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Methods, Return Types, and Constructors

In this lesson, we'll explore methods. We've used methods extensively in previous lessons, so you'll be familiar with a lot of the material. We'll also discuss concepts around method parameters, and reintroduce class constructors.

Methods

Class Function Members

Methods are one member of a set of class functions that provide some means of manipulating class data. Below is a complete list of these class functions, with a short description for each. This lesson will cover many of these functions, while others will be covered later. In addition to class function members, we also have class data members that include fields (class variables using a public access modifier), and constants (unchangeable fields). Classes also include class event members for dealing with events.

- **Property functions** are specialized functions used to access private class variable data, often referred to as accessors, or gets and sets. The C# property syntax defines each uniquely, thus setting this class of functions apart from the method syntax we'll cover in this lesson, and that you're already familiar with. Properties provide a "method-like" interface for accessing class variables.
- **Methods**, the focus of this lesson, are class constructs used to further manipulate class data or provide some other type of interaction with the users of a class relevant to the purpose of the class. Methods can be instance methods, the type we'll consider in this lesson, or static methods, a type we'll consider in a future lesson. An instance method requires an instance of a class before the method can be invoked.
- **Constructors** are class functions using a specialized syntax called whenever a class is initialized. We'll also discuss constructors in this lesson.
- **Finalizers** are the opposite of constructors, called when an object is destroyed. As objects are created, and then no longer used by going out of scope, or when an application domain is shut down, finalizers are called to free up the resources used by objects before the objects themselves are destroyed. We'll discuss these in a later lesson.
- **Operator functions** allow for the overloading of other C# operators, such as the + operator, when used with an object. We'll discuss these in a later lesson.
- **Indexer functions** allow objects to be indexed similar to an array or a collection, and will also be covered in a future lesson.

Class Methods

In C#, methods must be defined within a class definition. Below is the syntax definition of a method. Square brackets indicate an optional element.

```
OBSERVE: Method Syntax Definition

[access modifiers] return_type methodName ( [parameters] )
{
    // Method body
}
```

From the syntax definition, you can see that a class method declaration consists of an optional **access modifier**, a **return type**, the **name of the method**, an optional list of **input parameters** in parentheses, and the **method body**, surrounded by {curly braces}. A value of **void** may be used for the **return type** to indicate the method does not return a value. Method definitions that do include a valid **return type** must use the **return** keyword to return a value. We've specified *camelCase* for the **method name**, although you may use any valid method naming convention you prefer.

We'll discuss each of the method syntax elements below.

Access Modifiers

An access modifier determines the restrictions placed on the calling of a method. As indicated, an access modifier is optional. If omitted, the default access is **internal** access. The available access modifiers are listed below. You may also reference Microsoft's online MSDN article [Visual Studio 2010 Access Modifiers](#).

- **public**: The type or member can be accessed by any other code in the same assembly or another

assembly that references it.

- **private**: The type or member can be accessed only by code in the same class or struct.
- **protected**: 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.
- **internal**: The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal**: 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.

Typically, best programming practices encourage the use of explicitly declaring the access modifier for a method. You'll note that most of the methods we've created in our lessons have been private. We'll examine the difference between private and internal in our sample program shortly.

Note In programming parlance, calling a method is also known as *invoking* a method.

Method Parameters

Method parameters are optional, but when used consist of one or more parameter definitions, separated by commas. Below is a method parameter syntax definition. Note the use again of square brackets to indicate optional elements.

Method Parameter Definition

```
( [parameter_modifier] data_type parameterName [default] )
```

Each parameter definition consists of an optional **parameter modifier**, the **data type** of the parameter, a valid **parameter name**, and an optional **default value**.

Parameter modifiers consist of the following options. You may also reference Microsoft's online MSDN article [Visual Studio 2010 Method Parameters](#).

- **ref**: Parameter values are passed by reference, requiring prior initialization of the argument. We will discuss this modifier later in this lesson.
- **out**: Parameter values are passed by reference without prior initialization of the method argument. We will discuss this modifier in a later lesson.
- **params**: Specifies a method parameter that takes a variable number of arguments. We will discuss this modifier in a later lesson.

If no parameter modifier is specified, a parameter is said to be *passed by value*. We'll discuss this concept more later in the lesson.

Note

What is the difference between a method parameter and a method argument? These two terms are often interchanged, but they have very distinct meanings. A method parameter is the name of the parameter used within the method definition and referenced within the body of the method. A method argument is the value that is said to be passed to the method when the method is invoked. So, a method parameter refers to the name of the parameter, whereas the method argument is the value of the parameter.

No two methods within the same class may have the same method name and parameter list; otherwise, the program wouldn't know which version of the method to call. This unique method-name-plus-parameter-type-list is also known as a method's *fingerprint* or *signature*. Note that the parameter list does *not* refer to the names of the parameters, but their *data types*.

Method Return Types

A method return type may either be **void** or a specific data type. If a specific data type is indicated, the method definition must use the **return** keyword to return a value of the specified data type.

Note

The void return type indicates that a method does not return a value; however, a method may still include the return keyword, indicating an exit point from the method. A method may have as many return statements as required by the program, or no *return* statements, allowing the program to exit at the conclusion of execution of all lines of code within the method. Multiple exit points are not considered a best practice, and should be avoided.

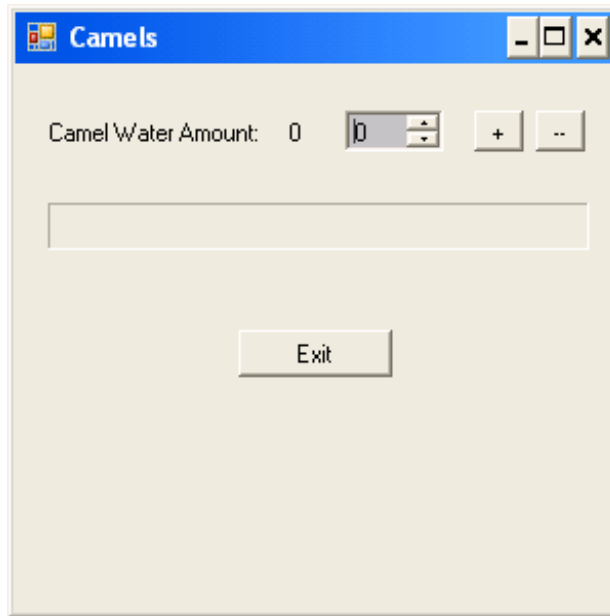
Coding: Camel I

Let's use the information we've learned so far to create a program that involves a fictitious camel, and more specifically, the hump of the camel. We're going to make it possible to add or remove water from the hump of our camel. It should be noted that camels don't actually store water in their hump, but a reservoir of fatty tissue that when metabolized can yield more than one gram of water per one gram of fat. Let's get started!

Create a new Visual Studio Windows Forms Application named **Camels**. Change the name **Form1.cs** to **Camels.cs**, and change the titlebar Text property to **Camels**. Click  to save your changes.

Now let's add a few controls. Note that we're using NumericUpDown  and ProgressBar  controls.

Modify the Camels Form by adding the controls listed below. Change each control's Name property to match the **Name Property** column, and each Text property to match the **Text Property** column. Arrange the controls as shown:



Object	Name Property	Text Property
Label	waterLabel	Camel Water Amount:
Label	waterAmountLabel	0
NumericUpDown	waterAmountNumericUpDown	
Button	addWaterButton	+
Button	removeWaterButton	--
ProgressBar	waterAmountProgressBar	
Button	exitButton	Exit

Click  to save your changes.

Next, let's add all of the event handlers we'll need, including one for the Camels Form Load event. You can add the events by double-clicking on the control, by selecting the correct events using the Events property page, or by a mixture of both methods.

Add the following events to the Camel Form. The double-clicking method will work for all of the listed events, as the methods are the default event handler for each control or Form.

Control	Event	Event Name
addWaterButton	Click	addWaterButton_Click
removeWaterButton	Click	removeWaterButton_Click
Camels Form	Load	Camels_Load
exitButton	Click	exitButton_Click

Next, we'll add a new class for handling our camel's hump, and the code to close the Form.

Select the **Camels.cs** Form Code Editor, and modify the code as shown below.

Camels.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Camels
{
    public partial class Camels : Form
    {
        private Hump _hump = new Hump();

        public Camels()
        {
            InitializeComponent();
        }

        private void addWaterButton_Click(object sender, EventArgs e)
        {
        }

        private void removeWaterButton_Click(object sender, EventArgs e)
        {
        }

        private void Camels_Load(object sender, EventArgs e)
        {
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }

    internal class Hump
    {
        private int _gallonsOfWater;
        private const int _maximumGallonsOfWater = 100;

        internal Hump()
        {
            _gallonsOfWater = 0;
        }

        internal int GallonsOfWater
        {
            get { return _gallonsOfWater; }
        }

        internal int MaximumGallonsOfWater
        {
            get { return _maximumGallonsOfWater; }
        }

        internal void addWater(int gallonsToAdd)
        {
            _gallonsOfWater += gallonsToAdd;
            if (_gallonsOfWater > _maximumGallonsOfWater)
                _gallonsOfWater = _maximumGallonsOfWater;
        }

        internal void removeWater(int gallonsToRemove)
        {
        }
    }
}
```

```

        {
            _gallonsOfWater -= gallonsToRemove;
            if (_gallonsOfWater < 0)
                _gallonsOfWater = 0;
        }
    }
}

```



to save your changes. You can run it, but it doesn't do much yet. Let's discuss the code we've added so far.

OBSERVE: Camels.cs

```

.
.
.
namespace Camels
{
    public partial class Camels : Form
    {
        private Hump _hump = new Hump();
        .
        .
        .
        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }

    internal class Hump
    {
        private int _gallonsOfWater;
        private const int _maximumGallonsOfWater = 100;

        internal Hump()
        {
            _gallonsOfWater = 0;
        }

        internal int GallonsOfWater
        {
            get { return _gallonsOfWater; }
        }

        internal int MaximumGallonsOfWater
        {
            get { return _maximumGallonsOfWater; }
        }

        internal void addWater(int gallonsToAdd)
        {
            _gallonsOfWater += gallonsToAdd;
            if (_gallonsOfWater > _maximumGallonsOfWater)
                _gallonsOfWater = _maximumGallonsOfWater;
        }

        internal void removeWater(int gallonsToRemove)
        {
            _gallonsOfWater -= gallonsToRemove;
            if (_gallonsOfWater < 0)
                _gallonsOfWater = 0;
        }
    }
}

```

Are you curious about the addition of a second **Hump** class in the same file? Yes, that's right, you can have more than one class within the same physical file. In a later lesson, we'll even discuss nesting classes within classes! But for now, we've added this class to illustrate a couple of points. The **Hump** class could just as easily have been defined in a separate own file. Notice that when you define a second class you *must* make that class, or any additional classes, have the **internal** access modifier. We've also changed all of the accessors and methods that would normally be public to also be **internal** to illustrate the use of this keyword.

The purpose of the **Hump** class is to provide a means of creating a **_hump** object we can use to add, remove, and track the amount of water stored in the camel's hump. The **_hump** object also includes code to make sure we don't attempt to remove more water than we have, or add more water than we can store. We use the read-only property **_gallonsOfWater** to store the current amount of water. Why is this property called a property, and how is it read-only? **_gallonsOfWater** is a property because it's a **private** class variable, and we've defined a **get** accessor. Because there is only a **get** accessor, and no **put** accessor, other classes can only **read** the value, and not assign a value, making the property read-only. You should also note that we've used the **return** keyword in our property accessors, just as we would have done if they'd been methods.

We've also defined a constant **_maximumGallonsOfWater**, using the **const** keyword. We use this constant to make sure we can't add more water than we can hold.

Finally, we added **the standard code** to make sure our application exits when the **exitButton** is clicked.

So far, our application doesn't allow us to add or remove water, so let's add that functionality, along with code to display the amount of water, and update our ProgressBar control to graphically display the water amount.

Select the **Camels.cs** Form Code Editor, and modify the code as shown below.

Camels.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Camels
{
    public partial class Camels : Form
    {
        private Hump _hump = new Hump();

        public Camels()
        {
            InitializeComponent();
        }

        private void addWaterButton_Click(object sender, EventArgs e)
        {
            _hump.addWater(Int32.Parse(waterAmountNumericUpDown.Value.ToString()
));
            updateWaterDisplay();
        }

        private void removeWaterButton_Click(object sender, EventArgs e)
        {
            _hump.removeWater(Int32.Parse(waterAmountNumericUpDown.Value.ToStrin
g()));
            updateWaterDisplay();
        }

        private void updateWaterDisplay()
        {
            waterAmountLabel.Text = _hump.GallonsOfWater.ToString();
            waterAmountProgressBar.Value = _hump.GallonsOfWater;
        }

        private void Camels_Load(object sender, EventArgs e)
        {
            waterAmountProgressBar.Minimum = 0;
            waterAmountProgressBar.Maximum = _hump.MaximumGallonsOfWater;
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }

    internal class Hump
    {
        private int _gallonsOfWater;
        private const int _maximumGallonsOfWater = 100;

        internal Hump()
        {
            _gallonsOfWater = 0;
        }

        internal int GallonsOfWater
        {
            get { return _gallonsOfWater; }
        }
    }
}
```

```
internal int MaximumGallonsOfWater
{
    get { return _maximumGallonsOfWater; }
}

internal void addWater(int gallonsToAdd)
{
    _gallonsOfWater += gallonsToAdd;
    if (_gallonsOfWater > _maximumGallonsOfWater)
        _gallonsOfWater = _maximumGallonsOfWater;
}

internal void removeWater(int gallonsToRemove)
{
    _gallonsOfWater -= gallonsToRemove;
    if (_gallonsOfWater < 0)
        _gallonsOfWater = 0;
}
}
```



and to run the program. Add water, enough to exceed 100 gallons, and then remove enough water to make it less than zero, to make sure it works. Let's discuss the latest changes.

OBSERVE: Camels.cs

```

.
.
.
namespace Camels
{
    public partial class Camels : Form
    {
        private Hump _hump = new Hump();

        public Camels()
        {
            InitializeComponent();
        }

        private void addWaterButton_Click(object sender, EventArgs e)
        {
            _hump.addWater(Int32.Parse(waterAmountNumericUpDown.Value.ToString()
));
            updateWaterDisplay();
        }

        private void removeWaterButton_Click(object sender, EventArgs e)
        {
            _hump.removeWater(Int32.Parse(waterAmountNumericUpDown.Value.ToStrin
g()));
            updateWaterDisplay();
        }

        private void updateWaterDisplay()
        {
            waterAmountLabel.Text = _hump.GallonsOfWater.ToString();
            waterAmountProgressBar.Value = _hump.GallonsOfWater;
        }

        private void Camels_Load(object sender, EventArgs e)
        {
            waterAmountProgressBar.Minimum = 0;
            waterAmountProgressBar.Maximum = _hump.MaximumGallonsOfWater;
        }
    }

    internal class Hump
    {
        .
        .
        .

        internal void addWater(int gallonsToAdd)
        {
            _gallonsOfWater += gallonsToAdd;
            if (_gallonsOfWater > _maximumGallonsOfWater)
                _gallonsOfWater = _maximumGallonsOfWater;
        }

        internal void removeWater(int gallonsToRemove)
        {
            _gallonsOfWater -= gallonsToRemove;
            if (_gallonsOfWater < 0)
                _gallonsOfWater = 0;
        }
    }
}

```

Our naming conventions for variables, methods, and controls should enable you to read the code easily and determine how it works. We call either `_hump.addWater()` or `_hump.removeWater()` to add or remove water. We added the `updateWaterDisplay()` method to refresh the textual and graphical display of the amount of water in the camel's hump. Since our `Hump` class does boundary checking on the water amount, we don't have to worry about trying to add or remove too much water.

We've also added code to the `Camels_Load` event handler to initialize the minimum and maximum range of the `ProgressBar` control. The `Camels_Load` event is called just before the Form is displayed for the first time.

The ref Keyword and Passing By Value Versus Passing By Reference

Previously, we mentioned that parameters can have parameter modifiers. For this lesson, we want to introduce the `ref` keyword, and show the difference between passing an argument to a method by value and by reference.

By default, an argument passed to a method as a parameter is said to be passed *by value*, meaning that the value of the argument is copied into the method, and that any changes made to this copy do not result in a change to the value of the original argument in the code that called the method. This rule applies to all data types, although in a future lesson we'll discuss the difference between value data types and reference data types. For this lesson, we are only concerned with value data types, such as integer.

When you use a parameter modifier like `ref`, the modifier is used both in the method parameter list definition (the fingerprint), and when you invoke the method, preceding the argument with the `ref` keyword. The `ref` parameter modifier requires that any method argument passed by reference be initialized before being passed.

Let's modify our `Camels.cs` project to use the `ref` keyword. We'll add a new Form event handler for the `Shown` event, an event that occurs when a Form is shown for the first time.

Select the **Camels.cs (Design)** Editor, select the Form (by clicking the title bar), select the Events icon  in the Properties window, and double-click the **Shown** event. Add code to the `Camels_Shown` event handler, and add the `testChangeInteger` method as shown.

Camels.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Camels
{
    public partial class Camels : Form
    {
        private Hump _hump = new Hump();

        public Camels()
        {
            InitializeComponent();
        }

        private void addWaterButton_Click(object sender, EventArgs e)
        {
            _hump.addWater(Int32.Parse(waterAmountNumericUpDown.Value.ToString()
));
            updateWaterDisplay();
        }

        private void removeWaterButton_Click(object sender, EventArgs e)
        {
            _hump.removeWater(Int32.Parse(waterAmountNumericUpDown.Value.ToStrin
g()));
            updateWaterDisplay();
        }

        private void updateWaterDisplay()
        {
            waterAmountLabel.Text = _hump.GallonsOfWater.ToString();
            waterAmountProgressBar.Value = _hump.GallonsOfWater;
        }

        private void Camels_Load(object sender, EventArgs e)
        {
            waterAmountProgressBar.Minimum = 0;
            waterAmountProgressBar.Maximum = _hump.MaximumGallonsOfWater;
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void testChangeInteger(ref int value)
        {
            value += 10;
        }

        private void Camels_Shown(object sender, EventArgs e)
        {
            int changeInteger = 0;

            testChangeInteger(ref changeInteger);
            MessageBox.Show("changeInteger value: " + changeInteger);
        }
    }

    internal class Hump
```

```

    {
        ... no change ...
    }
}

```



and to save and run the program. As soon as the Form is displayed, a MessageBox dialog box appears, showing that `changeInteger` is 10. Let's discuss how it works.

OBSERVE:

```

.
.
.
    private void testChangeInteger(ref int value)
    {
        value += 10;
    }

    private void Camels_Shown(object sender, EventArgs e)
    {
        int changeInteger = 0;

        testChangeInteger(ref changeInteger);
        MessageBox.Show("changeInteger value: " + changeInteger);
    }
.
.
.

```

Without the `ref` parameter modifier (try it!), the call to the `testChangeInteger()` method would have no impact on the `changeInteger` argument and the Message Box would show the value as 0. By adding `ref`, the addition of 10 to the `value` parameter is really referencing the `changeInteger` argument.

Class Constructors

Earlier in the lesson, we listed the various class function members, which in addition to methods includes *constructors*. We've used constructors before, and the Camels project we've worked on is no exception. A class may have no constructor, in which case C# will provide a default constructor that simply creates an instance of the object, or in OOP language, *instantiates* it, and initializes any class member variables to their default values. Default values for data types are what you might expect: **false** for boolean data types, and 0 for numeric data types. Also see Microsoft's online MSDN article [Default Values Table](#).

Of course, a class may also have one or more constructors. A constructor with no parameters is called the *default constructor*, and is the constructor automatically provided by C# when no constructors are coded. Below is the syntax definition of a constructor.

Constructor Syntax Definition

```

[access modifiers] constructorName ( [parameters] )
{
    // Constructor body
}

```

The syntax definition of a constructor is very similar to that of a method, except that a constructor does not have a return type. You should note that typically you use an access modifier that allows the constructor to be invoked when instantiating your classes. If you use a **private** access modifier for a constructor, that constructor will never be called. Just as with methods, which constructor is called when a class is created is determined by its *signature*, the parameters included with the call to the constructor.

Note

You can code a **private** constructor, but if it's the default constructor, you will prevent C# from automatically adding a default constructor, making your class uninstantiable. If you have other constructors with different parameters, one of these other constructors would have to be called to enable creating an instance of your class. There are situations where you may not want a class to be created, and using a **private** default constructor would accomplish this task, although a better method is to make the class **static**, a scenario we'll cover in a later lesson.

We'll continue to discuss constructors as we proceed with the other lesson topics. Constructors have a number of other features we can take advantage of, but many of them are in the context of other C# constructs and language features.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Initializers, Object Initializers, and Anonymous Types

In this lesson, we'll continue our discussion of classes by exploring object initialization and creation using the **new** keyword, and discuss more about constructors and accessors. We'll begin the discussion by exploring the difference between value data types and reference data types.

Types of Data Types

Value Data Types

In a previous lesson, we discussed the use of the parameter modifier **ref** to pass the parameter by reference instead of by value (the default method). Passing by reference means that any changes made to an argument within a method will be reflected in the calling code. Why? The parameter that uses the **ref** modifier is really a reference to the variable that was passed in as an argument to the method. In fact, when you add the **ref** parameter modifier to a parameter, the data type of the parameter is no longer the data type you've indicated in the method signature, but instead a reference to the data type of the parameter.

When thinking about pass by value and pass by reference, remember that some data types are known as *value* data types, such as integers, floating points, decimal numbers, and booleans. Value data types store the actual value of the data type. Whenever you pass a value data type as a method argument, you're passing a copy of the original variable, or, understood another way, in the method, you simply have the value of the argument, and no reference to the original variable in the calling code.

Reference Data Types

In contrast to value data types, C# contains numerous examples of *reference* data types, which are effectively objects. When you have a variable that is of a reference data type, the variable does not contain the actual variable data, but instead contains a reference to the location in memory where the object data is stored. Reference data types include strings, classes, arrays (which we'll discuss later), and other C# object constructs.

To help you appreciate why we need a reference data type, consider some of the classes we've created so far in our lessons. Some of these classes had four or five class member variables. When you create an instance of a class with a number of class member variables, each of those variables has memory storage (also known as allocation) needs, depending on their data types. And, if a class member variable is itself a reference data type, such as another class, the embedded class data type may have other class member variables. When we created an instance of the class, our class variable became a reference to the memory location where we created our class, whatever the memory requirements were. It would be very inefficient, and potentially time-consuming, if we had to copy all of that allocated memory every time we passed our class reference variable into a method.

Just as with value data types, when you call a method, and use a reference data type as an argument, you do pass a copy of the reference data type. The difference, though, is that by definition a reference data type is already a reference to a location in memory, so while you're not getting the original reference from the calling program, you are still getting a copy of the memory reference. What does this mean? This means that when you pass a reference data type into a method, without the **ref** keyword, any changes made in the method to the referenced object will be reflected in the calling program. Changes made in a method to a reference data type are in fact changing the underlying object in memory.

But what happens if you do use the **ref** parameter modifier with a reference data type? Again, just as with value data types, using **ref** with a reference data type passes a reference to the original reference variable in the calling program. So, not only can you continue to modify the data for the object wherever it is located in memory, you can also change where in memory the reference refers. Why would you ever want to change the reference location? Frequently, objects are used dynamically, meaning that a variable amount of data needs to be stored, which of course means the amount of memory allocated for an object changes. Using **ref** with a reference data type allows for the allocation of memory to accommodate the increased memory needs of an expanded object. Once new memory is allocated, the **ref** reference variable can be switched to point to this new memory location.

Note

You may be wondering what happens to previously allocated memory when we allocate new memory in a method and reassign a reference variable to point to this new memory location. If the reference parameter used the **ref** parameter modifier, the old memory is no longer referenced, and eventually will be deallocated by the Common Language Runtime (CLR) garbage collection process. The garbage collector monitors the memory heap for any unreferenced objects, and will release the resources held by these objects. For this reason, objects created in C# are said to be managed, in that the CLR monitors references to the objects.

Boxing and Unboxing

You can convert from a value type to a reference type, and from a reference type to a value type. Converting a value data type to a reference data type is called *boxing*, and the reverse process *unboxing*. We won't be exploring boxing and unboxing in this lesson, but you may reference the MSDN Online Reference article [Boxing and Unboxing](#) for more information.

The new Keyword

We've seen the **new** keyword before, so we'll just reiterate that the **new** keyword with reference data types is used to create memory space for objects on the heap, and invoke object constructors. Using the **new** keyword is not limited to reference data types, and can be used with value data types; however, as we've learned, value data types do not require an architecture that accommodates the possibility of a variable amount of memory, so for value data types, when you use the **new** keyword, you are invoking the default constructor for that value data type, and initializing the variable to the default value appropriate for that data type. An example will help.

Value Data Type new Example

```
// The following lines of C# code are ALL equivalent.
int myInt = new int();

int myInt = 0;
```

We also should remind you that all data types in C# are related to underlying .NET data types as part of the Common Type System (CTS). Why is this important? Besides making it possible to create programming languages that ensure data type compatibility, it also means that for every C# data type, there is an underlying .NET data type. Let's add some more equivalent statements to our earlier example.

Value Data Type new Example

```
// The following lines of C# code are ALL equivalent.
int myInt = new int();

int myInt = 0;

// CTS System.Int32
Int32 myInt = new Int32();

Int32 myInt = 0;

// Mixing
int myInt = new Int32();

Int32 myInt = new int();
```

Note

You may be wondering: if I use **new** with a value data type like **int**, is the resulting variable placed on the heap? The answer is no. We'll discuss more about the heap, and the stack, in a later lesson.

Object Initialization

We have a number of mechanisms we can use to initialize properties of an object. We're familiar with two of these mechanisms already: *constructors* and *accessors*. We'll revisit both of these methods, and introduce a third

mechanism: *object initializers*.

Constructors and Object Initialization

As we've defined our classes, we've also defined constructor methods that are called when an instance of a class is created. If multiple constructors are defined, C# determines which constructor to call by matching the fingerprint, or parameters. If no parameters are specified, then the default constructor is called. All classes in C# must have a constructor. If you as the programmer do not define any constructors for a class, C# will implicitly define and call an empty default constructor.

A class represents an object, and as such we use C# syntax to define properties for our class by creating private class member variables. We can initialize these properties through constructors, as seen in the code below.

Constructor Initialization Sample

```
// Sample code for illustration purposes, will not compile.

class Employee
{
    private int _id;
    private string _lastName;
    private string _firstName;

    public Employee()
    {
        // Default constructor
    }

    public Employee(int id, string lastName, string firstName)
    {
        _id = id;
        _lastName = lastName;
        _firstName = firstName;
    }
}

// How we would initialize object properties using constructor.
Employee employee = new Employee(123, "Lastname", "Firstname");
```

In the code above, the **Employee** class defines a default constructor (no parameters), and a constructor that takes three parameters: an employee ID, last name, and first name. When this constructor is called, an **Employee** object instance can be created setting these three properties. Using a constructor to initialize methods can be called *constructor object initialization*.

While using a constructor for object initialization is convenient, what happens if we add dozens of properties to the class definition? Rather than using multiple constructors, we can use class accessors.

Accessors and Object Initialization

As class properties expand, we can use class accessors for setting properties of an object. Unlike with a constructor, where we can initialize object properties when we create the object, setting object properties through an accessor requires separate lines of code for each property we want to initialize, as in the example below.

Accessor Initialization Sample

```
// Sample code for illustration purposes, will not compile.

class Employee
{
    private int _id;
    private string _lastName;
    private string _firstName;
    private DateTime _dateOfBirth;

    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }
    public string LastName {
        get { return _lastName; }
        set { _lastName = value; }
    }
    public string FirstName {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }

    public Employee()
    {
        // Default constructor
    }

    public Employee(int id, string lastName, string firstName)
    {
        _id = id;
        _lastName = lastName;
        _firstName = firstName;
    }
}

// How we would initialize object properties using accessors.
Employee employee = new Employee();
employee.Id = 123;
employee.LastName = "Lastname";
employee.FirstName = "Firstname";
employee.DateOfBirth = Convert.ToDateTime("3/1/1990");
```

In the example above, we've added accessors for the prior class properties, as well as adding a new accessor for **DateOfBirth**. To set the date of birth property, we would access this new accessor. The sample initialization code has also been modified to set all of the object properties using accessors, so we really don't need the second constructor.

Object Initializers

If an object has more and more properties, we could define a number of constructors with different parameters, but the possible combination of parameters quickly becomes unmanageable. Of course, we could use accessors for each new property, but as we've seen, initializing properties requires a separate line of code for each property. To address this issue, C# supports the concept of object initialization.

Object initializers employ a syntax that calls the default constructor, and then calls each accessor for properties we've indicated we want to set. We've modified our sample code below, removing the unneeded constructor.

Object_INITIALIZER Sample

// Sample code for illustration purposes, will not compile.

```
class Employee
{
    private int _id;
    private string _lastName;
    private string _firstName;
    private DateTime _dateOfBirth;

    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }
    public string LastName {
        get { return _lastName; }
        set { _lastName = value; }
    }
    public string FirstName {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }

    public Employee()
    {
        // Default constructor
    }
}

// How we would initialize properties using object initializers
Employee employee = new Employee()
{
    Id = 123,
    LastName = "Lastname",
    FirstName = "Firstname",
    DateOfBirth = Convert.ToDateTime("3/1/1990")
};
```

As you can see, we didn't change the class definition, other than removing the unneeded constructor. You can also see that after we call the **new** operator, we then specify the properties we want to set, and the value for each property. We can include all of the properties, or just the ones we want to set.

You'll notice that we've highlighted the parentheses as **()**. Why? For object initializers, by default, they use the default constructor. When using the default constructor, you don't need to specify the parentheses when using the object initializer syntax, and omitting the parentheses is very common. You can use object initialization with constructors that are not the default constructor, which would of course require the parentheses.

Note

Remember that C# will provide a default constructor if you don't provide one, but this only happens if you have not created any constructors. If you create a constructor with parameters but do not include a default constructor, and then attempt to create the object with no parameters, you'll get an error message indicating that your class does not contain a constructor that takes 0 arguments.

Automatic Properties

C# supports a different syntax for class properties called *automatic properties*. Previously, we've prefixed all class variables with an underscore, and then created accessors with the same name as the variable, replacing the lowercase letter with an uppercase letter and removing the underscore.

With automatic properties, you don't need to explicitly declare a class variable. However, you should only use automatic properties when you do not need any code in the property get or set bodies. If you do need additional logic, you should use the private class variable technique (also known as full property syntax) we've used throughout the lessons. To access the property from within your class, use the name of the property. The sample code below illustrates a rewrite of the Employee class using automatic properties. In all cases, you'll notice that we've removed the private class variables.

Object Initializers Using Automatic Properties

```
class Employee
{
    public int Id { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public Employee()
    {
        // Default constructor
    }
}
```

Many C# programmers prefer using automatic properties over explicit variables using the underscore prefix. If you later decide you need coding logic with your automatic properties, you'll need to convert your automatic property to the full property syntax. Currently, there is no automatic refactoring option in Studio to make this conversion, so you'll have to make the conversion yourself.

You might wonder why you would use an automatic property over a public class variable, or field. Essentially, automatic properties appear to be simply a more lengthy coding mechanism to store and retrieve data from a class variable, but automatic properties offer many advantages over public class fields. Some of the advantages listed will be covered in later lessons.

- Automatic properties are a more concise coding style over full property syntax.
- Properties accept a breakpoint when debugging, but fields do not.
- Support read-only and/or write-only accessibility.
- Conversion from an automatic property to a full property with private class variable does not represent a change in the compiled contract between any code that accesses the property, whereas changing from a field to a property does break the contract.
- The concept of reflection, an advanced topic, works the same with automatic and full properties, but differs between fields and properties.
- Data binding is possible to properties, but not to fields.

For the rest of this lesson, we'll continue to use the explicit variable definitions with the underscore prefix, or full property syntax, although we'll insert automatic properties in later lessons.

Coding: The Employee Database

Let's tie everything together with an employee database program to work with the concepts we've covered in the lesson.

Create a new Visual Studio Windows Forms Application named **EmployeeDatabase**. Change the name **Form1.cs** to **EmployeeDatabase.cs**, and change the title bar's Text property to **Employee Database**. Click  to save your changes.

Now let's add the controls we'll need.

Modify the form by adding the controls listed below. Change each control **Name** property to match the **Name Property** column, and each **Text** property to match the **Text Property** column. Arrange the controls as shown.

Object	Name Property	Text Property
Label	lastNameLabel	Last Name:
Label	firstNameLabel	First Name:
Label	dateOfBirthLabel	Date Of Birth:
Label	employeeCountLabel	Employee Count:
TextBox	lastNameTextBox	
TextBox	firstNameTextBox	
TextBox	dateOfBirthTextBox	
Label	employeeCountValueLabel	0
Button	addEmployeeButton	Add
Button	exitButton	Exit



to save your changes.

Add the following events to the EmployeeDatabase Form by double-clicking each Button control.

Control	Event	Event Name
addEmployeeButton	Click	addEmployeeButton_Click
exitButton	Click	exitButton_Click



to save your changes. Next, let's add an Employee class.

In the Solution Explorer, right-click the **EmployeeDatabase** project and select **Add | Class**. In the Add New Item dialog box, change the Name property to **Employee.cs** and click **Add**. In the EmployeeDatabase.cs Code Editor, modify the code according as shown below.

Employee.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EmployeeDatabase
{
    public class Employee
    {
        private int _id;
        private string _lastName;
        private string _firstName;
        private DateTime _dateOfBirth;

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }
        public string LastName {
            get { return _lastName; }
            set { _lastName = value; }
        }
        public string FirstName {
            get { return _firstName; }
            set { _firstName = value; }
        }
        public DateTime DateOfBirth
        {
            get { return _dateOfBirth; }
            set { _dateOfBirth = value; }
        }

        public Employee()
        {
            // Default constructor
        }
    }
}
```

Click  to save your changes. The Employee class should look very similar to the sample code we were using before, and in fact, it's identical code. So, let's switch back to the EmployeeDatabase.cs file, and work with the Employee class to illustrate object initializers.

Select the **EmployeeDatabase.cs** Code Editor, and modify the code again, as shown below.

EmployeeDatabase.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace EmployeeDatabase
{
    public partial class EmployeeDatabase : Form
    {
        private List<Employee> _employees = new List<Employee>();

        public EmployeeDatabase()
        {
            InitializeComponent();
        }

        private void addEmployeeButton_Click(object sender, EventArgs e)
        {
            // Increment employee counter.
            int employeeCount = _employees.Count + 1;

            // Validate date of birth.
            DateTime dateOfBirth;
            if (!DateTime.TryParse(dateOfBirthTextBox.Text, out dateOfBirth))
            {
                dateOfBirth = DateTime.MinValue;
            }

            // Use object initializers through default constructor.
            Employee employee = new Employee()
            {
                Id = employeeCount,
                LastName = lastNameTextBox.Text,
                FirstName = firstNameTextBox.Text,
                DateOfBirth = dateOfBirth
            };

            // Add new employee object.
            _employees.Add(employee);

            // Update the employee count display
            employeeCountValueLabel.Text = employeeCount.ToString();
        }

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

Click  and  to save your changes and run the program. Enter a last and first name and a date of birth and click **Add**. The employee count should update, and continue to update if you add more employees. When you're satisfied that everything works, click **Exit**. Let's discuss how this code works.

OBSERVE: EmployeeDatabase.cs

```
.
.
.
namespace EmployeeDatabase
{
    public partial class EmployeeDatabase : Form
    {
        private List<Employee> _employees = new List<Employee>();

        public EmployeeDatabase()
        {
            InitializeComponent();
        }

        private void addButton_Click(object sender, EventArgs e)
        {
            // Increment employee counter.
            int employeeCount = _employees.Count + 1;

            // Validate date of birth.
            DateTime dateOfBirth;
            if (!DateTime.TryParse(dateOfBirthTextBox.Text, out dateOfBirth))
            {
                dateOfBirth = DateTime.MinValue;
            }

            // Use object initializers through default constructor.
            Employee employee = new Employee()
            {
                Id = employeeCount,
                LastName = lastNameTextBox.Text,
                FirstName = firstNameTextBox.Text,
                DateOfBirth = dateOfBirth
            };

            // Add new employee object.
            _employees.Add(employee);

            // Update the employee count display
            employeeCountValueLabel.Text = employeeCount.ToString();
        }
    }
}
```

Most of the code should be pretty familiar to you by now. We create an Employee list object named **_employees**. When the Add button is clicked, we validate the date of birth, and **if the date is invalid**, we set the employee's date of birth to the minimum date value. Why? A DateTime data type is a value data type, and as such cannot be set to **null**. We may want to consider creating our own date of birth class, or changing the date of birth to a string and checking for an empty string, to account for when no date of birth is given. Of course, we could also make it required, and force the user to enter a valid date.

Next, we come to the object initializer code, where we create an **employee** instance, and setting each of the object properties, including **Id**, **LastName**, **FirstName**, and **DateOfBirth**. As mentioned before, using the object initializer syntax allows us to set all of the properties, or only the properties we want. After creating the object, we add the **employee** object to the **_employees** List, and update the count of the number of employees.

So, as you can see, object initializers are a great alternative to creating numerous constructors, adding a great deal of flexibility to object initialization without adding a lot of cumbersome code to our classes.

Anonymous Types

We have one final topic to cover that uses object initialization: *anonymous types*. An anonymous type is exactly what

the name implies, an object with an implicit type (thus anonymous). Below is an example of an anonymous type.

Anonymous Type
<pre>// Create an anonymous type var employee = new { Id = 123, LastName = "Lastname", FirstName = "Firstname", DateOfBirth = Convert.ToDateTime("3/1/1990") }; // Output the LastName property of the anonymous type Console.WriteLine("The employee last name is " + employee.LastName);</pre>

As you can see from our object declaration, we're using a new C# keyword, **var**. We then specify the variable name, **employee**, then use the **new** keyword without a class name to create an object instance. We also include object initializers to declare *read-only* properties. Note that we did say *read-only*: once instantiated and initialized, anonymous properties cannot be modified.

Once created, an anonymous type instance can only be referenced through the anonymous type instance variable. In our example, the anonymous type instance variable is **employee**. Anonymous types only include read-only properties. No other class functions, such as events or methods, are allowed. Anonymous instance variables must not be null. The name of the anonymous data type is not accessible to C#, which means you also can't create other variables from an anonymous type. If you need a data type name, use a class.

When would you use an anonymous type? Anonymous types are useful from creating "on-the-fly" objects that we only need to access for a short time. We'll see uses of anonymous data types when we talk later about arrays, and about the .NET query language LINQ.

Popup Forms and Dialogs

We've created popup forms before, even changing them to be Dialog boxes. We want to further explore how to make a popup form as part of expanding your knowledge of how to use C#. Also, you may find this information in your homework projects.

A popup form is used to prompt the user for data, or to make a decision. Popup forms may be modal, which means that the focus of an application is with the popup form, not allowing you to change to any other part of the application, until you dismiss the popup form. If a popup form is not modal, it is modeless, which means you can switch back to the application without dismissing the popup form.

When creating a popup form, we add a new Windows Form to our application, and modify the form just as we have been doing throughout our lessons with the main form that is provided by C# automatically when we create a Windows forms application. To instantiate an instance of the form, we use the **new** operator, then call either the ShowDialog() method of the form to display the form as a modal popup, or the Show() method of the form to display the form as a modeless popup.

Windows forms, when dismissed, such as when you click an OK or Cancel button, may return a **DialogResult** result that the calling code may evaluate to determine what needs to happen next. For any Button on a form, you can change the DialogResult property to any of the possible choices: None, OK, Cancel, Abort, Retry, Ignore, Yes, or No. In fact, if you set the DialogResult property of the Cancel Button control to **Cancel**, when you click on the Cancel Button, you won't need to add an event handler that calls the form's Close() method. For the OK Button control, you could also set the DialogResult property to **OK**, but a better solution is to set the DialogResult property in the OK Button event handler and call the Close() method once you've verified that everything on the Form is correct. In other words, you may want to validate the form after the user clicks the OK button, and determine if you want to actually close the Form. If there are errors, you may want to display some type of error alert, and not set the DialogResult or Close the Form.

The code below illustrates how to create a popup form, display the form as a modal dialog box, and then check to see if the DialogResult value was **OK**.

Modal Form

```
// Declare an EmployeeForm variable, create an instance of the EmployeeForm Form,
// and assign the new EmployeeForm instance object to the variable
EmployeeForm employeeForm = new EmployeeForm();

// Display the employeeForm instance, and assign the return results to
// a DialogResult variable
DialogResult dialogResult = employeeForm.ShowDialog();

// Check the return value, and extract any data
if (dialogResult == DialogResult.OK)
{
    // Extract the employee information from the Form using the accessor
    Employee employee = employeeForm.EmployeeInfo;
    // Update the employee ID
    employee.Id = _employees.Count + 1;
    // Add the employee object to our List
    _employees.Add(employee);
    // Update a ListBox that displays the employee information
    displayEmployees();
}
```

In addition to a form, you can use the C# **MessageBox** class for simple user responses, like Yes or No. The **MessageBox** has numerous overloaded methods. We're going to demonstrate the method with parameters for the text to display in the dialog box, text for the caption of the dialog box, and which buttons to display. The code snippet below also includes capturing the result using the **DialogResult** class.

MessageBox

```
DialogResult dialogResult = MessageBox.Show("Delete this employee?", "Delete Employee",
MessageBoxButtons.YesNo);
```

This short tutorial on popup Forms and dialogs should help you as you continue to learn C# and create Windows Forms based applications.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

The this Keyword, Static and Instance Members

In this lesson, we'll revisit the topic of class instances and the **this** keyword, and cover the concept of static properties, methods, and classes.

Class Instances

Instances

By now, you've had numerous opportunities to create classes, and then create instances of the classes. When you use the **new** keyword, you are creating a *class instance*. What exactly is a class instance? When you create a class instance, that object exists in memory on the *memory heap*. Each class instance resides in its own unique memory space. As we've learned, classes are composed of variables and class functions (methods, accessors, etc). As you create multiple class instances, class variables must be unique on a *per-class instance* basis. So, if you created hundreds of instances of the same class, you would have hundreds of unique versions of each class variable.

What about programming code? Do we need multiple copies of the code? No, we do not, so class functions are only stored once. If you're interested in where in memory the single code copy resides, that's not as straightforward to answer. Before our C# code can execute, it needs to be compiled. The .NET architecture employs a mechanism known as Just In Time (JIT) compilation to compile the code, and keeps the compiled code available should it be needed. This compiled code doesn't remain indefinitely, so it's possible that the class functions are essentially not in memory. When they are finally compiled, they're kept in what is typically referred to as a code memory segment, a space reserved for code. Effectively, you can think of this memory as a dynamic code heap.

What does this notion of a class instance mean to us as C# OOP programmers? As we've learned so far, if we want to use a class, we have to create an instance of that class. Using **public** class functions or fields (public class variables), we can access and utilize our class instance.

Self-Referencing Using the 'this' Keyword

You may already be familiar with the **this** keyword. Many new OOP programmers have trouble understanding what **this** stands for and how to use it. Let's dive in.

The **this** keyword is only used within a class, and effectively refers to the class instance. What does that mean? From within a class, whenever you want to access class instance members, you use **this**. If the class instance member name is unique, then you can omit the **this** keyword. There are times, though, when you must use it, as we'll discuss next. So, when you see the **this** keyword, remember that **this** represents the current class instance.

When is using **this** optional, and when is it required? Below are the most common circumstances (we'll refer back to these numbers [later](#)):

1. May Use: To clarify the code refers to a class instance member.
2. Must Use: To qualify class instance members that would otherwise be hidden by other items of the same name, known as shadowing.
3. Must Use: When passing the instance object itself as a parameter to a method.
4. Must Use: When returning an instance object itself from a method.
5. Must Use: When passing parameters between class constructors, known as constructor chaining.
6. Must Use: To declare class indexers (discussed in a later lesson).
7. Must Use: To declare class extension methods (discussed in a later lesson).

Coding: Boxing

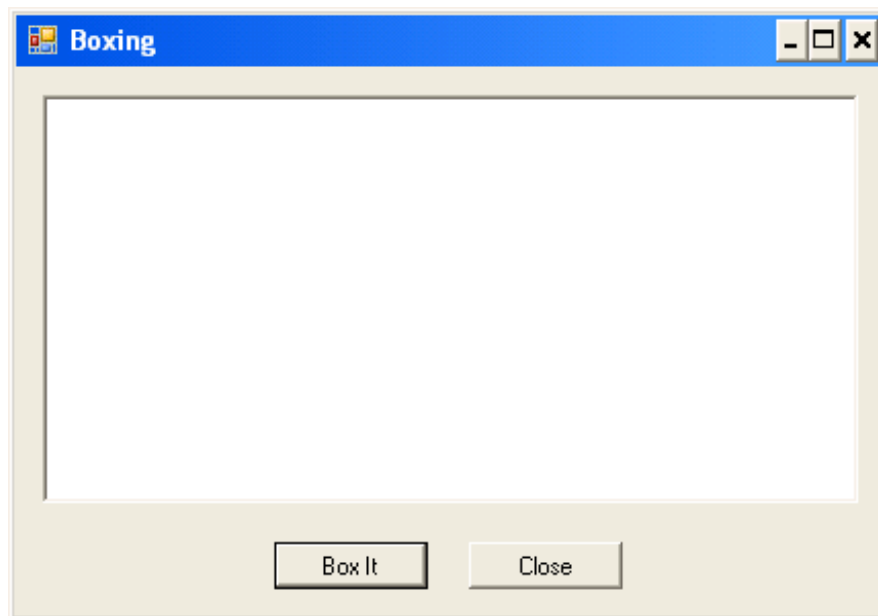
Let's create a project to experiment with the **this** keyword and classes.

Select **File | New | Project**. Change the project name to **Boxing** and click **OK**.

Click the entry for **Form1.cs** and change it to **Boxing.cs**. Change the form title bar's Text property in the Properties

Window to **Boxing**. Click  to save your changes.

Modify the form to look like this:



Change each control's Name property to match the Name Property column, and Text property to match the Text Property column, below. When you finish, click  to save your changes.

Object	Name Property	Text Property
ListBox	boxingListBox	
Button	boxingButton	Box It
Button	closeButton	Close

Add event handlers for the **boxingButton** and **closeButton** controls by double-clicking on each control to automatically generate the default event handler code. Studio generates an appropriate name for the event handler that consists of the name of the control, an underscore separator, and the event name.

Modify the **Boxing.cs** code as shown below. The event handler code was added previously.

```
Boxing.cs

using ...

namespace Boxing
{
    public partial class Boxing : Form
    {
        public Boxing()
        {
            InitializeComponent();
        }

        private void boxButton_Click(object sender, EventArgs e)
        {
        }

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

As you can see, all we added was the code to close the form when the closeButton is clicked. Let's next add a couple of new classes.

Right-click on the **Boxing** project item in the Solution Explorer, and select **Add | Class....** In the Add New Item dialog box, change the Name from Class1.cs to **Box.cs**, then click **Add**.

In the Box.cs Code Editor, modify the code as shown:

```
Box.cs

using ...

namespace Boxing
{
    class Box
    {
        // Automatic class properties.
        public double Length { get; set; }
        public double Width { get; set; }
        public double Height { get; set; }

        // Typical constructor used with double, double, double fingerprint.
        public Box(double Length, double Width, double Height)
        {
            this.Length = Length;
            this.Width = Width;
            this.Height = Height;
        }

        // Default constructor, using this to call constructor with double, double, double fingerprint,
        // Note using explicit 0.0 double literal to prevent any forced type conversions, and
        // placing empty method body curly braces on same line to conserve space
        public Box() : this(0.0, 0.0, 0.0) { }

        // Private constructor that takes a Box as a parameter, used to clone itself.
        private Box(Box Box) : this(Box.Length, Box.Width, Box.Height) {}

        // A method to return a cloned version of the current Box.
        // Note that cloning is rarely this simple!
        public Box Clone() { return new Box(this); }

        // Calculate volume of box.
        public double Volume()
        {
            return this.Length * this.Width * this.Height;
        }
    }
}
```

Click  to save your changes. Before running or explaining this code, we'll also add the code that will use the Box class.

In the Boxing.cs Code Editor, modify the code as shown:

Boxing.cs

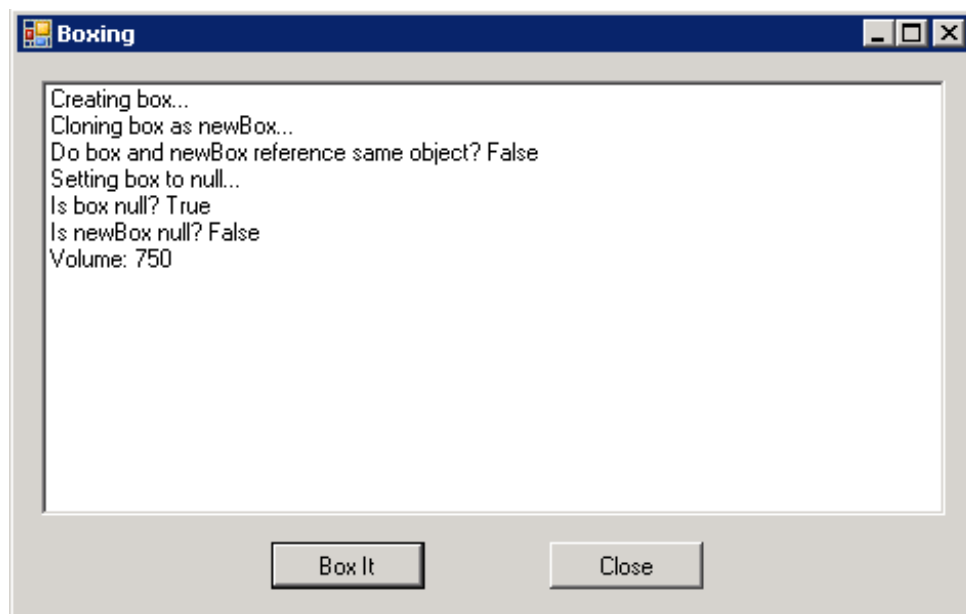
```
using ...

namespace Boxing
{
    public partial class Boxing : Form
    {
        public Boxing()
        {
            InitializeComponent();
        }

        private void boxingButton_Click(object sender, EventArgs e)
        {
            // Fun with Box.
            boxingListBox.Items.Add("Creating box...");
            Box box = new Box(5, 10, 15);
            boxingListBox.Items.Add("Cloning box as newBox...");
            Box newBox = box.Clone();
            boxingListBox.Items.Add("Do box and newBox reference same object? " + ReferenceEquals(box, newBox));
            boxingListBox.Items.Add("Setting box to null...");
            box = null;
            boxingListBox.Items.Add("Is box null? " + (box == null));
            boxingListBox.Items.Add("Is newBox null? " + (newBox == null));
            boxingListBox.Items.Add("Volume : " + newBox.Volume());
        }

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

Click  to save, and  to run the program. Click **Box It** to see the results:



Click **Close** to close the form. Now let's discuss how it works.

Box.cs

```
using ...

namespace Boxing
{
    class Box
    {
        // Automatic class properties.
        public double Length { get; set; }
        public double Width { get; set; }
        public double Height { get; set; }

        // Typical constructor used with double, double, double fingerprint.
        public Box(double Length, double Width, double Height)
        {
            this.Length = Length; [2]
            this.Width = Width; [2]
            this.Height = Height; [2]
        }

        // Default constructor, using this to call constructor with double, double, double fingerprint,
        // Note using explicit 0.0 double literal to prevent any forced type conversions, and
        // placing empty method body curly braces on same line to conserve space
        public Box() : this(0.0, 0.0, 0.0) { } [5]

        // Private constructor that takes a Box as a parameter, used to clone itself.
        private Box(Box Box) : this(Box.Length, Box.Width, Box.Height) {} [5]

        // A method to return a cloned version of the current Box.
        // Note that cloning is rarely this simple!
        public Box Clone() { return new Box(this); } [3]

        // Calculate volume of box.
        public double Volume()
        {
            return this.Length * this.Width * this.Height; [1]
        }
    }
}
```

As you examine this code, note the different uses of the **this** keyword, the **Box** class, and the constructors. Also, review the different ways we listed that you can use the **this** keyword. We've added numbers marked in **[red]** that relate to the numbers listed above to help you recognize which **this** usage is represented by the particular code. In our code so far, we've illustrated: **[1]**, using **this** to clarify that a referenced variable is a class variable; **[2]**, to prevent shadowing; **[3]**, passing **this** as a method parameter; and **[5]**, constructor chaining. We'll illustrate **4** in another class shortly.

Next, let's analyze the code in the `boxingButton_Click` event handler in `Boxing.cs`.

Boxing.cs

```
.  
.   
.   
    private void boxingButton_Click(object sender, EventArgs e)  
    {  
        // Fun with Box.  
        boxingListBox.Items.Add("Creating box...");  
        Box box = new Box(5, 10, 15);  
        boxingListBox.Items.Add("Cloning box as newBox...");  
        Box newBox = box.Clone();  
        boxingListBox.Items.Add("Do box and newBox reference same object? " + ReferenceEquals(box, newBox));  
        boxingListBox.Items.Add("Setting box to null...");  
        box = null;  
        boxingListBox.Items.Add("Is box null? " + (box == null));  
        boxingListBox.Items.Add("Is newBox null? " + (newBox == null));  
        boxingListBox.Items.Add("Volume : " + newBox.Volume());  
    }  
.   
.   
. 
```

The code to add items to a ListBox should be familiar by now. First, we create a **Box** instance, then declare a second **Box** variable, using the Clone method to create a copy of our first instance and setting our second **Box** variable to this cloned reference. We then test to see if both of the Box variables refer to the same object in memory using the **ReferenceEquals** method. When you run the code, you'll see that both Box variables do not reference the same memory location, so we know that our cloning created two separate instance objects.

Next, we set our first **Box** instance to **null**, effectively de-referencing the variable. Remember, .Net tracks references to objects, and when all references are removed, that memory is marked for deletion and will eventually be recovered by the Garbage Collector. Setting an instance variable to **null** removes that reference. We confirm that our first instance variable is **null**, and confirm that despite removing the first reference, the second instance variable is not **null**. Finally, we calculate and display the volume of our cloned **Box**.

Let's add a second class so we can illustrate **4**, returning **this** from a method.

Right-click on the **Boxing** project item in the Solution Explorer, and select **Add | Class....** In the Add New Item dialog box, change the Name from Class1.cs to **SquareBox.cs**, then click **Add**.

In the SquareBox.cs Code Editor, modify the code as shown below.

SquareBox.cs

```
using ...

namespace Boxing
{
    class SquareBox
    {
        // Box field.
        public Box Box;

        // Automatic property for single dimension value (length, width and height are
all the same).
        public double Dimension { get; set; }

        // Constructor with double fingerprint.
        // Note that we use this constructor to create our Box object.
        public SquareBox(double Dimension)
        {
            this.Box = new Box(Dimension, Dimension, Dimension);
            this.Dimension = Dimension;
        }

        // Default constructor, calls constructor with double fingerprint, passing doub
le literal,
        // using space-conserving method body coding convention.
        public SquareBox() : this(0.0) { }

        // Return a reference to the current object.
        public SquareBox Reference()
        {
            return this;
        }
    }
}
```

Now let's also add the code that will use the SquareBox class.

Select the **Boxing.cs** Code Editor, and modify the code as shown:

Boxing.cs

```
using ...

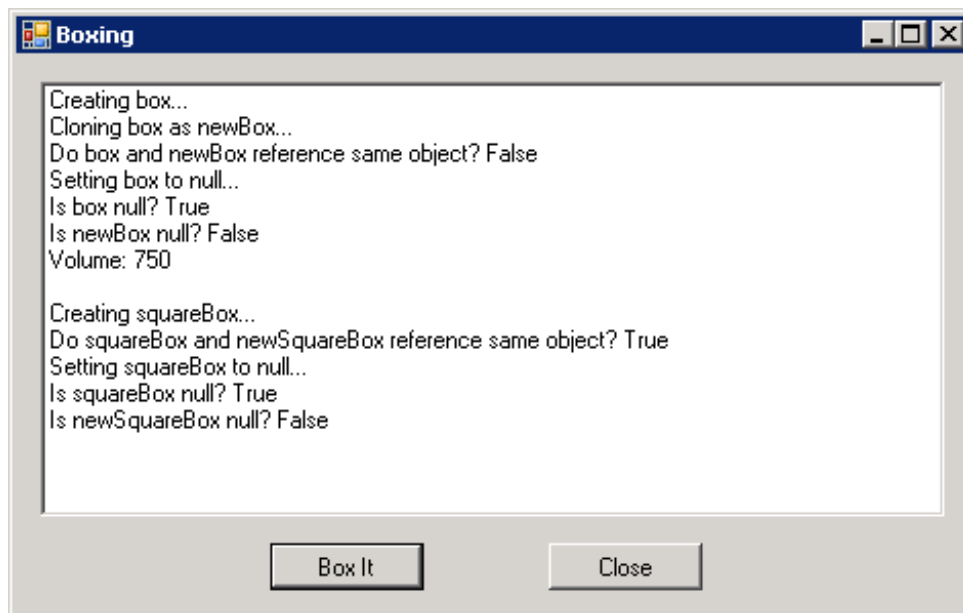
namespace Boxing
{
    public partial class Boxing : Form
    {
        public Boxing()
        {
            InitializeComponent();
        }

        private void boxButton_Click(object sender, EventArgs e)
        {
            // Fun with Box.
            boxingListBox.Items.Add("Creating box...");
            Box box = new Box(5, 10, 15);
            boxingListBox.Items.Add("Cloning box as newBox...");
            Box newBox = box.Clone();
            boxingListBox.Items.Add("Do box and newBox reference same object? " + ReferenceEquals(box, newBox));
            boxingListBox.Items.Add("Setting box to null...");
            box = null;
            boxingListBox.Items.Add("Is box null? " + (box == null));
            boxingListBox.Items.Add("Is newBox null? " + (newBox == null));
            boxingListBox.Items.Add("Volume : " + newBox.Volume());

            // Fun with SquareBox.
            boxingListBox.Items.Add(""); // Empty line
            boxingListBox.Items.Add("Creating squareBox...");
            SquareBox squareBox = new SquareBox(20);
            SquareBox newSquareBox = squareBox.Reference();
            boxingListBox.Items.Add("Do squareBox and newSquareBox reference same object? " + ReferenceEquals(squareBox, newSquareBox));
            boxingListBox.Items.Add("Setting squareBox to null...");
            squareBox = null;
            boxingListBox.Items.Add("Is squareBox null? " + (squareBox == null));
            boxingListBox.Items.Add("Is newSquareBox null? " + (newSquareBox == null));
        }

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

Click  and  to run the program. Click **Box It** to see the results:



Click **Close** to close the form. Let's discuss how it works.

```

SquareBox.cs

using ...

namespace Boxing
{
    class SquareBox
    {
        // Box field.
        public Box Box;

        // Automatic property for single dimension value (length, width and height are
all the same).
        public double Dimension { get; set; }

        // Constructor with double fingerprint.
        // Note that we use this constructor to create our Box object.
        public SquareBox(double Dimension)
        {
            this.Box = new Box(Dimension, Dimension, Dimension); [1]
            this.Dimension = Dimension; [2]
        }

        // Default constructor, calls constructor with double fingerprint, passing doub
le literal,
        // using space-conserving method body coding convention.
        public SquareBox() : this(0.0) { } [5]

        // Return a reference to the current object.
        public SquareBox Reference()
        {
            return this; [4]
        }
    }
}

```

Again, note the different uses of the **this** keyword, the **SquareBox** class, and the constructors. Again review the different ways we listed that you can use the **this** keyword. We've again added numbers marked in **[red]** to help you recognize which **this** usage is represented by the particular code. In the SquareBox class code, we've again illustrated **[1]**, using **this** to clarify that a referenced variable is a class variable; **[2]**, to prevent shadowing; and **[5]**, constructor chaining; and we've added **[4]**, returning the instance object itself from a method.

Next, let's analyze the code in the boxingButton_Click event handler in Boxing.cs that relates to using the SquareBox

class.

Boxing.cs

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

        // Fun with SquareBox.
        boxingListBox.Items.Add(""); // Empty line
        boxingListBox.Items.Add("Creating squareBox...");
        SquareBox squareBox = new SquareBox(20);
        SquareBox newSquareBox = squareBox.Reference();
        boxingListBox.Items.Add("Do squareBox and newSquareBox reference same object? " + ReferenceEquals(squareBox, newSquareBox));
        boxingListBox.Items.Add("Setting squareBox to null...");
        squareBox = null;
        boxingListBox.Items.Add("Is squareBox null? " + (squareBox == null));
        boxingListBox.Items.Add("Is newSquareBox null? " + (newSquareBox == null));
    }
.
.
.
```

Our Boxing.cs code to test the SquareBox class is almost identical to the code we used for the Boxing class but this time, rather than try to clone, or make a copy of, our object, we create a reference to our object. When we test to see if both instance variables reference the same object in memory, we find out that they do indeed reference the same object. When we set the first instance variable to **null**, we get the same results as before, but should we have? Would you have expected that, because we set the first instance variable to **null**, we would somehow be deleting the object in memory? Setting the instance variable to **null** only de-references that instance variable, so the second instance variable continues to reference the same memory location.

Static Members and Classes

Static Keyword

Now that we've revisited class instances, and discussed the **this** keyword as a reference to members of a current class instance, we can talk about **static** members and classes. The term **static** generally refers to something that doesn't change, and in the context of many other computer languages, that's part of the meaning, but in C#, the meaning and usage is reserved to work explicitly with class members and class definitions. In short, the **static** keyword qualifies class members and classes for usage without the need to create an instance of the class. From an OOP perspective, **static** members are used to separate data and behavior that is independent from whatever might happen to an object (class instance).

Note

If you research the term **static** online, you may come across the original usage of static in other computer languages: *static variables*. C# does not support *method-level* static variables, only *class-level* static variables.

Class Instances Versus Static Instances

So, what does independence from a class instance mean? Typically, we create a class instance, and then reference the class members using the class instance variable and the dot notation, as shown below. However, if a property or method is defined as **static**, you do **not** need the class instance variable (also shown below):

Class Instances Versus Static Instances

```
// Class instance
SquareBox squareBox = new SquareBox();
squareBox.Dimension = 25;

// Static instance
int numberOfDimensions = SquareBox.Dimensions;
```

In the example above, the **SquareBox** reference is *not* a reference to a class instance, but instead a reference to the class itself. We do *not* need to create an instance of the **SquareBox** class to reference the **Dimensions** property (note that the word is plural and would return 3, as a square box has three physical dimensions: length, width, and height). Below is a sample of how we would define a **SquareBox** class using the **static** keyword to create the C# **Dimensions** read-only automatic property (we're making the assumption that in our dimension, physical objects only have three dimensions, although sci-fi fans may disagree).

Static Class Property

```
public class SquareBox
{
    // Define the static property Dimensions.
    public static int Dimensions { get { return 3; } private set { } }
}
```

Note

The creation of a static read-only automatic property for **Dimensions** is a bit contrived, as we could just as easily have created a constant. Nevertheless, you should note that we did deviate from our earlier statement about when to use an automatic property: when you do not need additional coding. If you simply want to return a value, or perform other types of operations that will not involve the property name, you can add a method body to either the get or the set; however, you must either provide no method bodies, or both method bodies, even if the method body is empty, as is the case with the set for **Dimensions** above.

We can add **static** to any property, or method, in a class definition. Here's an example of a static method.

Static Class Method

```
public class SquareBox
{
    // Define the static method VolumeCalculation
    public static string VolumeCalculation()
    {
        return "Length x Width x Height";
    }
}
```

Just as with a static property, we can call a static method by referencing the class:

Calling a Static Class Method

```
string volumeCalculation = SquareBox.VolumeCalculation();
```

Classes can also include a static constructor. A static constructor could be used to initialize static fields in a class. A static constructor is called before an instance of a class is created, and before a static member function is called or accessed. A static constructor is only ever called once. Also, static constructors do not have an access modifier:

Static Class Method

```
public class SquareBox
{
    public static int Dimensions { get; private set; }

    // Define the static constructor to initialize the static property
    static SquareBox()
    {
        // Note that you can not use this with Dimensions, as we are not using a
        reference to an instance
        Dimensions = 3;
    }
}
```

Note

When accessing static class members, you cannot use the **this** keyword. Why? The **this** keyword is an instance reference variable, but with a static class member, we are not accessing the class instance.

We can create a static class. A static class *cannot* be instantiated. When you create a static class, any method or property that you want to make available via a static reference to the class must also be prefixed with the **static** keyword, as shown below.

Static Class with Static Method and Field

```
static class Volumes
{
    public static string Description = "Volume Calculations";

    public static string BoxVolumeCalculation()
    {
        return "Length x Width x Height";
    }

    public static string SphereVolumeCalcuation()
    {
        return "4/3 * PI * (radius * radis * radius)";
    }
}
```

So that's how we use the **static** keyword. You'll have a chance to code using **static** in this lesson's homework project.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Finalizing Objects, and Memory Concepts (Stack versus Heap)

In this lesson, we'll discuss topics surrounding memory, including the computer stack and heap, garbage collection, forcing memory deallocation, and class destructors.

Computer Memory

Computer Memory and Allocation

This lesson won't be an extensive discussion of the physical aspects of computer memory, but instead we'll focus on computer memory, and memory allocation, as they relate to developing software. Memory restrictions continue to become less an overall factor when writing a software application, but certain programming practices should always be considered or followed to reduce overall memory requirements, and prevent potential application failures due to errors related to memory allocation. We'll introduce these recommended practices throughout all future lessons as we introduce the related C# topic. For now, a simple guideline is to remove any unused variables and classes.

As we write computer software, we create a variety of C# components wrapped within classes that will require computer memory: methods, accessors, event handlers, variables, etc. As our program executes, memory is required to hold the code that is executing, the code that has just executed, the code to execute next, parameters passed into methods, and instantiated objects. This lesson will help you appreciate more of the technical details of memory allocation, and raise issues you should consider as you write software that creates instances of objects.

Stack Versus Heap

We've already discussed value data types and reference data types. We've learned that a value data type is stored on the *stack*, and that reference data types are stored on the *heap*. What exactly are the heap and stack?

The stack is a memory object allocated when a program begins execution, and has a variety of names: call stack, execution stack, control stack, run-time stack, program stack, etc. We'll simply continue to refer to it as the stack. A programming stack is a LIFO (last-in first-out) object, with typical functions such as push (to add to the top of the stack), and pop (to remove the top of the stack), and our stack is no different. As your program executes, the stack is created, and internal to C# and .NET, a reference is created that points to the top of the stack. Any data associated with the currently executing block of code is stored, or pushed, onto the stack as the data comes into scope. If we branch to a new section of code, we push the return reference onto the stack, and as we enter the new block of code, any data in the new block is added to the stack. As we return from our branches, we pop, or remove, the data related to where we were, and the return reference, essentially unwinding the stack contents.

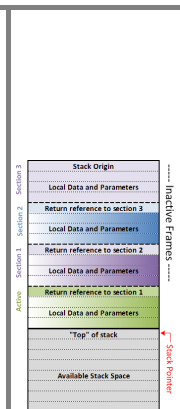
The image to the right represents a typical programming stack. The stack representation includes four complete stack frames, three inactive frames, and one active, or current, frame.

Each frame, except for the first frame, includes a return reference to the previous frame, as well as any local data, and any parameters that may have been included.

We've also included an area that represents available stack space. Typically, each process within an application is given a limited amount of stack space. Should an application exceed the allocated stack space amount, an error will occur. This limited amount of space is one reason why reference data types are allocated on the heap, and not the stack.

We've also indicated the **stack pointer**, representing the next allocation space on the stack.

Why is the "top" of the stack at the bottom? Memory allocation can proceed in either direction within the physical memory, and we want to make sure you don't make the assumption that the "top" of the stack is always at the top.



The heap is memory allocated "on the fly" when the **new** operator is called, or with reference data types such as strings. Potentially, the heap could grow as big as needed, only limited by the physical and virtual memory

of your computer. For C# applications, the heap is actually a "managed" heap, meaning that all heap allocations and deallocations are tracked by the .NET architecture. We'll continue to refer to the C# managed heap as simply the heap.

When objects are created on the heap, we can create multiple references to the object. As each additional reference is created, internally a reference counter is incremented. So long as at least one reference to an allocated heap object exists within scope, that object will remain on the heap. When the reference counter reaches zero, then that allocated heap object may be removed by the .NET Garbage Collector, which we'll discuss more shortly.

We've summarized the differences between the stack and the heap in the table below.

Type of Memory	Data Type	Item Order	Item Lifetime	Item Add	Item Removal	Removal Timing
Stack	Value Data Types	Sequential (LIFO)	Execution Block Scope	Push	Pop	Deterministic (upon Pop)
Heap	Reference Data Types	Random	Reference Count	new Operator	Garbage Collection	Non-Deterministic

As you review the differences between the stack and the heap, note that heap memory allocation is random, unlike the stack, where the memory allocated is in a sequential order. Random memory allocation means that when an object is instantiated on the heap, it could be anywhere, and as the memory is deallocated, blocks of previously allocated memory will be cleared, potentially leaving blocks of allocated memory, then unallocated memory, followed again by allocated memory.

Garbage Collection

The .NET managed heap tracks memory allocations, and when the references to the memory reach zero, .NET employs a mechanism known as the garbage collector to free up previously allocated memory. In the table listing the differences between the stack and the heap, we indicated that heap memory deallocation is *non-deterministic*, meaning that even though your code may no longer reference a block of heap memory, the actual deallocation of that memory may be delayed. If you're interested to know more about how the garbage collector works, and how it determines what memory is deallocated when, see the Microsoft MSDN article [Object Lifetime View](#).

Sometimes, you want to control how an object is deallocated, or perform some amount of cleanup just before destroying an object. We'll discuss this topic next.

Finalizing Objects and Destructors

As we've discussed, the .NET garbage collector will deallocate managed resources, but what about unmanaged resources that we may use, such as an open file connection, or a database connection? We may want to maintain such connections throughout the lifetime of an object, only releasing the connections and destroying any objects related to the connections when the object is finally destroyed. The destruction of unmanaged resources is accomplished as part of *finalizing* an object, and in C# is managed in the class destructor.

Class destructor? Yes, that's right, just as a class has a constructor, it may also have a destructor. Syntactically, a destructor is almost identical to a constructor, except that you include a tilde (~) before the class name. Destructors, just like constructors, do not return a value; however, unlike constructors, destructors do not take parameters. Purely static classes cannot include a destructor.

Class Destructor
<pre>public class Employee { public Employee() { // Empty constructor } public ~Employee() { // Destructor } }</pre>

Alright, let's do some coding!

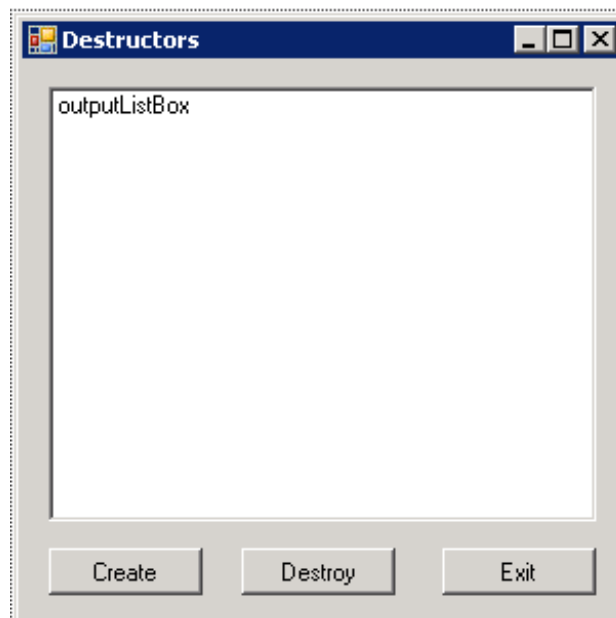
Coding

Let's explore ways to look at memory usage using the .NET System.GC class. We'll use the GC GetTotalMemory method to ask the garbage collector for an approximation of how much memory is currently allocated and thus managed. Why an approximation? The garbage collector is heavily optimized to prevent impacting the performance of an application, so memory flagged for deallocation may not yet be reclaimed. We will pass in **true** as a parameter to GetTotalMemory to ask the garbage collector to take additional time to try to more accurately calculate allocated memory. Typically, as OOP programmers using C#, we let the garbage collector deal with all of our managed memory requests, and would not call the GetTotalMemory method, except in our case to explore memory usage in .NET. After peeking into memory allocation, we'll use the .NET Diagnostics class to look into constructors and destructors by creating a class that uses a Stopwatch object to monitor how long an object is instantiated (created) then destroyed. This simple class will contain a List field (a public member variable) of data type object, which means that we can add any data type that is based on the object class. Specifically, we're going to create a large byte array to make sure we take a lot of space. We'll be covering arrays in a later lesson, so for now you'll just need to enter the code related to the byte array as it is presented in the lesson. You should note, though, that we use the **new** keyword with our byte array, so the memory allocated is from the heap, and is managed by the .NET garbage collector.

Select **File | New | Project**. Change the project name to **Destructors** and click **OK**.

Click the entry for **Form1.cs** and change it to **Destructors.cs**. Change the form title bar's Text property in the Properties Window to **Destructors**. Click  to save your changes.

Modify the form to look like this:



Change each control's Name property to match the Name Property column below, and Text property to match the Text Property column. Arrange the controls similar to the image. When you finish, click  to save your changes.

Object	Name Property	Text Property
ListBox	outputListBox	
Button	createButton	Create
Button	destroyButton	Destroy
Button	exitButton	Exit

Double-click each of the **createButton**, **destroyButton**, and **exitButton** controls to automatically generate the default event handler code.

Add an event handler for the Destructors Form Shown event by selecting the Form (clicking on the Form title bar), clicking the Events icon  to view the Events Properties Window, and double-clicking the **Shown** event.

Modify the **Destructors.cs** code as shown below.

Destructors.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Destructors
{
    public partial class Destructors : Form
    {
        public Destructors()
        {
            InitializeComponent();
        }

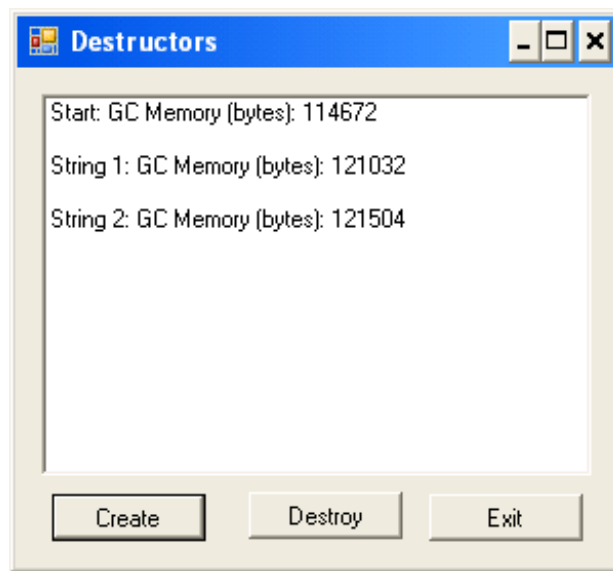
        private void createButton_Click(object sender, EventArgs e)
        {
        }

        private void destroyButton_Click(object sender, EventArgs e)
        {
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void Destructors_Shown(object sender, EventArgs e)
        {
            outputListBox.Items.Add("Start: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());
            outputListBox.Items.Add("");
            string testString = "This is a string that requires memory allocation on the heap";
            outputListBox.Items.Add("String 1: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());
            outputListBox.Items.Add("");
            testString += "Changing the string value results in deallocation of original heap memory, and allocation of new memory";
            outputListBox.Items.Add("String 2: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());
            outputListBox.Items.Add("");
        }
    }
}
```

Click  and  to save and run the program. You should see something like this (your memory values may be different):



Click **Exit** to close the program. Let's discuss how it works.

OBSERVE: Destructors.cs

```
.  
. .  
private void Destructors_Shown(object sender, EventArgs e)  
{  
    outputListBox.Items.Add("Start: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());  
    outputListBox.Items.Add("");  
    string testString = "This is a string that requires memory allocation on the heap";  
    outputListBox.Items.Add("String 1: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());  
    outputListBox.Items.Add("");  
    testString += "Changing the string value results in deallocation of original heap memory, and allocation of new memory";  
    outputListBox.Items.Add("String 2: GC Memory (bytes): " + GC.GetTotalMemory(true).ToString());  
    outputListBox.Items.Add("");  
}  
. . .
```

We display the amount of memory allocated by the **GetTotalMemory** method of the **GC** object. We next declare and initialize **testString**. As a string, **testString** is a reference data type stored on the stack, that references (or points) to memory on the heap where the actual string contents are stored. Your output will show an increase in the managed memory reported by the garbage collector. We next concatenate additional text to our **testString** variable, and as the text indicates, when we change the value of a string, we have to allocate new memory to hold the new string contents, and deallocate the original memory. The content of the **testString** reference value is changed to reference the new memory location, and the previously allocated memory is flagged for eventual deallocation by the garbage collector.

Now, let's add a new class so we can take a look at a destructor. We'll name it "Bag," a term that is frequently used in programming to refer to an object that holds another object.

Right-click the **Destructors** project item in the Solution Explorer, and select **Add | Class....** In the Add New Item dialog box, change the Name from Class1.cs to **Bag.cs**, and then click **Add**.

In the Bag.cs Code Editor, modify the code as shown below.

Bag.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace Destructors
{
    class Bag
    {
        public List<object> Items { get; set; }

        private Stopwatch _stopWatch = new Stopwatch();

        public Bag()
        {
            _stopWatch.Start();
            Console.WriteLine("Bag constructor, start stop watch");
            Items = new List<object>();
        }

        ~Bag()
        {
            _stopWatch.Stop();
            Console.WriteLine("Bag destructor, stop watch stopped, object existed for "
+
                _stopWatch.ElapsedMilliseconds / 1000 + " seconds");
        }
    }
}
```

Click  to save your changes. Before we run it, we'll add code to Destructors.cs to instantiate the Bag.cs class, but first let's discuss the code we've added.

OBSERVE: Bag.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace Destructors
{
    class Bag
    {
        public List<object> Items { get; set; }

        private Stopwatch _stopWatch = new Stopwatch();

        public Bag()
        {
            // Constructor
            _stopWatch.Start();
            Console.WriteLine("Bag constructor, start stopwatch");
            Items = new List<object>();
        }

        ~Bag()
        {
            // Destructor
            _stopWatch.Stop();
            Console.WriteLine("Bag destructor, stopwatch stopped, object existed for "
+
                _stopWatch.ElapsedMilliseconds / 1000 + " seconds");
        }
    }
}
```

We've declared a public **List** automatic property field. By making this field public, we explicitly expose our **List** methods, making it easy to manipulate the **Items** class field. We've also added the **_stopWatch** object of type **Stopwatch** from the **Diagnostics** class, giving us stopwatch timing capability. In the constructor, we start our **_stopWatch**, and create our **Items List** object. When our object is eventually destroyed (when the garbage collector reclaims the memory used by the instance of the Bag class), the Bag destructor is called, and our **_stopWatch** will be stopped. We'll use the **ElapsedMilliseconds** method, dividing by 1000 to convert to seconds, to display how much time elapsed between when the instance of Bag was created, and then reclaimed, by the garbage collector.

Now let's create an instance of the Bag class.

Modify the **Destructors.cs** code as shown below.

Destructors.cs

```

.
.
.
    private Bag _bag = null;

    public Destructors()
    {
        InitializeComponent();
    }

    private void createButton_Click(object sender, EventArgs e)
    {
        // Only create a single Bag.
        if (_bag == null)
        {
            // Create a new Bag instance.
            _bag = new Bag();

            // Create a variable that requires allocating a large block of heap mem
ory.
            byte[] byteArray = new byte[100000];
            for (int i = 0; i < byteArray.Length; i++)
            {
                byteArray[i] = 1;
            }

            // Add the variable to our Bag.
            _bag.Items.Add(byteArray);

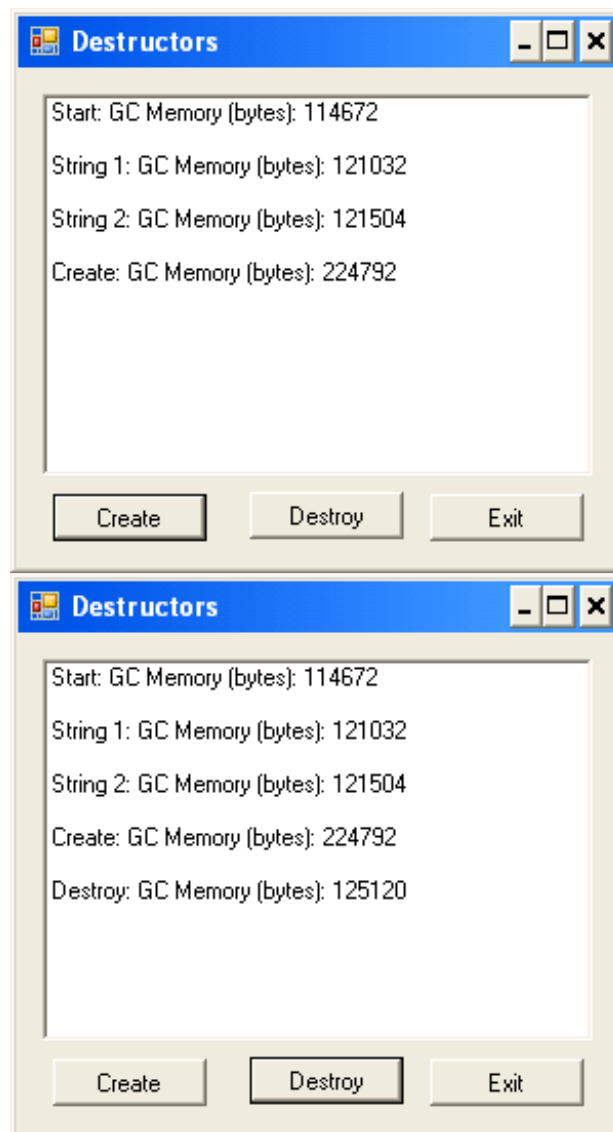
            outputListBox.Items.Add("Create: GC Memory (bytes): " + GC.GetTotalMemo
ry(true).ToString());
            outputListBox.Items.Add("");
        }
    }

    private void destroyButton_Click(object sender, EventArgs e)
    {
        // Make sure Bag has been created.
        if (_bag != null)
        {
            // Remove the reference to our Bag allocated memory.
            _bag = null;

            outputListBox.Items.Add("Destroy: GC Memory (bytes): " + GC.GetTotalMem
ory(true).ToString());
            outputListBox.Items.Add("");
        }
    }
.
.
.

```

Click  and  to save and run the program. Click **Create**, and then click **Destroy**. You should see something like this (again, your memory values may be different):



Let's discuss how this code works.

Destructors.cs

```
.
.
.
private Bag _bag = null;

public Destructors()
{
    InitializeComponent();
}

private void createButton_Click(object sender, EventArgs e)
{
    // Only create a single Bag.
    if (_bag == null)
    {
        // Create a new Bag instance.
        _bag = new Bag();

        // Create a variable that requires allocating a large block of heap memory.
        byte[] byteArray = new byte[100000];
        for (int i = 0; i < byteArray.Length; i++)
        {
            byteArray[i] = 1;
        }

        // Add the variable to our Bag.
        _bag.Items.Add(byteArray);

        outputListBox.Items.Add("Create: GC Memory (bytes): " + GC.GetTotalMemory(true)
        .ToString());
        outputListBox.Items.Add("");
    }
}

private void destroyButton_Click(object sender, EventArgs e)
{
    // Make sure Bag has been created.
    if (_bag != null)
    {
        _bag = null;
        // Remove the reference to our Bag allocated memory.

        outputListBox.Items.Add("Destroy: GC Memory (bytes): " + GC.GetTotalMemory(true)
        .ToString());
        outputListBox.Items.Add("");
    }
}
.
.
.
```

We declare a private class variable `_bag` from our `Bag` class. In our `createButton Click` event handler, we create a byte array variable `byteArray` that allocates a large block of heap memory. After creating the byte array, we add `byteArray` to our `_bag` instance. The allocation of the large block of heap memory is reflected in the amount of managed memory reported by the garbage collector.

In the `destroyButton Click` event handler, we set our `_bag` reference to null. From our earlier discussion, you should note that the `_bag` variable, stored on the stack, no longer references, or points to, the allocated memory. By setting the `_bag` reference variable to null, the garbage collector may reclaim this memory. We report the amount of managed memory, which should have decreased if the garbage collector did reclaim the memory.

So, that's our examination of stack and heap memory, and using a destructor. We'll revisit the destructor later when we deal with unmanaged memory and resources.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click

HAND IT IN at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Overloading Methods, Overloading Operators, and Returning Multiple Values Using out

In this lesson, we'll learn more about how to make methods and operators more versatile with overloading, and we'll return to discuss using the **out** parameter qualifier.

Overloading

Method Overloading

We've previously seen *method overloading*, where we use the same name for multiple class methods within the same class, but each identically-named class method differs in the number of parameters, the parameter data types, or both. We identified that the parameter list of a class method is known as the method's *fingerprint*, or *signature*. Method overloading is useful in maintaining OOP uniformity of name, while supporting different data types. An example of method overloading is shown in the snippet below, where we create a static class for squaring numbers.

Method Overloading
<pre>public static class NumberSquare { public int Square(int value) { return value * value; } public long Square(long value) { return value * value; } public float Square(float value) { return value * value; } public double Square(double value) { return value * value; } }</pre>

With a single method name, **Square**, we can accommodate a variety of data types.

Operator Overloading

As we create new data types using classes, we typically add a number of class methods that represent the actions we can apply to instances of our class. Wouldn't it be more convenient if we could use the C# operators we're familiar with, like the plus sign (+), to work with our new classes? We can, through *operator overloading*.

Operator overloading allows us to create classes that work with C# operators. For a list of operators that may be overloaded, along with a few restrictions, see the MSDN article [Overloadable Operators](#). The syntax of overloading an operator is shown in the code snippet below.

Operator Overloading

```
public class MyNumber
{
    private int _number;

    public MyNumber(int number)
    {
        _number = number;
    }

    // Overload +, a binary operator, requires two parameters.
    public static MyNumber operator +(MyNumber firstParameter, MyNumber secondParameter)
    {
        firstParameter._number += secondParameter._number;
        return firstParameter;
    }

    // Overload !, a unary operator, requires one parameter.
    public static MyNumber operator !(MyNumber firstParameter)
    {
        firstParameter._number *= -1;
        return firstParameter;
    }
}

class TestMyNumber
{
    public TestMyNumber()
    {
        // Create two instances of class.
        MyNumber myNumberOne = new MyNumber(14);
        MyNumber myNumberTwo = new MyNumber(16);

        // Use operators.
        MyNumber myNumberThree = myNumberOne + myNumberTwo;
        myNumberThree = !myNumberThree;

        // The next line will fail.
        myNumberThree += 5;
    }
}
```

In our code snippet, we overloaded the plus (+) and not (!) operators. The plus operator is a binary operator, meaning it requires two parameters, whereas the not operator is a unary operator, only requiring a single parameter. Operator overload methods are always declared as public and static, and use the **operator** keyword before the **operator** to be overloaded. Overloaded operators also must have a **return** statement, although the return data type does not have to be the same type as the class. If you do elect to not return the same data type as the class, you'll need to resolve any type errors that result.

In our operator overload example, for each overloaded operator we have to decide what it means to apply that operator to an instance of our class. You can also see our test class that uses the **MyNumber** class. We also added a line of code that will fail, where we attempt to add the integer value 5 to our class instance. Why does it fail? The integer addition fails because we have not defined an operator overloaded method that has the correct fingerprint of **MyNumber** with an integer.

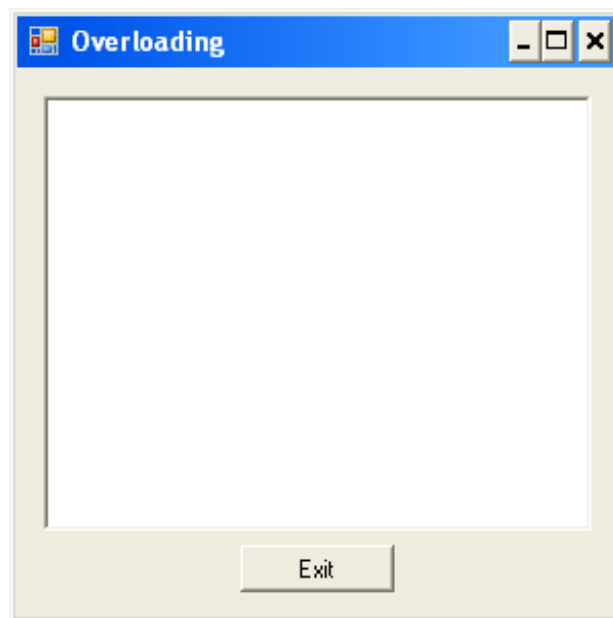
Coding

Now we'll create a Bag class that uses overloaded methods to add integer and string data types. We'll also overload the + operator to support adding one Bag to another Bag.

Select **File | New | Project**. Change the project name to **Overloading** and click **OK**.

Click the entry for **Form1.cs** and change it to **Overloading.cs**. Change the form title bar's Text property in the Properties Window to **Overloading**. Click  to save your changes.

Modify the form to look like this:



Change each control's Name property to match the Name Property column below, and Text property to match the Text Property column. Arrange the controls similar to the image. When you finish, click  to save your changes.

Object	Name Property	Text Property
ListBox	outputListBox	
Button	exitButton	Exit

Double-click the **Exit** Button to automatically generate the default event handler code.

Add the usual exitButton_Click() code to **Overloading.cs** code as shown below:

```
Overloading.cs
.
.
.
    private void exitButton_Click(object sender, EventArgs e)
    {
        this.Close();
    }
.
.
.
```

Add an event handler for the Overloading Form Shown event by selecting the Form (clicking on the Form title bar), selecting the Events icon  to view the Events Properties Window, and double-clicking the **Shown** event.

Now let's add our Bag class.

Right-click on the **Overloading** project item in the **Solution Explorer**, and select **Add | Class....** In the Add New Item dialog box, change the Name from Class1.cs to **Bag.cs**, then click **Add**.

Select the **Bag.cs** Code Editor, and modify the code as shown below.

Bag.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Overloading
{
    class Bag
    {
        // Class variables
        private List<object> _items;

        // Accessors
        public List<object> Items
        {
            get { return _items; }
        }

        // Constructors
        public Bag()
        {
            _items = new List<object>();
        }

        // Overloaded methods
        public void Add(int contents)
        {
            _items.Add(contents);
        }

        public void Add(string contents)
        {
            _items.Add(contents);
        }

        // Overloaded operators
        public static Bag operator +(Bag firstBag, Bag secondBag)
        {
            // Make sure we have data in both of our bags.
            if (firstBag != null && secondBag != null)
            {
                // Add contents from second bag to first bag.
                foreach (object contents in secondBag.Items)
                {
                    firstBag.Items.Add(contents);
                }
            }

            // Return first bag (including added second bag contents).
            return firstBag;
        }
    }
}
```

Click  to save your changes. We'll add code to Overloading.cs shortly to instantiate the Bag.cs class, but first let's discuss this code.

Bag.cs

```
.  
.   
.   
  
    // Overloaded methods  
    public void Add(int contents)  
    {  
        _items.Add(contents);  
    }  
  
    public void Add(string contents)  
    {  
        _items.Add(contents);  
    }  
  
    // Overloaded operators  
    public static Bag operator +(Bag firstBag, Bag secondBag)  
    {  
        // Make sure we have data in both of our bags.  
        if (firstBag != null && secondBag != null)  
        {  
            // Add contents from second bag to first bag.  
            foreach (object contents in secondBag.Items)  
            {  
                firstBag.Items.Add(contents);  
            }  
        }  
  
        // Return first bag (including added second bag contents).  
        return firstBag;  
    }  
  
.   
.   
. 
```

As you can see, we added two overloaded **Add** methods, one that accepts an integer parameter, and one that accepts a string. Then, we overloaded the plus (+) operator to support adding the contents of one **Bag** (**secondBag**) to another **Bag** (**firstBag**), and return the expanded **Bag**. Let's now use our **Bag** class by creating an instance of the class.

Modify the **Overloading.cs** code as shown below.

Overloading.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Overloading
{
    public partial class Overloading : Form
    {
        private Bag _myBag = new Bag();

        public Overloading()
        {
            InitializeComponent();
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void Overloading_Shown(object sender, EventArgs e)
        {
            // Declare some content data.
            string lastName = "Lastname";
            int age = 21;

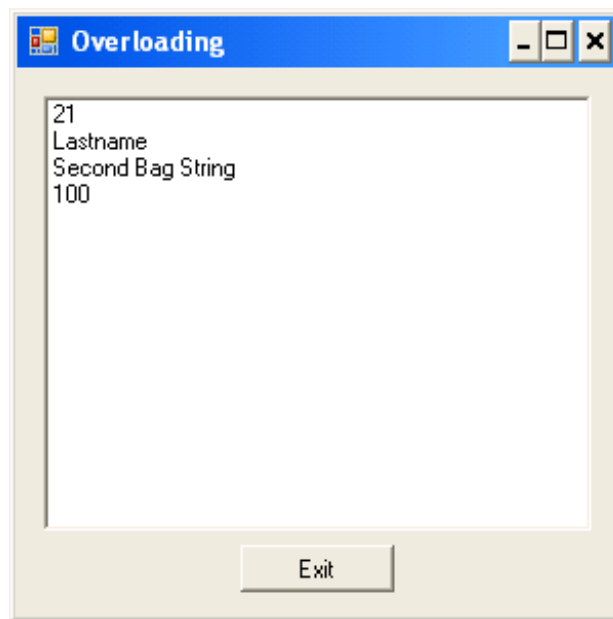
            // Add the content data to our permanent bag.
            _myBag.Add(age);
            _myBag.Add(lastName);

            // Create a temporary bag, and add string and integer literal contents.
            Bag _temporaryBag = new Bag();
            _temporaryBag.Add("Second Bag String");
            _temporaryBag.Add(100);

            // Add the contents of our temporary bag to our permanent bag.
            _myBag += _temporaryBag;

            // Output the contents of our permanent bag to the ListBox.
            foreach (object contents in _myBag.Items)
            {
                outputListBox.Items.Add(contents.ToString());
            }
        }
    }
}
```

Click  and  to save your changes and run the program. You should see this:



Let's discuss how this code works.

Overloading.cs

```
.
.
.
public partial class Overloading : Form
{
    private Bag _myBag = new Bag();

    public Overloading()
    {
        InitializeComponent();
    }

    private void Overloading_Shown(object sender, EventArgs e)
    {
        // Declare some content data.
        string lastName = "Lastname";
        int age = 21;

        // Add the content data to our permanent bag.
        _myBag.Add(age);
        _myBag.Add(lastName);

        // Create a temporary bag, and add string and integer literal contents.
        Bag _temporaryBag = new Bag();
        _temporaryBag.Add("Second Bag String");
        _temporaryBag.Add(100);

        // Add the contents of our temporary bag to our permanent bag.
        _myBag += _temporaryBag;

        // Output the contents of our permanent bag to the ListBox.
        foreach (object contents in _myBag.Items)
        {
            outputListBox.Items.Add(contents.ToString());
        }
    }
}
.
.
.
```

We add a private **Bag** variable, **_myBag**, and use the overloaded **Add** methods to add an integer and a string to

`_myBag`. We create a temporary `Bag`, `_temporaryBag`, again adding a string and an integer. We use the overloaded `+` operator to add our `_temporaryBag` to our permanent `_myBag`. Finally, we loop through the contents of `_myBag`, displaying using our `ListBox` that the contents of both `Bags` were indeed added in `_myBag`.

The out Keyword

Parameter Modifier Review

We've seen that method parameters may have parameter modifiers, such as `ref`, `out`, and `params`. We've discussed that the `ref` keyword is used to modify a method parameter to pass a reference to the parameter argument, rather than a copy of the parameter. We've also discussed the difference between value data types and reference data types, where the `ref` parameter modifier is required if you want to modify the contents of a parameter and have that change persist in the calling code. We've discussed that reference data types do not require a parameter modifier to have the changes reflected in the calling code, as reference data types by default pass a reference to the allocated memory. And finally, we've discussed how adding the `ref` parameter modifier to a reference value type would allow us to change the referenced memory, effectively reallocating new memory, and have that new memory reference returned to the calling code. If any of these concepts are vague, feel free to return to the appropriate lessons and review the material. You can also glance at the summary table below. We'll explain more about the table shortly, after we introduce the `out` keyword.

The out Parameter Modifier

The `out` keyword works somewhat like `ref`, in that it forces a pass by reference, returning any changes to the calling code. However, `out` requires that the parameter be assigned a value within the called code. What does that mean? It means that if you pass in a value data type using `out`, you must assign a value to the parameter. If you pass in a reference data type using `out`, you must also assign a value, but for a reference data type; that means you have to create a reference to a memory location, such as when you use the `new` keyword.

Effectively, `ref` requires initialization before usage, allowing you to preserve any original data unless you explicitly change it in the called code; `out` requires value assignment in the called code, removing any prior data that may have been assigned. The table below summarizes the differences.

Type of Data	No Parameter Modifier	ref Parameter Modifier	out Parameter Modifier
Value Data Types	No changes persist in calling code.	Changes persist in calling code, prior data value may remain if not changed, no memory reassignment as value stored on stack.	Changes persist in calling code, prior value will be lost, value assignment mandatory, no memory reassignment as value stored on stack.
Reference Data Types	Changes persist in calling code, prior referenced data values may remain if not changed, no memory reassignment allowed.	Changes persist in calling code, prior referenced data values may remain if not changed, memory reassignment allowed.	Changes persist in calling code, all prior referenced data values lost, memory reassignment mandatory.

Method Fingerprint/Signature and Parameter Modifiers

We do need to emphasize that using a parameter modifier impacts the fingerprint or signature of a method. Typically, an overloaded method must differ by the data type of the parameters, and by the number of parameters. A method signature also will differ based on the use of a parameter modifier. In the code snippet below, the two methods are legitimately different methods because of the use, or non-use, of parameter modifiers.

Method Fingerprint/Signature and Parameter Modifiers

```
public static class NumberSquare
{
    public int Square(int value)
    {
        return value * value;
    }

    public void Square(ref int value)
    {
        value *= value;
    }
}
```

Note

Although method signatures may differ based on a parameter modifier, if the only difference between a method signature is **ref** or **out**, the method signatures will be considered the same and will produce an error. The code snippet below will produce an error.

Fingerprints Must Differ By More Than Parameter Modifier

```
public static class NumberSquare
{
    // This code snippet will produce an error as the overloaded method Square s
    // signature
    // must differ by more than ref and out.
    public void Square(ref int value)
    {
        value *= value;
    }

    public void Square(out int value)
    {
        value = 4;
    }
}
```

Multiple Return Values Using out

As you know, a method can only have a single return value. One practical use of **out** is to produce the equivalent of multiple return values. We've seen this technique applied before when we use the **TryParse** method, as shown in the code snippet below. **TryParse** will return a boolean value to indicate if the conversion was successful or not, and uses the **out** parameter modifier to return the converted value.

Multiple Return Values Using out

```
public class Multiple
{
    public Multiple()
    {
        int wages;

        if (!int.TryParse("353.26", out wages))
        {
            Console.WriteLine("Error converting value");
        }
    }
}
```

Coding

To illustrate the usage of the **out** parameter modifier, let's modify our Overloading project, adding a Clone method to the Bag class that uses the **out** parameter modifier, and creating a Clone of our permanent Bag object.

Modify the **Bag.cs** code as shown below.

```
Bag.cs

.
.
.

    // Constructors
    public Bag()
    {
        _items = new List<object>();
    }

    // Public methods
    public int Clone(out Bag resultBag)
    {
        // Create our new out Bag.
        resultBag = new Bag();

        // Copy the contents to our new out Bag.
        foreach (object contents in _items)
        {
            resultBag.Items.Add(contents);
        }

        // Return the count of items in our new out Bag.
        return resultBag.Items.Count;
    }

    // Overloaded methods
.
.
.
```

Click  to save your changes. Let's discuss how this code works.

```
Bag.cs

.
.
.

    // Public methods
    public int Clone(out Bag resultBag)
    {
        // Create our new out Bag.
        resultBag = new Bag();

        // Copy the contents to our new out Bag.
        foreach (object contents in _items)
        {
            resultBag.Items.Add(contents);
        }

        // Return the count of items in our new out Bag.
        return resultBag.Items.Count;
    }

.
.
.
```

Our **Clone** method creates a new **Bag** instance, using the **resultBag** parameter, iterating through the contents of the **_items** List, adding each item in the List to our new **resultBag**. Finally, we return the number of items in the new **Bag resultBag** object.

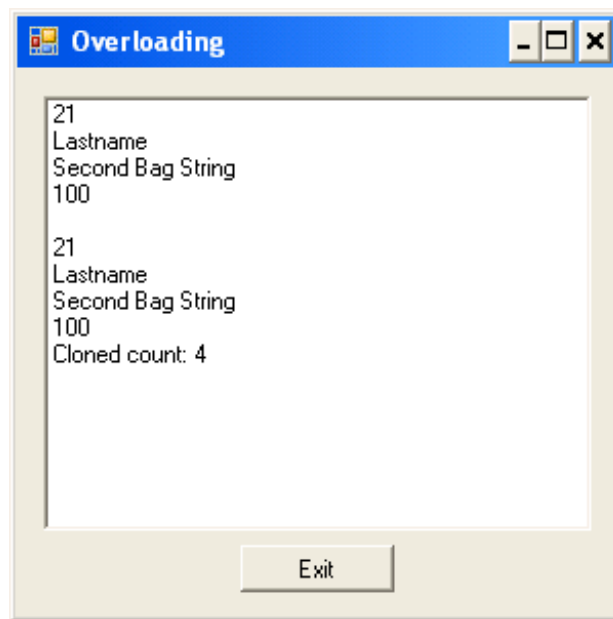
Now, let's modify Overloading.cs to use the new Clone method.

Modify the **Overloading.cs** code as shown below.

Overloading.cs

```
.  
.   
.   
  
private void Overloading_Shown(object sender, EventArgs e)  
{  
    // Declare some content data.  
    string lastName = "Lastname";  
    int age = 21;  
  
    // Add the content data to our permanent bag.  
    _myBag.Add(age);  
    _myBag.Add(lastName);  
  
    // Create a temporary bag, and add string and integer literal contents.  
    Bag _temporaryBag = new Bag();  
    _temporaryBag.Add("Second Bag String");  
    _temporaryBag.Add(100);  
  
    // Add the contents of our temporary bag to our permanent bag.  
    _myBag += _temporaryBag;  
  
    // Output the contents of our permanent bag to the ListBox.  
    foreach (object contents in _myBag.Items)  
    {  
        outputListBox.Items.Add(contents.ToString());  
    }  
  
    // Create a clone of our permanent bag.  
    Bag clonedBag;  
    int itemsInClone = _myBag.Clone(out clonedBag);  
  
    // Output the contents of the cloned bag to the ListBox.  
    outputListBox.Items.Add("");  
    foreach (object contents in clonedBag.Items)  
    {  
        outputListBox.Items.Add(contents.ToString());  
    }  
  
    // Display count of items in cloned bag to the ListBox.  
    outputListBox.Items.Add("Cloned count: " + itemsInClone);  
}  
  
.   
.   
. 
```

Click  and  to save your changes and run the program. You should see the following output:



Let's discuss how this code works.

```
Overloading.cs
.
.
.
    // Create a clone of our permanent bag.
    Bag clonedBag;
    int itemsInClone = _myBag.Clone(out clonedBag);

    // Output the contents of the cloned bag to the ListBox.
    outputListBox.Items.Add("");
    foreach (object contents in clonedBag.Items)
    {
        outputListBox.Items.Add(contents.ToString());
    }

    // Display count of items in cloned bag to the ListBox.
    outputListBox.Items.Add("Cloned count: " + itemsInClone);
.
.
.
```

We declare a **Bag** variable called **clonedBag**. Notice that we don't instantiate the object. We could, but the instance would be destroyed when we pass the **clonedBag** variable into the **Clone** method using the **out** parameter modifier. The **Clone** method will create a new instance of the **Bag** class, copying the contents of **_items** into our new **clonedBag** object, and returning the count of items in the new **clonedBag** object saved in **itemsInClone**. We output the contents of **clonedBag**, and the **itemsInClone**, showing that the cloning worked.

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Debugging

In this lesson, we'll dig deeper into how to debug our programs, and discuss the difference between compile-time and runtime errors. We'll also cover how to output debug statements, use debug breakpoints, evaluate variables at runtime, trace our code using the call stack, and so much more.

This lesson includes the following sections:

- [A Sample Debugging Program](#)
- [Breakpoints, Expression Evaluation, and Watches](#)

Throughout the lessons, you should already have begun to take advantage of the debugging features of Visual Studio to run and debug your programs. The ability to use a visual debugging tool is one of the primary features of modern integrated development environments (IDEs). We introduced some of the basic debugging techniques earlier, and we will review them again as we add to the list of debugging features you can use in Studio.

Errors may exist in our software at different levels. Errors that prevent the program from compiling, which therefore prevent the program from running, are called *compile-time errors*. Even if your program compiles, it still may not run correctly. Errors that occur when your program runs are called *runtime errors*. Finally, if your program compiles and runs, but the results you get from your program are incorrect, the cause may be *logic errors*. You've undoubtedly seen compile-time errors before, unless you've managed to make no mistakes in your code. If so, congratulations! When you encounter a compile-time error, Studio presents you with an alert that helps find the line of code that contains the error, and hints as to what is wrong and possibly how to resolve the error. For this lesson, we'll focus on using the Studio debugging features at runtime. These features help resolve runtime and logic errors.

As the word implies, debugging is removing "bugs," or defects, from your software, but with Studio, you can use the same tools you would use for debugging to learn more about the C# language, and how your code behaves. The Studio debugging tools enable you to step through your code, evaluate variables, and so much more.

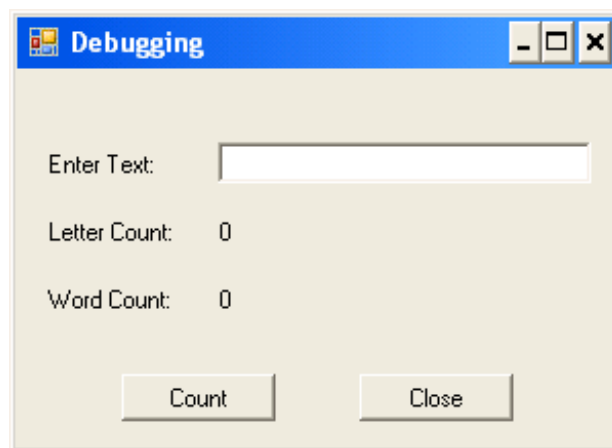
A Sample Debugging Program

For this lesson, we'll use a simple program to explore the debugging features of Studio. Let's get started.

Select **File | New | Project**, change the project Name to **Debugging**, and click **OK**. Locate the entry for Form1.cs and change it to **Debugging.cs**. Click the Form1 Form title bar, find the Text property in the Properties Window, and change it to **Debugging**. Click  to save your changes.

Now let's add the controls we'll need.

Modify the Debugging Form by adding the controls listed below. Change each control's Name property to match the Name Property column, and each Text property to match the Text Property column in the table. Arrange the controls like in the image below.



Object	Name Property	Text Property
Label	enterTextLabel	Enter Text:
TextBox	dataTextBox	

Label	letterCountLabel	Letter Count:
Label	letterCountValueLabel	0
Label	wordCountLabel	Word Count:
Label	wordCountValueLabel	0
Button	countButton	Count
Button	closeButton	Close

Click  to save your changes. Next, let's add all of the event handlers we'll need.

Double-click each Button control to add the click events. Click .

Control	Event	Event Name
countButton	Click	countButton_Click
closeButton	Click	closeButton_Click

Next, let's add a couple of methods and code to the event handlers.

Select the Debugging.cs Form Code Editor and modify the code as shown below.

Debugging.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Text.RegularExpressions;

namespace Debugging
{
    public partial class Debugging : Form
    {
        public Debugging()
        {
            InitializeComponent();
        }

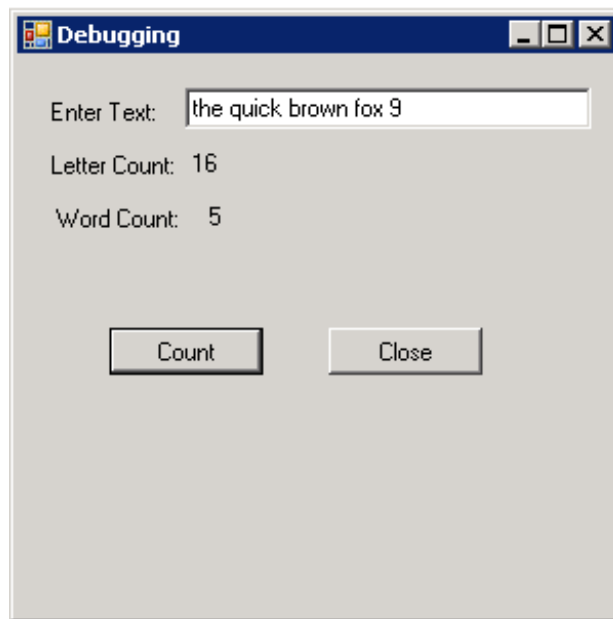
        private void countButton_Click(object sender, EventArgs e)
        {
            letterCountValueLabel.Text = countLetters(dataTextBox.Text).ToString();
            wordCountValueLabel.Text = countWords(dataTextBox.Text).ToString();
        }

        private void closeButton_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private int countWords(string textData)
        {
            int wordCount = 0;
            wordCount = Regex.Matches(textData, @"[A-Za-z0-9]+").Count;
            return wordCount;
        }

        private int countLetters(string textData)
        {
            int letterCount = 0;
            letterCount = Regex.Matches(textData, @"[A-Za-z]").Count;
            return letterCount;
        }
    }
}
```

Click  and  to save and run the program. Enter a sentence of a few words, and include some numerals, such as "the quick brown fox 9," and click **Count**. You should see something like this:



Let's discuss the code.

OBSERVE: Debugging.cs

```
using System;
.
.
.
using System.Text.RegularExpressions;

namespace Debugging
{
    public partial class Debugging : Form
    {
        .
        .
        .
        private void countButton_Click(object sender, EventArgs e)
        {
            letterCountValueLabel.Text = countLetters(dataTextBox.Text).ToString();
            wordCountValueLabel.Text = countWords(dataTextBox.Text).ToString();
        }
        .
        .
        .
        private int countWords(string textData)
        {
            int wordCount = 0;
            wordCount = Regex.Matches(textData, @"[A-Za-z0-9]+").Count;
            return wordCount;
        }

        private int countLetters(string textData)
        {
            int letterCount = 0;
            letterCount = Regex.Matches(textData, @"[A-Za-z]").Count;
            return letterCount;
        }
    }
}
```

This sample program counts the number of letters and words in a user-entered text string. Letters include any letter in the English alphabet, whether uppercase or lowercase. When you clicked the **Count** Button, the `countButton_Click` event handler "fired," calling the `countLetters` and `countWords` methods. Each of these methods returns an int value that we use to set the appropriate Label.

We introduced the **Regex** class that requires the new **using** statement. **Regex** supports regular expressions. A regular expression is a pattern-finding and replacing notation that's been around for many years. You can search online for more information on how to create a regular expression. Microsoft also provides a nice overview at [Regular Expressions](#). We'll use **Regex** to remind you of a handy debugging feature you should have seen before: Intellisense Quick Info. With Quick Info, you can hover your mouse pointer over any C# identifier and see additional information. Below, we've hovered over **Regex**:

```
private int countWords(string textData)
{
    int wordCount = 0;

    wordCount = Regex.Matches(textData, @"[A-Za-z0-9]+").Count;

    return wordCount;
}
```

class System.Text.RegularExpressions.Regex
Represents an immutable regular expression.

Note that we're using the class name **Regex** without creating an object instance, so **Regex** must be a *static* class.

You may also have noticed the **@** symbol before the second **Regex** parameter, which is the actual regular expression. The **@** symbol when used before string literals effectively "escapes" the text within the string literal. Escaping is using a special symbol before a character to change the meaning of the character. We've seen special escape characters already: **\n**, for carriage return; and perhaps **\t**, for horizontal tab. The **** character is the escape character in C#, but it is also used in regular expressions, and even in Windows file path names. If we want to use the **** as a backslash, and not get an error, we would have to put two back slashes, ****, to get the single slash we want—effectively, escaping the escape. Or, we can place the **@** symbol before the string literal.

Note The **@** symbol is used for other purposes in C# that we'll cover as those topics are introduced in the lessons.

Breakpoints, Expression Evaluation, and Watches

In the Visual Code Editor window, we can add one or more breakpoints to our code. A breakpoint is exactly what the name implies: a point in your source code where execution will stop, or break. Let's add a breakpoint, then execute our code in debug mode and stop at the breakpoint.

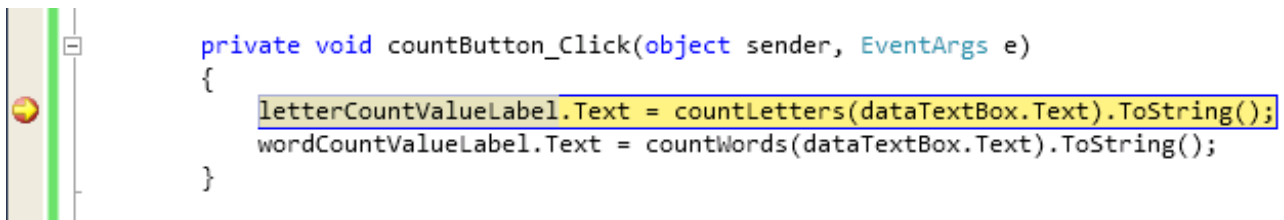
Select the Debugging.cs Form Code Editor, and scroll the window so that you can see the **countButton_Click** event handler. Move your mouse pointer to the left of the line that sets the **letterCountValueLabel.Text**, which is the first line in the event handler. Keep moving your mouse to the left of the outline lines, and the change bars, until you get to a vertical section that spans the entire Code Editor. Click in this area until you get a red circle that is parallel to the **letterCountValueLabel.Text** line of code, and highlights the line of code in red. (If you place the red circle next to the wrong line, click the red circle again to remove it.)



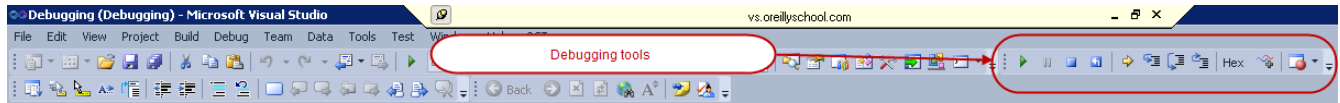
```
private void countButton_Click(object sender, EventArgs e)
{
    letterCountValueLabel.Text = countLetters(dataTextBox.Text).ToString();
    wordCountValueLabel.Text = countWords(dataTextBox.Text).ToString();
}
```

Note When you run a program, this lesson text disappears while the program takes control of Visual Studio. The lesson text will reappear when the program ends. However, when the program pauses at your breakpoint, you won't be able to see the lesson text. To bring back the lesson text, select **Windows | C# Part 2**. Then, right-click the **C# Part 2** tab and select **Move to previous tab group**.

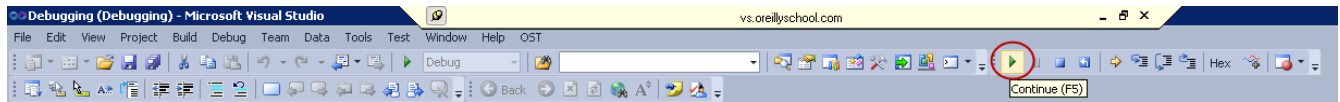
Click the Start Debugging button  to run the program. Type **My dog is brown** and click **Count**. The program stops executing, with Studio highlighting the breakpoint in yellow:



While the program is paused at a breakpoint, several debugging tools are added to the toolbar:



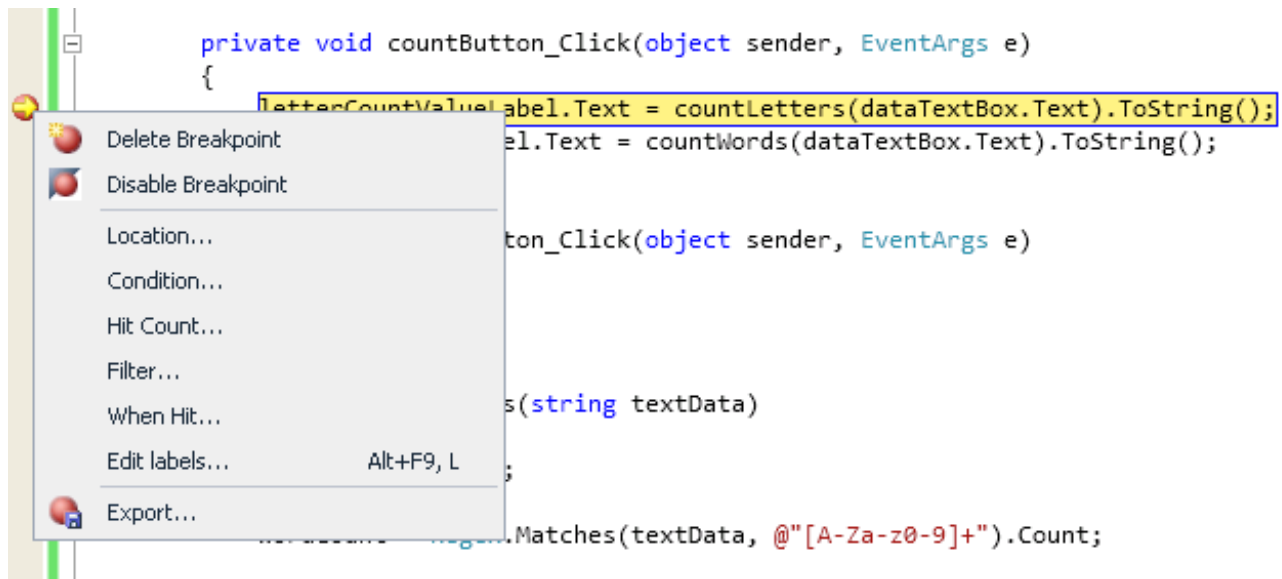
Especially note the Continue button:



Click that button when you're ready to continue running the program.

We're not limited to simply pausing our code whenever a breakpoint is encountered. Let's get a list of breakpoint options.

With our code paused at the first breakpoint, right-click the red breakpoint circle:



As you can see, we can delete the breakpoint, which can also be accomplished by simply clicking the red breakpoint circle again, as the breakpoints are effectively toggles. We can also disable a breakpoint, which is convenient in keeping a breakpoint for later use, but skipping it when it isn't needed. The other breakpoint options each bring up a dialog box. Each of these breakpoint options are explained below.

Note When you save a file in Studio, you also save the breakpoint information.

Breakpoint Option	Dialog	Description
Location	File Breakpoint	Provides exact information about breakpoint.
Condition	Breakpoint Condition	Set breakpoint trigger conditions.
Hit Count	Breakpoint Hit Count	Set breakpoint trigger hit criteria.
Filter	Breakpoint Filter	Set advanced breakpoint trigger conditions.
When Hit	When Breakpoint Is Hit	Set advanced behavior when breakpoint is triggered.
Edit Labels	Edit Breakpoint Labels	Set a label (name) to a breakpoint.

With small programs, most of these advanced breakpoint options are unnecessary, but with larger programs, it's very common to set advanced breakpoint options to only break under certain circumstances. Feel free to experiment with any of these advanced options, such as the hit count option.

With the program paused at this first breakpoint, we can perform a number of useful debugging, and learning, tasks, such as variable, field, and property evaluation. Let's try it!

Hover over the **dataTextBox.Text** identifier in the yellow highlighted code. Make sure you place the mouse pointer over the **Text** property:

```

's(dataTextBox.Text).ToString();
ataTextBox.Text).

```

dataTextBox.Text "My dog is brown."

That's pretty useful to be able to hover over a property and see the value. But you're not limited to just properties; you can also view the TextBox object.

Hover over the **dataTextBox** identifier in the yellow highlighted code. Make sure you place the mouse pointer over the **dataTextBox** object:

```

private void countButton_Click(object sender, EventArgs e)
{
    letterCountValueLabel.Text = countLetters(dataTextBox.Text).ToString();
    wordCountValueLabel.Text = countWords(dataText
}

```

dataTextBox {Text = "My dog is brown."}

You'll notice that with the TextBox object, you have an expansion control, the plus symbol, to expand the popup to display other information about the identifier you've selected.

Hover over the **dataTextBox** identifier in the yellow highlighted code. Make sure you place the mouse pointer over the **dataTextBox** object. Click the expansion plus symbol to see expanded information:

```

ataTextBox.Text).ToString();
ex dataTextBox {Text = "My dog is brown."}

```

base {System.Windows.Forms.TextBoxBase}	{Text = "My dog is brown."}
acceptsReturn	false
AcceptsReturn	false
AutoCompleteCustomSource	{System.Windows.Forms.AutoCompleteStringCollection}
autoCompleteCustomSource	{System.Windows.Forms.AutoCompleteStringCollection}
autoCompleteMode	None
AutoCompleteMode	None
AutoCompleteSource	None
autoCompleteSource	None
characterCasing	Normal
CharacterCasing	Normal
CreateParams	{CreateParams {'EDIT', "", 0x560100c0, 0x200, {98, 35, 182, 20}}}
fromHandleCreate	false
Multiline	false
PasswordChar	0 '\0'

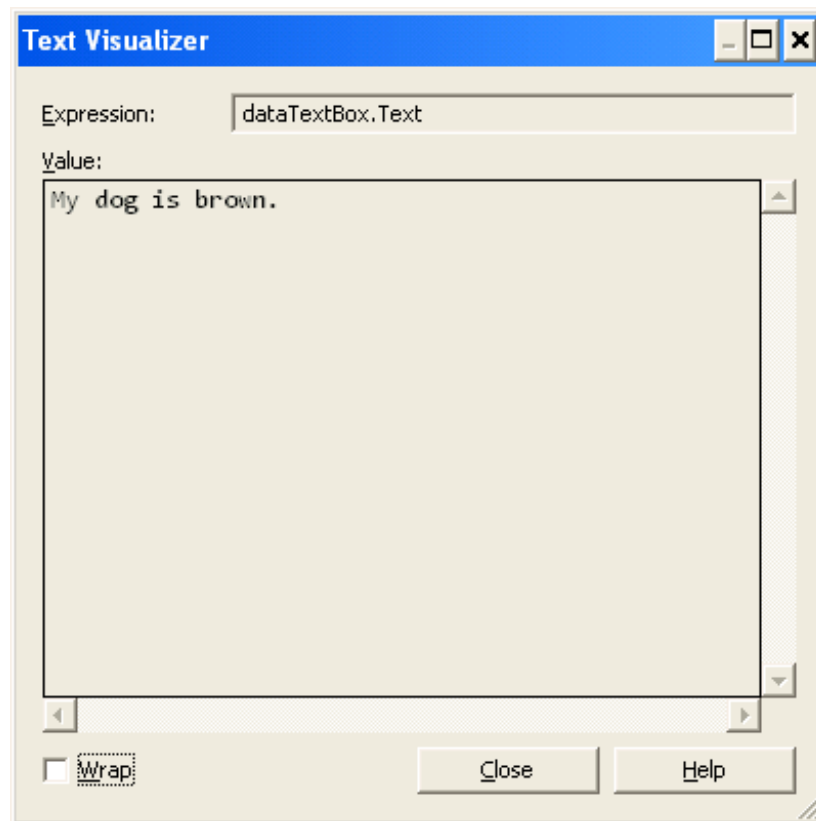
We're mostly interested in the **Text** property of the TextBox, but in the expanded view, you can see all of the fields and properties for the TextBox object. You should note that at the bottom of the expanded view is a black down arrow that serves as a scrolling arrow to see more of the fields or properties.

Note

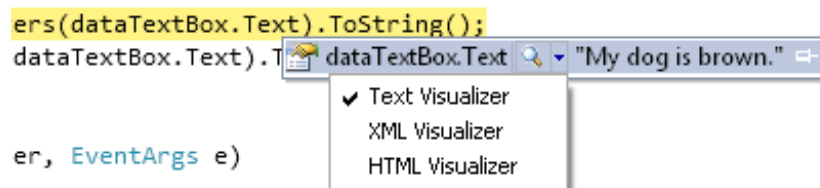
If you do not see the plus character to view the expanded information box, you probably hovered over just the **Text** identifier. Intellisense is context-sensitive, showing the most limited information it can provide based on what you're hovering over. To see the expanded view option, you have to hover over the **TextBox** identifier, **dataTextBox**.

In the popup that appears when you hover over an identifier, you'll notice that there is a magnifying glass button, and a dropdown arrow next to the magnifying glass. The magnifying glass, when clicked, will display one of three data Visualizers: a Text Visualizer, an XML Visualizer, or an HTML Visualizer. By default, the Text Visualizer is selected, and will be displayed. A Visualizer is a very convenient tool for seeing data in a different format, or when the data is lengthy. Let's bring up the Text Visualizer.

Hover over the **dataTextBox** identifier in the yellow highlighted code. Make sure you place the mouse pointer over the **dataTextBox** object. Click the magnifying glass symbol to see the Text Visualizer:

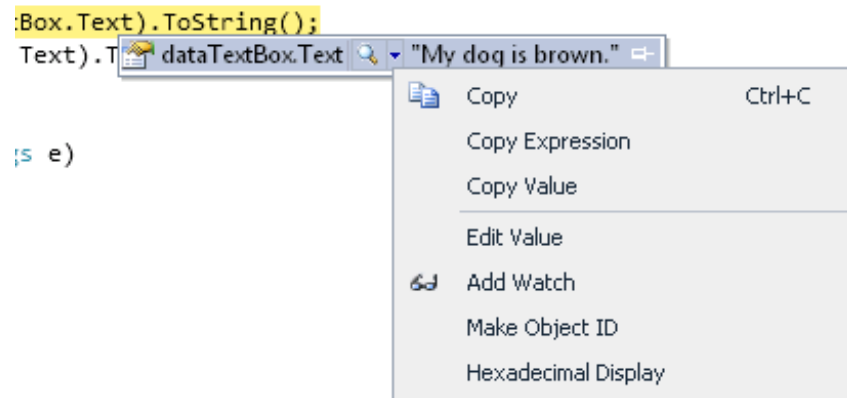


Click **Close**. You can change the selected Visualizer by clicking the dropdown arrow, and selecting the desired Visualizer:



You still have more options. You can also right-click on the popup, and see a menu option to copy or edit the value of the identifier you're evaluating, and a number of other options. Let's bring up the popup menu first.

Hover over the **dataTextBox.Text** identifier in the yellow highlighted code. Make sure you place the mouse pointer over the **Text** property. Right-click the popup that appears to bring up a new popup menu.



Below are descriptions of the Copy options.

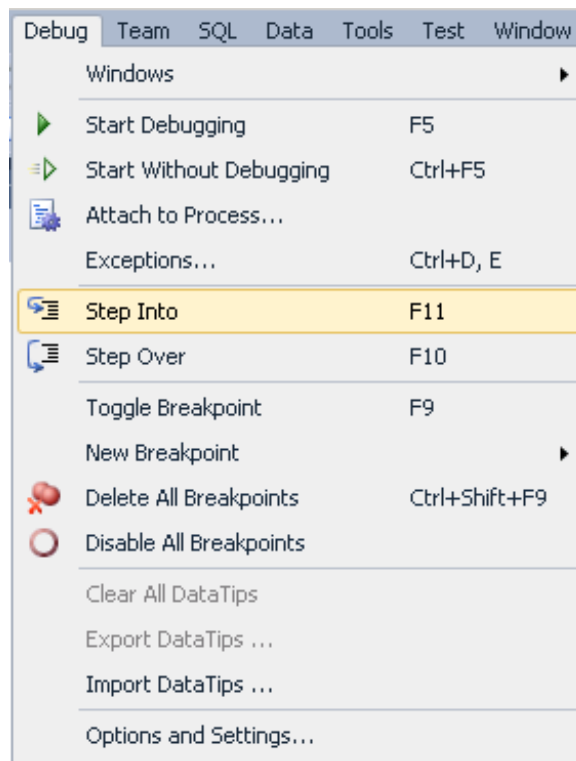
Copy Choice	Value
Copy	<code>dataTextBox.Text = "My dog is brown."</code>
Copy Expression	<code>dataTextBox.Text</code>
Copy Value	<code>"My dog is brown."</code>

Another option from the popup menu is **Add Watch**, which will be our next topic, after we explore moving forward in our code. Once stopped at a breakpoint, when you're ready to move forward, you have a number of options; the most common ones are:

Move Option	Description
Step Over	Execute the current statement, but do not step into any code that might be accessed.
Step Into	Execute the current statement, stepping into any code that is accessed.
Run To Cursor	Place mouse pointer on another line of code, and execute until that line of code is encountered.
Continue	Resume normal execution, breaking at next breakpoint, if any.
Stop	Stop executing the program.

You should still be at the first break point. Let's use the **Step Into** option.

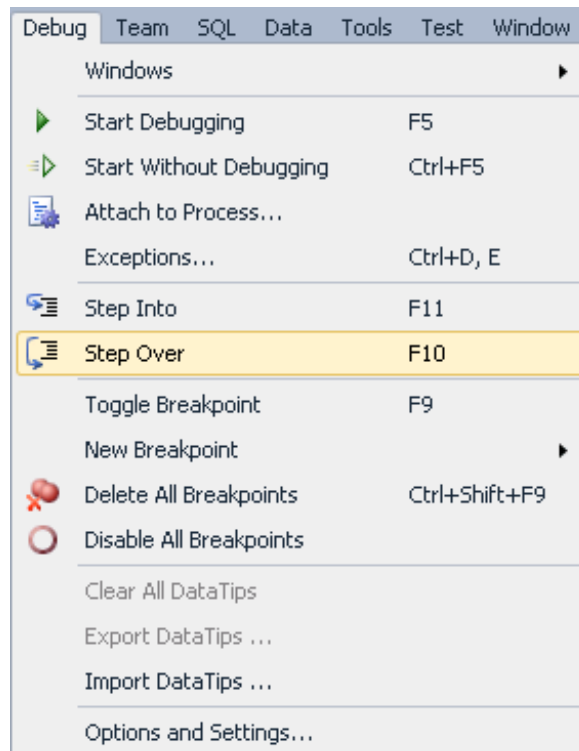
Select **Debug | Step Into**, or locate and click the Step Into button on the Studio toolbar.

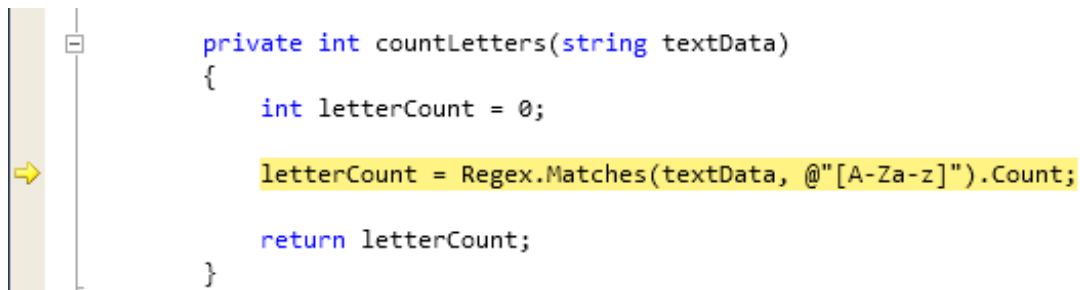


Note The hotkeys in Visual Studio may change.

After selecting the **Step Into** debug option, you are in the **countLetters** method. Once you've stepped into the code you want to examine, you typically switch to using the **Step Over** method, to prevent stepping into other code, unless you want to continue to step further and further into your code. In our case, the **countLetters** method is the final level we want to step into.

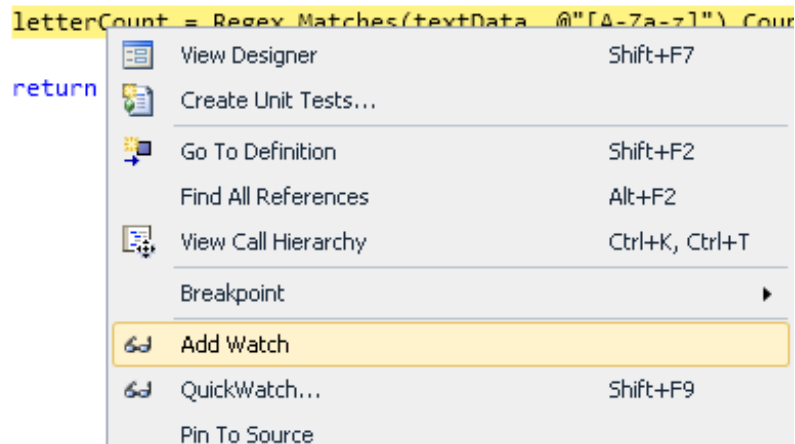
Select **Debug | Step Over**, or find and click the Step Over button on the Studio toolbar. Continue to select the **Step Over** option until Studio highlights in yellow the line of code that contains the **Regex** code.



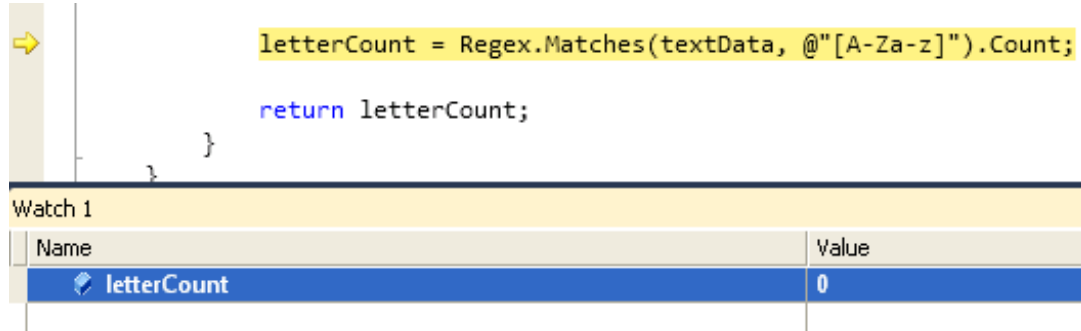


Now, let's add the Watch on **letterCount**. Rather than use the method we've used before to add a Watch variable, we'll use a much simpler method.

Right-click on the **letterCount** variable in the **countLetters** method. From the popup menu, select **Add Watch**.

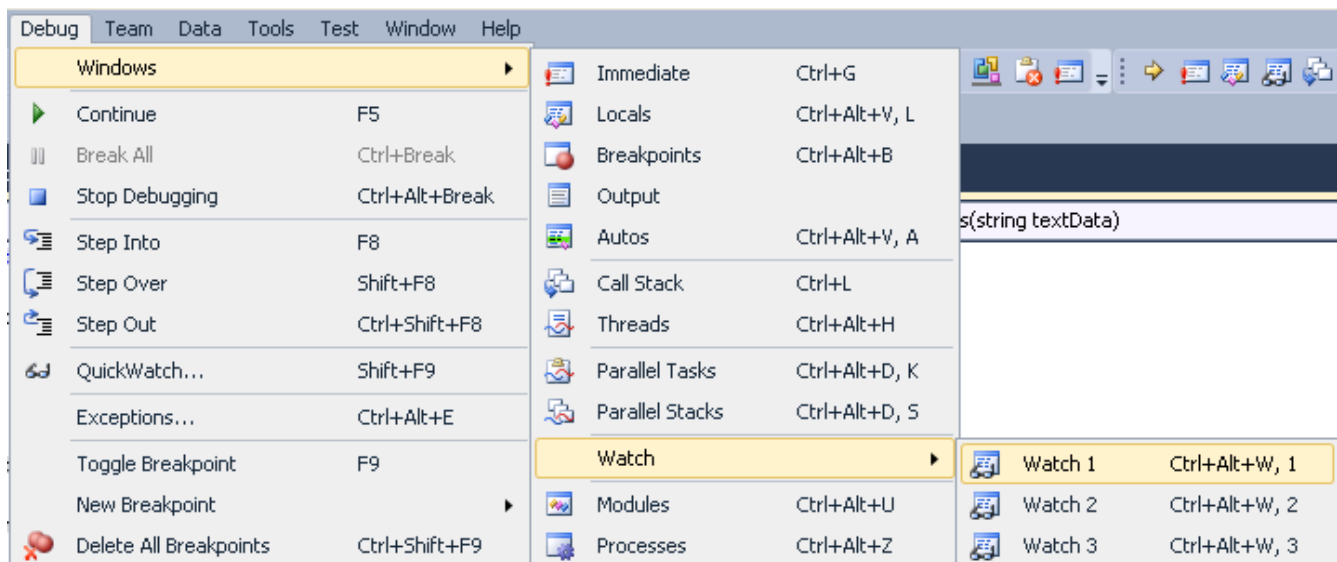


Once you've added the Watch, you should see the **Watch 1** tab, with the **letterCount** variable.



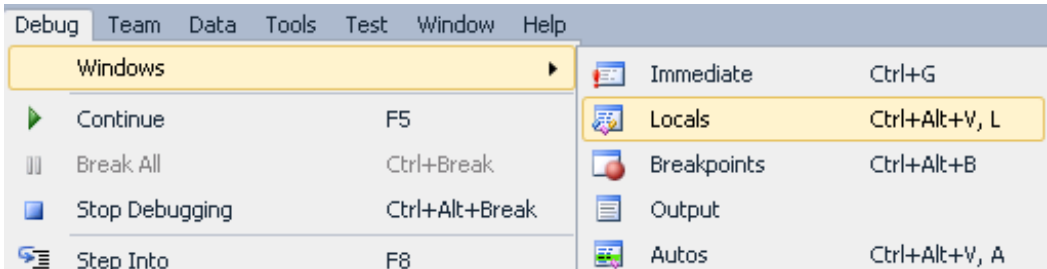
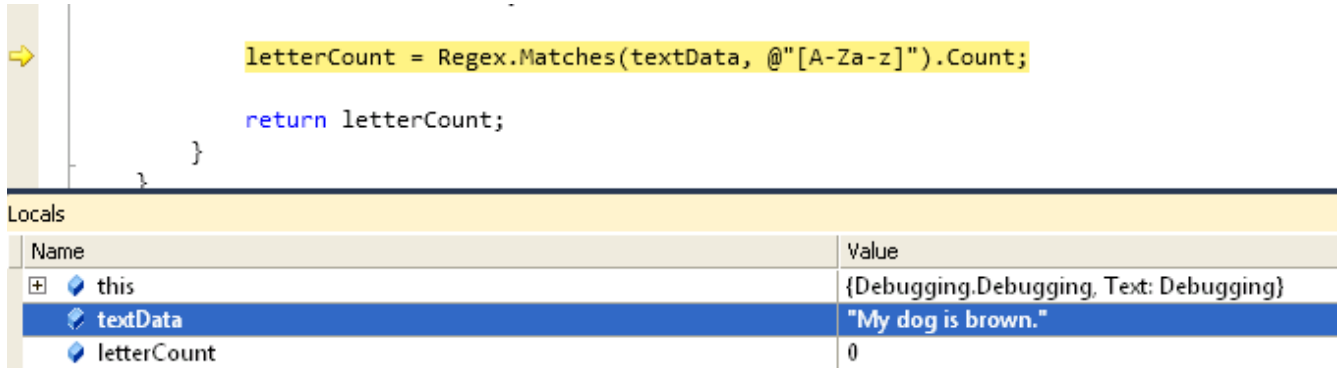
Note

If for some reason you're unable to see the Watch 1 tab, you may have to make it visible using **Debug | Windows | Watch | Watch 1**.



While you can add as many Watch items as you'd like, Studio already has an easy way to see variables that are local to the current scope in the **Locals Window**.

Locate and select the **Locals** tab (if the Locals tab is not visible, select **Debug | Windows | Locals**).



We've looked at the Locals and Watch 1 tab, so you might want to look at what other tabs are available while you're debugging a C# program. Some of the tabs aren't appropriate to discuss in this lesson, but there are three other tabs you will find pretty useful, so you should be familiar with them.

The first is the **Output** tab, which we've used before. Whenever your program generates **Console** output, such as with a **System.Console.WriteLine** statement, the output appears in the **Output** tab.

The second useful tab is the **Call Stack** tab. This tab displays the stack path the code has followed to get to the current line. Only those lines of code that called to a different location in the code that jumped into a new scope, such as with a method call, are shown on the **Call Stack** tab. You can double-click any entry, and you'll be taken to that location in the code.

Note

You will probably notice that certain entries in the **Call Stack** appear light grey, or greyed out. These lines of code represent calls into other programming code, such as the .NET framework, that are not code within your project. You can try to jump to these lines of code, but unless you have the source code installed, you will receive a message that no source code is available.

The third useful tab is the **Immediate** tab. This tab is a very interesting tab, as you can type in C# code to evaluate

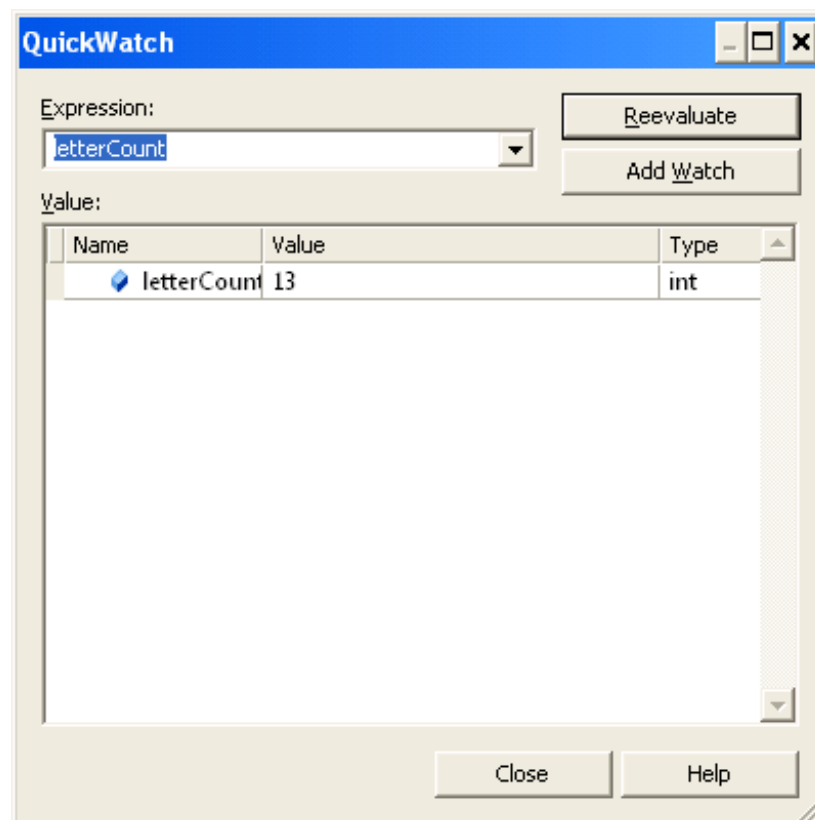
expressions, display the results of expression results, and even change the value of identifiers. To display identifier or expression results, the Immediate window supports a shortcut symbol, the question mark (?), for output. Here are some examples of what you can do in the Immediate Window:

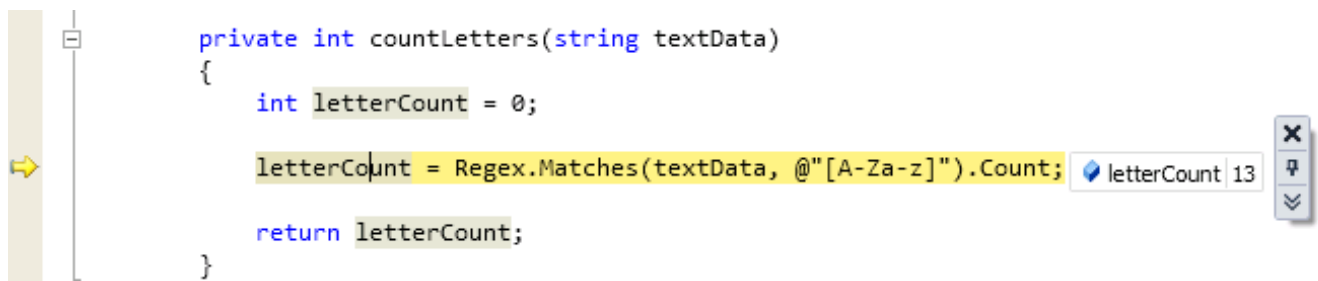
```
Immediate Window
? letterCount
0
letterCount = 13
13
? letterCount
13
? letterCount + 3
16
? letterCount
13
```

From the sample output, you can see we display the current value of letterCount, then change the value of letterCount to 13, and confirm the value change by again displaying the current value, then displaying the result of adding 3 to the value of letterCount, and then displaying the current value, which was unchanged, as the previous statement had added 3 to value without changing the value. The Immediate window can be a very powerful tool!

Before finishing this section, let's return to Watch variables once again for a few final features. One feature is **QuickWatch**, a quick way to bring up information on an identifier, and **Pin to Source**, a great way to have a Watch variable appear next to the line of code. Both of these options are available when you right-click an identifier. Go ahead, try it!

Right-click the **letterCount** variable, and select **QuickWatch**. Close the QuickWatch dialog box. Right-click the **letterCount** variable again, and select **Pin to Source**. Then, hover the mouse pointer over the pinned **letterCount** watch, and select the **X** option to remove it.





Click the continue button to let the program continue, and click **Close** to end the program. You can close the window we opened earlier now too.

So, that concludes our study of the debugging features in Studio. No, we haven't covered all of them, but the ones we did cover should help you feel more confident in stepping through your code, and letting Studio help you learn more about the C# language!

Before you move on to the next lesson, do your homework! Right-click in the window where this lesson text appears and select **Back**. Then select **Quiz** for this lesson in the syllabus and answer the quiz questions. When you finish the quiz questions, click **HAND IT IN** at the bottom of that window. Then do the same with the Project(s) for the lesson. Your instructor will grade your quiz(zes) and project(s) and provide guidance if needed.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

One-Dimensional Arrays, the foreach Keyword, Initializing, and the params Keyword

In this lesson, we'll learn about arrays (an important programming storage structure), the **foreach** keyword, and the **params** keyword. Also, we'll present a final project.

This lesson includes the following sections:

- [Array Fundamentals](#)
- [Advanced Array Topics](#)
- [Course Project](#)

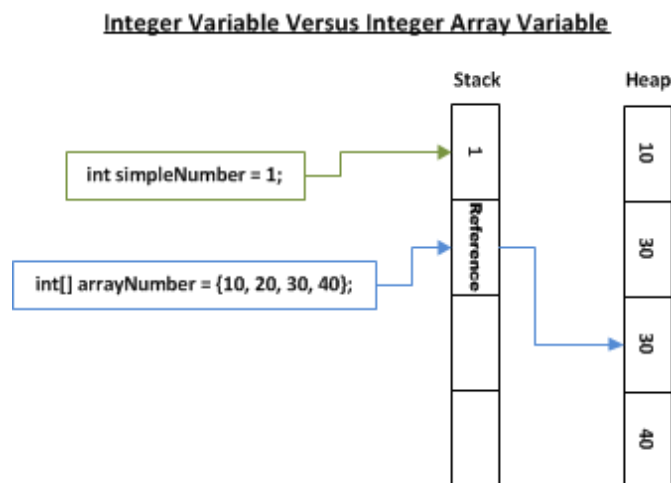
Let's get started!

Array Fundamentals

What Is An Array?

We've used numerous variable types as we've learned the C# language. Also, we've learned the difference between a value data type and a reference data type. Value data type variables store a single value, whereas a reference data type stores a reference to a memory location. The memory location may store a single value, or a data structure that stores multiple values. An array is a reference data type that holds multiple values, or elements, for a specific data type. Even though an array can hold multiple values, each value is referenced through a single variable name.

The following image illustrates the difference between a simple integer variable and an integer array variable. The simple integer variable stores the value of the variable on the stack, whereas the integer array variable stores a reference to a memory location on the stack, which references heap memory, which stores each element of the array.



The illustration shows a way to declare and initialize an array with values. The next section will cover array declaration and initialization in detail.

Array Declaration, Allocation, and Initialization

From the previous illustration, we can see that an array declaration uses the bracket symbols (`[]`) to indicate we're creating an array. When declaring an array, you have the option to simply declare the array without allocating any array elements. As a reference data type, if no memory is allocated for the array, the value of the array variable is **null**, indicating that the value stored on the stack for the array does not reference any memory. The code snippet below shows how to declare an integer array that does not reference any heap memory.

Array Declaration

```
// Integer array declaration with a value of null.  
int[] integerArray;
```

How many elements are in the array? As we indicated, no heap memory has been allocated, so there are 0 elements in the array. To allocate heap memory space for the array, we use the **new** keyword. The following code snippet shows how to declare an integer array, with an initial size of five, which means the integer array has pre-allocated space for five elements.

Array Declaration and Allocation

```
// Integer array declaration with pre-allocated space for five integer elements.  
int[] integerArray = new int[5];
```

When creating memory space for an array, you must specify the size of the array, unless you're also initializing it. We'll learn more about initializing arrays in a moment. The number of elements allocated for an array may be changed using the **new** keyword, but you must be aware that any data stored in the elements is lost. The code snippet below shows how to change the number of array elements. The default values stored in array elements are whatever is appropriate for the data type of the array. For an integer, the default value is zero, so resizing an array will set all of the element values to zero, regardless of their previous values.

Changing Array Allocation Size

```
// Integer array declaration with pre-allocated space for five integer elements.  
  
int[] integerArray = new int[5];  
// Changing the number of array elements to 25.  
// Note that any previously stored values in the array elements are erased.  
integerArray = new int[25];
```

When creating and allocating space for an array, we can also initialize the array elements with values using the curly braces {}:

Initializing An Array

```
// Integer array declaration with pre-allocated space for five integer elements,  
// with array element initialization.  
int[] integerArray = new int[5]{ 1, 2, 3, 4, 5 };
```

When initializing an array, you can omit the array size:

Initializing An Array Without Size

```
// Integer array declaration with pre-allocated space for five integer elements,  
// with array element initialization, omitting array size.  
int[] integerArray = new int[]{ 1, 2, 3, 4, 5 };
```

Note

When using an array size during array initialization, the number of array initialization values must match the size of the array. Omitting the array size during initialization is a standard practice, creating an array with a size that matches the number of array initialization values.

Array initialization is so common that the C# language allows you to omit the use of the **new** keyword:

Initializing An Array Without new Keyword

```
int[] integerArray = { 1, 2, 3, 4, 5 };
```

So, the following array declarations are all equivalent:

Equivalent Array Initialization Statements

```
int[] integerArray = new int[5]{ 1, 2, 3, 4, 5 };  
int[] integerArray = new int[]{ 1, 2, 3, 4, 5 };  
int[] integerArray = { 1, 2, 3, 4, 5 };
```

Although we've listed these as equivalent initialization techniques, we need to point out that these are only equivalent when you first declare an array using the initialization technique without the **new** keyword. If you

already have an array, and you want to initialize the array to a different set of values, you must use the **new** keyword.

Accessing Array Elements

We've seen how to initialize an array with a group of values. We can also access individual array elements using the square brackets []. Array elements in C# are 0-based, which means that array element numbers start at zero, not one.

Accessing Array Elements

```
int[] integerArray = { 1, 2, 3, 4, 5 };
integerArray[0] = 10;
integerArray[4] = 50;
// Output the fifth array element value.
System.Console.WriteLine("Fifth array element value is " + integerArray[4]);
```

The code above changes the value of the first [0] array element from **1** to **10**, and the fifth [4] element from **5** to **50**. As we mentioned, the **0** element is the first element in an array. If you count the number of array initializer values, the number is five, so the size of the array—the number of array elements—is five. We can access the size of an array using the **Length** property, if the array has been allocated space. If the array has simply been declared, the C# compiler will display an error. The code snippet below illustrates the use of the **Length** property of an array.

Array Length Property

```
// Display the Length, or size, of an array.
int[] integerArray = { 1, 2, 3, 4, 5 };
System.Console.WriteLine("Array length: " + integerArray.Length);
```

This code would output the line "Array length: 5."

Coding

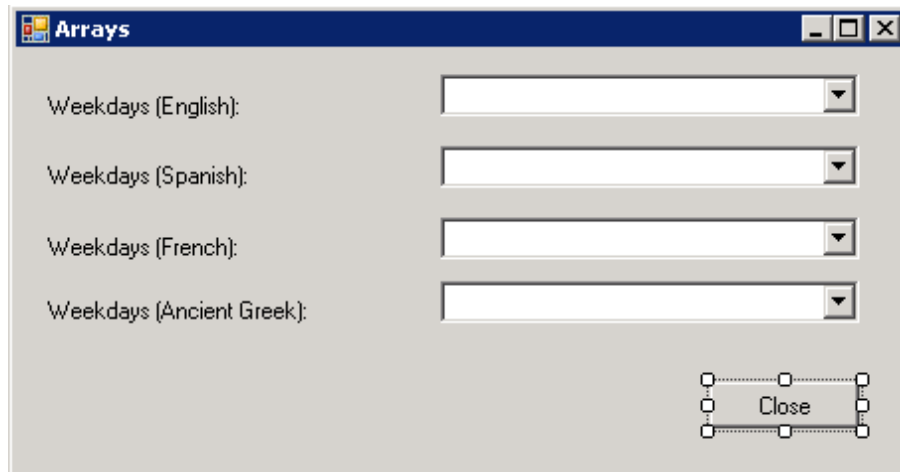
Okay, let's see arrays in action!

Select **File | New | Project**. Change the Name of the project to **Arrays** and click **OK**.

Change the entry for Form1.cs to **Arrays.cs**, and the Form1 Form title bar's Text property to **Arrays**. Click  to save your changes.

Now let's add the controls we'll need.

Add the controls listed below. Change each control Name property to match the Name Property column, and each Text property to match the Text Property column. Arrange the controls similar to the image below.



Note

For each ComboBox, we've also included the DropDownStyle property that needs to be set to DropDownList. The choices are: Simple, a single, editable entry; DropDown, a multi-selection, editable entry; and DropDownList, a multi-selection, non-editable list.

Object	Name Property	Text Property	DropDownStyle
Label	weekdayEnglishLabel	Weekdays (English):	
ComboBox	weekdaysEnglishComboBox		DropDownList
Label	weekdaySpanishLabel	Weekdays (Spanish):	
ComboBox	weekdaysSpanishComboBox		DropDownList
Label	weekdayFrenchLabel	Weekdays (French):	
ComboBox	weekdaysFrenchComboBox		DropDownList
Label	weekdayAncientGreekLabel	Weekdays (AncientGreek):	
ComboBox	weekdaysAncientGreekComboBox		DropDownList
Button	closeButton	Close	

Click  to save your changes.

Double-click each ComboBox control and the Button control to add the following events to the Arrays form.

Control	Event	Event Name
weekdaysEnglishComboBox	SelectedIndexChanged	weekdaysEnglishComboBox_SelectedIndexChanged
weekdaysSpanishComboBox	SelectedIndexChanged	weekdaysSpanishComboBox_SelectedIndexChanged
weekdaysFrenchComboBox	SelectedIndexChanged	weekdaysFrenchComboBox_SelectedIndexChanged
weekdaysAncientGreekComboBox	SelectedIndexChanged	weekdaysAncientGreekComboBox_SelectedIndexChanged
closeButton	Click	closeButton_Click

Next, let's add a couple of methods, and code to the event handlers.

Select the **Arrays.cs** Form Code Editor, and modify the code as shown below.

Note

Okay, so we thought we'd be funny by adding Ancient Greek. You should type in the sample code, just as we've always recommended, but for the Spanish and Ancient Greek arrays, you can copy the array initialization, unless you enjoy typing letters with diacritic marks and Ancient Greek!

Arrays.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Arrays
{
    public partial class Arrays : Form
    {
        public Arrays()
        {
            InitializeComponent();

            // Set up weekday ComboBox controls.
            populateWeekdayComboBoxes();
        }

        private void populateWeekdayComboBoxes()
        {
            // Declare weekday arrays.
            string[] daysOfWeekEnglish = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
            string[] daysOfWeekSpanish = { "Domingo", "Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado" };
            string[] daysOfWeekFrench = { "Dimanche", "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi" };
            string[] daysOfWeekAncientGreek = { "ἡμέρα Ἡλίου (heméra Helíou)",
                                                "ἡμέρα Σελήνης (heméra Selénes)",
                                                "ἡμέρα Ἄρεως (heméra Áreos)",
                                                "ἡμέρα Ἑρμοῦ (heméra Hérμου)",
                                                "ἡμέρα Διός (heméra Diós)",
                                                "ἡμέρα Ἀφροδίτης (heméra Aphrodítes)",
                                                "ἡμέρα Κρόνου (heméra Krónou)" };

            // Populate each ComboBox.
            populateWeekdayComboBox(weekdaysEnglishComboBox, daysOfWeekEnglish);
            populateWeekdayComboBox(weekdaysSpanishComboBox, daysOfWeekSpanish);
            populateWeekdayComboBox(weekdaysFrenchComboBox, daysOfWeekFrench);
            populateWeekdayComboBox(weekdaysAncientGreekComboBox, daysOfWeekAncientGreek);

            // Set default to the first entry in each ComboBox (0-based index).
            setWeekDayComboBoxes(0);
        }

        private void populateWeekdayComboBox(ComboBox comboBox, string[] daysOfWeek)
        {
            // Make sure we have a valid array.
            if (daysOfWeek != null)
            {
                // Clear, then add the days of the week to the ComboBox.
                comboBox.Items.Clear();
                for (int i = 0; i < daysOfWeek.Length; i++)
                {
                    comboBox.Items.Add(daysOfWeek[i].ToString());
                }
            }
        }

        private void setWeekDayComboBoxes(int weekDay)
```

```

    {
        // Set each ComboBox SelectedIndex.
        weekdaysEnglishComboBox.SelectedIndex = weekDay;
        weekdaysSpanishComboBox.SelectedIndex = weekDay;
        weekdaysFrenchComboBox.SelectedIndex = weekDay;
        weekdaysAncientGreekComboBox.SelectedIndex = weekDay;
    }

    private void weekdaysEnglishComboBox_SelectedIndexChanged(object sender,
EventArgs e)
    {
        // SelectedIndex changed, so change all of the rest of the ComboBox
controls.
        setWeekDayComboBoxes (weekdaysEnglishComboBox.SelectedIndex);
    }

    private void weekdaysSpanishComboBox_SelectedIndexChanged(object sender,
EventArgs e)
    {
        // SelectedIndex changed, so change all of the rest of the ComboBox
controls.
        setWeekDayComboBoxes (weekdaysSpanishComboBox.SelectedIndex);
    }

    private void weekdaysFrenchComboBox_SelectedIndexChanged(object sender,
EventArgs e)
    {
        // SelectedIndex changed, so change all of the rest of the ComboBox
controls.
        setWeekDayComboBoxes (weekdaysFrenchComboBox.SelectedIndex);
    }

    private void weekdaysAncientGreekComboBox_SelectedIndexChanged(object se
nder, EventArgs e)
    {
        // SelectedIndex changed, so change all of the rest of the ComboBox
controls.
        setWeekDayComboBoxes (weekdaysAncientGreekComboBox.SelectedIndex);
    }

    private void closeButton_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}

```

Click  and  to save your changes and run the program. Select a day in an of the Weekdays combo boxes and note how the day changes in the other boxes. Cool! When you finish, click **Close**. Let's discuss how it works.

```

.
.
.
namespace Arrays
{
    public partial class Arrays : Form
    {
        public Arrays()
        {
            InitializeComponent();

            // Set up weekday ComboBox controls.
            populateWeekdayComboBoxes();
        }

        private void populateWeekdayComboBoxes()
        {
            // Declare weekday arrays.
            string[] daysOfWeekEnglish = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
            .
            .
            .

            // Populate each ComboBox.
            populateWeekdayComboBox(weekdaysEnglishComboBox, daysOfWeekEnglish);
            populateWeekdayComboBox(weekdaysSpanishComboBox, daysOfWeekSpanish);
            populateWeekdayComboBox(weekdaysFrenchComboBox, daysOfWeekFrench);
            populateWeekdayComboBox(weekdaysAncientGreekComboBox, daysOfWeekAncientGreek);

            // Set default to the first entry in each ComboBox (0-based index).
            setWeekDayComboBoxes(0);
        }

        private void populateWeekdayComboBox(ComboBox comboBox, string[] daysOfWeek)
        {
            // Make sure we have a valid array.
            if (daysOfWeek != null)
            {
                // Clear, then add the days of the week to the ComboBox.
                comboBox.Items.Clear();
                for (int i = 0; i < daysOfWeek.Length; i++)
                {
                    comboBox.Items.Add(daysOfWeek[i].ToString());
                }
            }
        }

        private void setWeekDayComboBoxes(int weekDay)
        {
            // Set each ComboBox SelectedIndex.
            weekdaysEnglishComboBox.SelectedIndex = weekDay;
            weekdaysSpanishComboBox.SelectedIndex = weekDay;
            weekdaysFrenchComboBox.SelectedIndex = weekDay;
            weekdaysAncientGreekComboBox.SelectedIndex = weekDay;
        }

        private void weekdaysEnglishComboBox_SelectedIndexChanged(object sender, EventArgs e)
        {
            // SelectedIndex changed, so change all of the rest of the ComboBox controls.
            setWeekDayComboBoxes(weekdaysEnglishComboBox.SelectedIndex);
        }
    }
}

```

```
.  
.
```

We'll focus on the English version of the arrays, but the discussion applies equally to the other languages.

As you typed in the code, and studied how the code works, you'll notice that we tried to create methods that we could reuse, and that simplified and condensed the techniques we would need to make the program work. As you continue to program, in C# or any language, you will develop the ability to see patterns in your code, and think of programming techniques, such as methods, that will make your code easier to follow. For this sample program, we isolated the types of activities we would need:

- Create arrays that hold each language days of the week.
- Populate each ComboBox with the language-appropriate days of the week.
- Set all ComboBox controls to the same day of the week (initially, and in response to the user changing the day in any of the ComboBox controls).

As a result of identifying these activities, we created three methods to assist in our programming: **populateWeekdayComboBoxes**, **populateWeekdayComboBox**, and **setWeekDayComboBoxes**. Each of these methods corresponds to one of the identified activities. We start the process of creating the arrays by calling **populateWeekdayComboBoxes**. For our program, we need string arrays, specified by the **string[]** syntax. We use the shortcut method of omitting the **new** operator, and simply specify the initialization values, shown in our code {between the braces}.

Within the **populateWeekdayComboBoxes** method, we call the **populateWeekdayComboBox** method for each language we want to display, passing the correct ComboBox control (**weekdaysEnglishComboBox**) and language array (**daysOfWeekEnglish**). Note that for passing an array, we add the array brackets after the data type (**string[]**). After each ComboBox control has been populated from the language arrays, we then need to set all of the ComboBox controls to the same item, which we accomplish by calling **setWeekDayComboBoxes**, passing the number 0, to indicate we want to select the first item in the ComboBox, as ComboBox controls use 0-based indexing, just like arrays.

Within the **populateWeekdayComboBox**, we need to populate each ComboBox control from the language array. We've created a generic **for** loop that will loop through each element in the array using the array **Length** property of the **daysOfWeek** array parameter, incrementing the **i** loop counter variable, and then using **i** to access the correct element of the language array (**daysOfWeek[i]**). Note that we used the **<** (less-than) boolean comparison operator to tell when to stop iterating through the language array elements, as the array **Length** property returns seven (7), but 0-based arrays will have elements zero through 6 (0-6). We've also called the **ToString()** method to illustrate that you need a **string** data type for a ComboBox, and that you can convert your array data, although since we're already dealing with a **string** array, this step is unnecessary.

Within the **populateWeekdayComboBox**, you'll notice that before attempting to iterate through the elements of the language array, we first checked to make sure the language array was not **null**. Adding this step ensures that our code will not raise an error if someone attempts to call the routine with a language array that has not been allocated.

Finally, note that within each ComboBox event handler, we call the **setWeekDayComboBoxes** method, passing the index value of the currently selected item in the ComboBox (**weekdaysEnglishComboBox.SelectedIndex**), thus setting all of the ComboBox controls to the same index value.

So, that's an introduction to arrays! They're very versatile, and very fast. Next, let's move onto some advanced array topics.

Advanced Array Topics

Array Assignments

We've seen how to access individual array elements, but we need to remember that an array is a reference data type, so the rules, or semantics, of a reference data type apply. If you have a reference data type, and you make an assignment to another reference data type, you are not creating a new instance of that data type, but instead are creating a new reference to the first referenced memory. Confused? Let's explain that again, but we'll use arrays to illustrate the point, and a code snippet to help.

In the code snippet below, we declare and initialize an array. Then, we declare a second array, and assign the first array to the second array, in the same line of code. Finally, we declare a third array, and on a separate line of code, assign the first array to the third array.

Array to Array Assignments

```
// Declare and initialize an array.
int[] myFirstArray = { 1, 2, 3 };
// Declare and assign a second array.
int[] mySecondArray = myFirstArray;
// Declare a third array.
int[] myThirdArray;
myThirdArray = myFirstArray;
```

All three arrays will be the same. Notice that we omit the square brackets when dealing with array-to-array assignments. Why? We're not dealing with individual array elements, but the entire array. As a reference data type, the array variable is the memory location that stores the actual array elements.

Note

Arrays, and other reference data types, share a common thread that we need to remember when it comes to garbage collection. An object is not removed from memory until all of the references to that memory are removed. If you perform reference data type assignments, as we've just performed with our arrays, the reference count must reach zero before the garbage collector will free the memory. When all of our reference variables are in the same local scope, all of the references are cleared when the variables go out of scope. But, if we use class scope for a reference variable, and then assign a local reference variable to a class reference variable, even when the local scope is closed, because we've referenced the memory at the class level, that memory will not be freed until the object itself is deleted, or we clear the reference with our code.

Array Iteration and the foreach Statement

The **foreach** statement is a useful tool for iterating through the elements of an array (and other collection data types that will be covered later). The **foreach** statement does more than simply iterate, though—it also enumerates each element as the looping through the elements moves forward, giving you an instance of that specific element. Below is the general syntax of the **foreach** statement.

The foreach Statement Syntax

```
foreach (data type identifier in expression)
{
    embedded statement
}
```

The **data type** and **identifier** serve as the declaration of a read-only iteration variable that represents each element in the **expression** component. The **expression** component must be some type of a collection object, and in our case for this lesson, it must be an array. A **foreach** loops through an array in the index order, starting at zero, and progresses in order through all elements in the array.

Let's see an example. The code below shows where we used a **for** loop in our Arrays example to iterate through the language array, and the equivalent **foreach** code.

Replacing for With foreach

```
// Array iteration using a for loop.
for (int i = 0; i < daysOfWeek.Length; i++)
{
    comboBox.Items.Add(daysOfWeek[i].ToString());
}

// Array iteration using a foreach statement.
foreach (string dayOfWeek in daysOfWeek)
{
    comboBox.Items.Add(dayOfWeek);
}
```

When iterating through an array or other type of collection object, using a **foreach** statement results in "cleaner" (easier to read) code, with a benefit that when iterating, the iteration instance variable is read-only.

The params Parameter Modifier

Previously, we've seen the **out** and **ref** parameter modifiers, and now we'll complete the list of modifiers with the **params** modifier. The **params** modifier indicates that a method parameter may take a variable number of

arguments. Remember, an argument is the actual parameters you supply when you call a method, so a variable number of arguments means that you could call a method that uses the **params** modifier with no arguments, or a hundred arguments, with each argument separated by a comma.

As an alternative, you may also call a method that uses the **params** modifier passing an array as the single argument, so long as the passed array argument data type matches the declared data type of the method.

The code snippet below uses the **params** modifier, and illustrates either method of parameter passing using the **params** modifier.

The params Parameter Modifier

```
// A method that uses the params modifier.
private void displayNames(params string[] names)
{
    foreach (string name in names)
        System.Console.WriteLine("Name: " + name);
}

// How we would call the above method.
// Passing a comma-delimited list as the parameters.
displayNames("Sam", "Fred", "Bob");

// Creating a compatible string array, and passing array as the parameter.
string[] names = {"Tom", "John", "Kathy"};
displayNames(names);
```

You may have a method with more than one parameter mixed with a parameter that uses the **params** modifier, but the parameter that uses the **params** modifier must be the last parameter, and you may only have one parameter that uses the **params** modifier. Why? Method arguments are already separated by a comma, so if we had more than one **params** modifier argument, how would we know where one comma-delimited list ended for one parameter, and the next comma-delimited list began for another parameter? The answer is, we wouldn't be able to tell the difference.

Anonymous Arrays

We've seen anonymous types before, and arrays support the anonymous feature as well. The term *anonymous* means that the data type is inferred by the compiler from the type of data we're referencing. Below is a code snippet that uses the earlier array examples, and includes the equivalent definition as an anonymous array.

Anonymous Arrays

```
// Standard array declaration and initialization.
int[] integerArray = new int[] { 1, 2, 3, 4, 5 };

// Equivalent array declaration and initialization as an anonymous array.
var integerArray = new[] { 1, 2, 3, 4, 5 };
```

Anonymous arrays use the **var** keyword rather than the data type, and require that you use the **new** keyword.

That concludes this lesson on arrays, so now it's time for a final project to wrap up this course!

Course Project

The Assignment

The final project for this course will be an alarm clock application. Below you'll find all of the information you'll need to create this application, including the required functionality in the Statement of Purpose section. You should be familiar with each of the sections, as they were used in the first course.

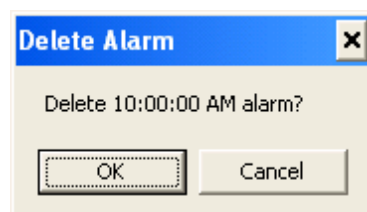
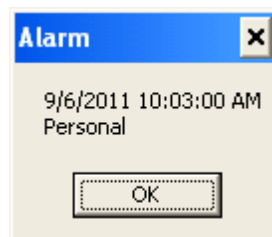
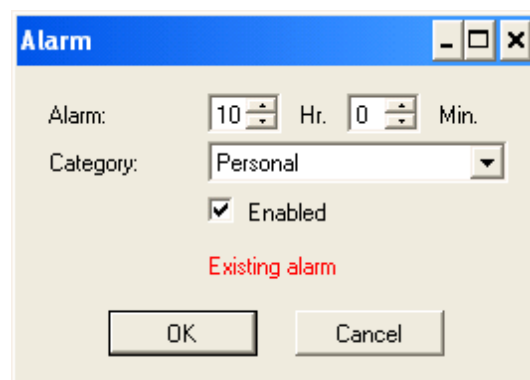
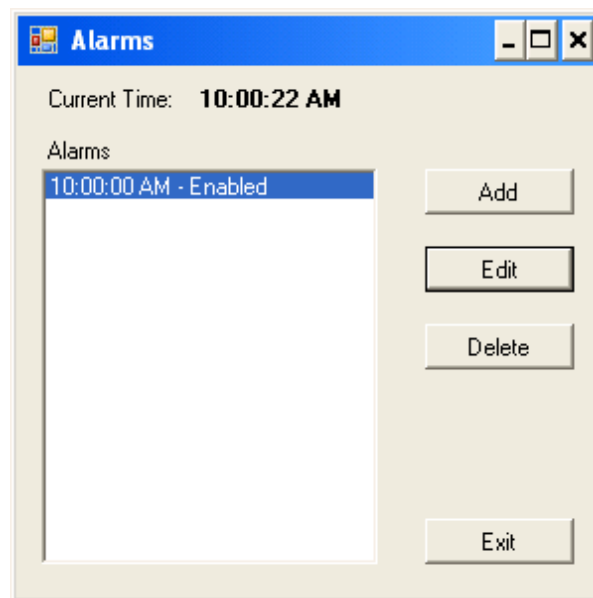
The Materials

Statement of Purpose

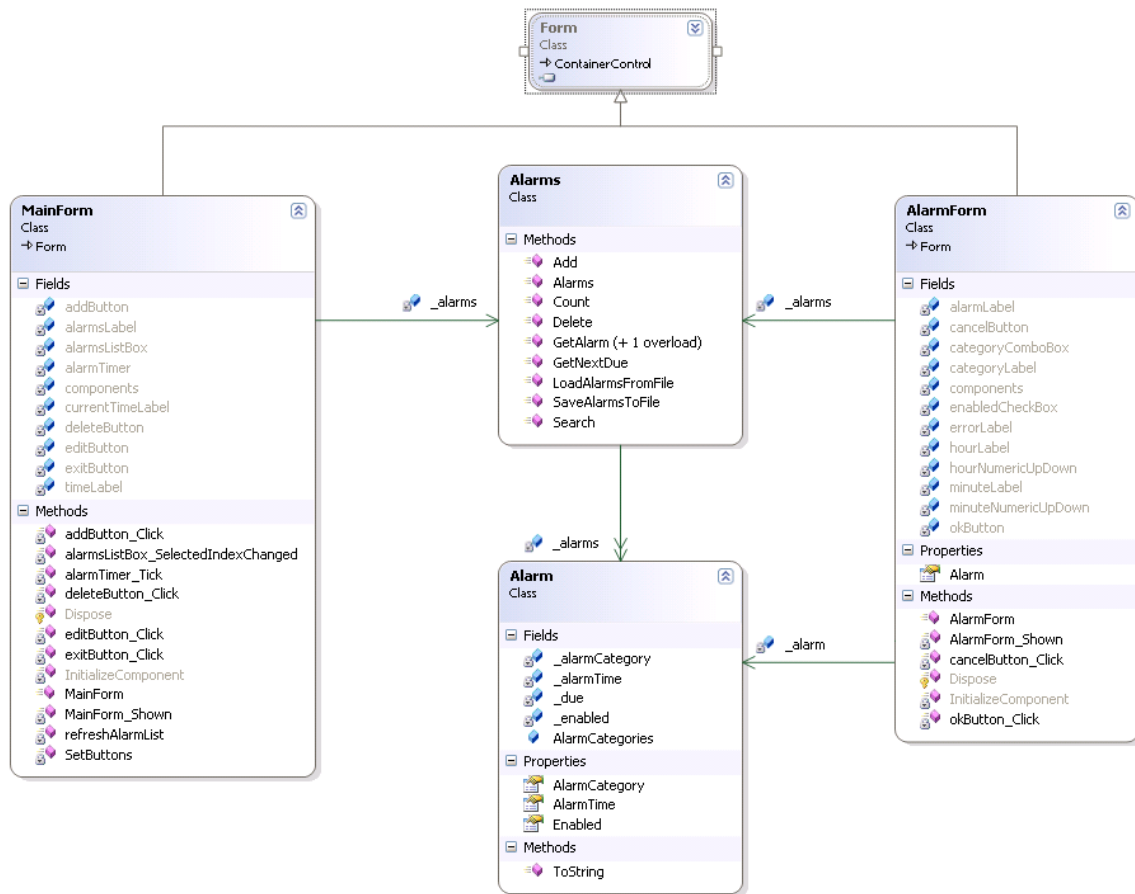
The application will display a list of user-added daily alarms. Alarms are time-based only. Users will be able to add, edit, and delete alarms. The alarms will be displayed in a list. Alarms may be enabled or disabled, and included in one of a user-selectable category list that includes "Personal" and "Business." Only the hour and

minute components of the alarm are set. When an alarm is triggered, the application will display an alarm dialog box that may be dismissed. Alarms are saved to a file when the application closes, and restored from a file when the application starts.

UI Prototype



Class Design Diagrams



Note

The class diagrams below include the full method signatures. By default, Studio displays only the Name of the item in the class diagram; to expand the member format to include the full signature and return type, right-click the class diagram window and select "Change Members Format."

MainForm

Class

→ Form

Fields

- _alarms : Alarms
- addButton : Button
- alarmsLabel : Label
- alarmsListBox : ListBox
- alarmTimer : Timer
- components : IContainer
- currentTimeLabel : Label
- deleteButton : Button
- editButton : Button
- exitButton : Button
- timeLabel : Label

Methods

- addButton_Click(object sender, EventArgs e) : void
- alarmsListBox_SelectedIndexChanged(object sender, EventArgs e) : void
- alarmTimer_Tick(object sender, EventArgs e) : void
- deleteButton_Click(object sender, EventArgs e) : void
- Dispose(bool disposing) : void
- editButton_Click(object sender, EventArgs e) : void
- exitButton_Click(object sender, EventArgs e) : void
- InitializeComponent() : void
- MainForm()
- MainForm_Shown(object sender, EventArgs e) : void
- refreshAlarmList() : void
- SetButtons([bool enabled = false]) : void

AlarmForm

Class

→ Form

Fields

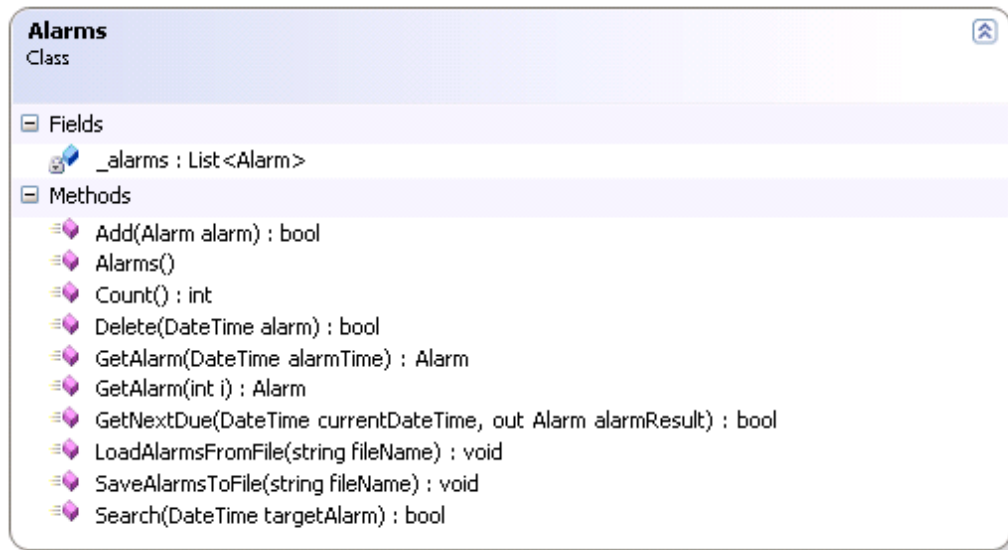
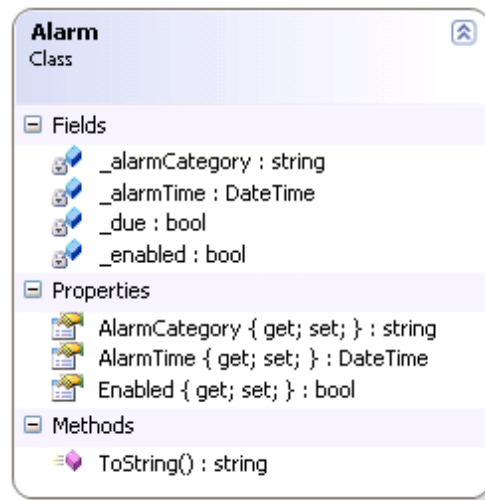
- _alarm : Alarm
- alarmLabel : Label
- cancelButton : Button
- categoryComboBox : ComboBox
- categoryLabel : Label
- components : IContainer
- enabledCheckBox : CheckBox
- errorLabel : Label
- hourLabel : Label
- hourNumericUpDown : NumericUpDown
- minuteLabel : Label
- minuteNumericUpDown : NumericUpDown
- okButton : Button

Properties

- Alarm { get; set; } : Alarm

Methods

- AlarmForm()
- AlarmForm_Shown(object sender, EventArgs e) : void
- cancelButton_Click(object sender, EventArgs e) : void
- Dispose(bool disposing) : void
- InitializeComponent() : void
- okButton_Click(object sender, EventArgs e) : void



TOE Chart

Task	Object	Event
MainForm:		
Display/refresh alarms	alarmsListBox	Form Shown Click (Add, Edit, Delete)
Check/trigger alarms	alarmTimer, timeLabel	Tick
Alarms ListBox selection	editButton, deleteButton	selectedIndexChange
Add alarm	alarmForm, alarmsListBox, alarmTimer	Click
Edit alarm	alarmForm, alarmsListBox, alarmTimer	alarmsListBox item selected, Click
Delete alarm	alarmForm, alarmsListBox, alarmTimer	alarmsListBox item selected, Click
Exit application	exitButton	Click
AlarmForm:		
Save alarm (Add, Edit)	okButton	Click
Cancel save alarm (Add, Edit)	cancelButton	Click

Use Cases - MainForm

- Start application, restore alarms from file.
- Add alarm.
- Edit existing alarm.
- Delete existing alarm.
- Exit application, save alarms to file.

Use Cases - AlarmForm

- Add alarm, verify alarm does not already exist before saving.
- Edit alarm, populate using current alarm data, verify alarm does not already exist before saving.
- Delete existing alarm, prompt for confirmation before removing alarm.

Coding Requirements

As you create this project, we want to make sure you use specific programming techniques you've learned in this course. The list below includes specific programming requirements in addition to delivering the functionality defined in the Statement of Purpose.

- You must have an Alarm class, and an Alarms class.
- You will need to implement most, if not all, of the methods in the class diagram in order for the Alarms application to function.
- The Alarm class must define a public AlarmCategories string array that contains at least "Personal" and "Business."
- Use class initializers when creating an instance of the Alarm class.

The Milestones

The project milestones are:

- Create MainForm Form.
- Add Alarm class.
- Add Alarms class.
- Code MainForm time clock to show the current time using timer Tick event.
- Code Add alarm functionality.
- Code alarm trigger in timer Tick event.
- Code Edit alarm functionality.
- Code Delete alarm functionality.
- Code save alarms to file.
- Code restore alarms from file.
- Test, test, and test!

As you read through the milestones, take note of any that you might not have thought of. Spend time thinking about why the milestone is in the list. Previously, we mentioned incremental development. These milestones reflect that concept. No matter what project you undertake, breaking the project up into manageable tasks and milestones makes it much easier to complete. It's okay if you miss some of the milestones the first time you make the list (or the second, or third, and so on). As you work through the milestones, you might realize you forgot one. Rather than panic, and feel overwhelmed, simply update your milestones with the new task. Once it's added, you can continue with your work. With practice, you'll get better and better at determining the appropriate milestones and tasks as you continue to develop software.

Testing

You'll note that the last milestone is all about testing (did we emphasize that enough?). You should consider two types of testing: *unit testing* and *integrated testing*. Unit testing is the testing you perform as you move through your coding steps. You don't have to perform a test after each milestone, but, as you make significant changes, it might be helpful to test those changes to make sure they work as expected before you move on. In the computer industry, software development is typically split into modules that can be tested as units. New pieces are not considered acceptable until they pass a unit testing process.

Integrated testing is the process of testing a product or piece of product that is composed of much smaller components. Integrated testing detects errors that might not be apparent in unit testing. After you complete all

of your milestones, you'll need to make sure you do your final integrated testing.

Hints

Here are some hints that will help you complete this final course project.

Create MainForm GUI Objects First

Your first task should be to set up the GUI for your MainForm. You can refer to the class diagram to determine what components you'll need, but you may prefer to only look at the UI prototype, and try to create your GUI and events without referencing the class diagram. Once you've made your first attempt at designing the MainForm GUI, and adding the necessary events, you can then compare what you created to the class diagram to validate your naming conventions and needed events.

Create Your Alarm and Alarms Classes Initially with Basic Functionality

Once you create your primary GUI, consider creating the Alarm class, and then the Alarms class. From the class diagram, you can see that the Alarms class "has" or consists of Alarm objects in a collection object. The class diagram also shows other methods we added to assist us when using the Alarms class.

Use a Timer Component

In order to trigger an alarm, we need to have a regular timed event occur, which means we need a Timer component. Before tackling adding alarms, add a timer and a timer Tick event that will update the current time clock on the MainForm. You will need to set the Interval property to less than a second to make sure that the seconds update frequently. The Timer component interval is in milliseconds, or thousandths of a second.

Disable and enable Timer Component Appropriately

When using a Timer component, you should be aware of times when you want to disable the timer, such as when the user is adding a new alarm. You can use the Timer Enabled property to stop the timer, and then to restart the timer after adding the new alarm. We included indications of when you should stop and start the timer in the TOE.

Use DialogResult to Process the AlarmForm OK and Cancel Buttons

We've used the Close() method to close a Form, but sometimes you want the Form to remain open until the user has entered the correct information. Rather than use Close(), we can also use the DialogResult Form property. When this property is set in C# code, the Form will close, and the DialogResult value may be checked in the calling code. The code snippet below shows the appropriate DialogResult code to process the Cancel Button. You can use DialogResult.OK for the okButton.

AlarmForm.cs DialogResult for cancelButton

```
private void cancelButton_Click(object sender, EventArgs e)
{
    this.DialogResult = DialogResult.Cancel;
}
```

Use Form Accessors to Retrieve Form Values

Once the AlarmForm has closed, you can check if DialogResult.OK was returned, and if so, you should use an AlarmForm accessor to retrieve the values set by the user. You can easily use an Alarm member variable, but you have to remember to set the Alarm properties in AlarmForm before closing the form.

For editing, simply set the Alarm properties before displaying the AlarmForm, and then use the AlarmForm properties to populate the AlarmForm UI elements.

Use DateTime TimeOfDay Property

We are only creating daily alarms, not alarms with specific dates, so make sure you use the DateTime TimeOfDay property to extract only the time component of the DateTime variable for comparisons.

Define a ToString() override method for your Alarm class

Many C# objects will attempt to use the ToString() method of an object for display. If you define a ToString() method for your Alarm class, and then add an Alarm object to your alarmsListBox, you can control what is displayed in the ListBox based on how you define your ToString() method. Below are two snippets of code to

explain what we mean.

Alarm.cs ToString() Override Method

```
public override string ToString()
{
    return this._alarmTime.ToLongTimeString() + " - " + (this._enabled ? "Enabled" : "Disabled");
}
```

This snippet uses the C# override statement to override the default ToString() method for the Alarm class provided by C#. We haven't discussed the override statement yet, so you can just type in the C# code as described in the code snippet. This code also includes a few very useful programming techniques:

- Use the DateTime ToLongTimeString() to extract only the Time component of a DateTime object.
- You can have a return statement with conditional evaluation expressions without having to create method variables.
- When using the ternary operator with string concatenation, you will need to surround the expression with parenthesis to correctly evaluate the expression.
- If we want to change what is displayed about an alarm, we only need to change the code in this ToString() method.

So how would we use this ToString() method override? The next snippet shows the steps necessary to take advantage of our overridden method.

MainForm.cs refreshAlarmList Method

```
private void refreshAlarmList()
{
    // Clear the ListBox.
    alarmsListBox.Items.Clear();

    // Add each Alarm to the ListBox, relying on overridden ToString() Alarm method for display.
    for (int i = 0; i < _alarms.Count(); i++)
    {
        alarmsListBox.Items.Add(_alarms.GetAlarm(i));
    }

    // Select the first entry in the ListBox, if any alarms remain.
    if (alarmsListBox.Items.Count > 0)
    {
        alarmsListBox.SelectedIndex = 0;
    }

    // Enable/disable edit/delete buttons that rely on an item being selected in ListBox.
    SetButtons(alarmsListBox.Items.Count > 0 && alarmsListBox.SelectedIndex >= 0);
}
```

Congratulations! You've almost completed the C# Part 2 course! Don't forget to complete the quizzes and submit the final project for the lesson, as you've done for the others.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.