C#.NET 1: Introduction to Object-Oriented Programming Using C#

Introduction

Welcome to the O'Reilly School of Technology **Introduction to Object-Oriented Programming Using C#** course!

Course Objectives

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

- code solutions and compile C# projects within the .NET framework.
- create and manipulate GUI components in C#.
- construct classes, methods, and accessors, and instantiate objects.
- demonstrate knowledge of object-oriented concepts such as encapsulation and polymorphism.
- design user experience and functional requirements for a full-fledged C#.NET project.

In this course, you will learn your way around both Visual Studio and the .NET Framework. You will work with a variety of form controls and base class libraries to create simple Graphical User Interfaces (GUIs). The course covers variables, relational operators, decision statements, classes, and methods and additional topics that will provide a foundation upon which you can build your knowledge of object oriented design concepts and the C# programming language.

You will create several applications throughout the course, which will enhance your professional portfolio and help you advance toward certificate completion.

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.

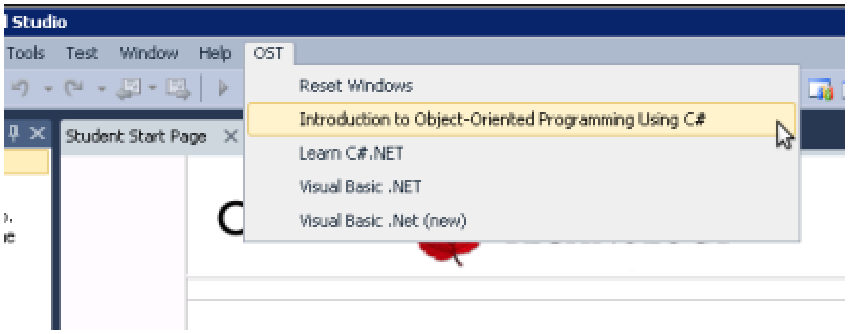
Understanding the Learning Sandbox Environment

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

Note When you see this icon 💾, go ahead and save your work. You can save any time you like, and you'll want to get into the habit of saving your projects often. Whenever you pause to think, move your cursor over the **Save** icon on the Toolbar and click.

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 display other courses you have enrolled in as well.



Let's Do Something!

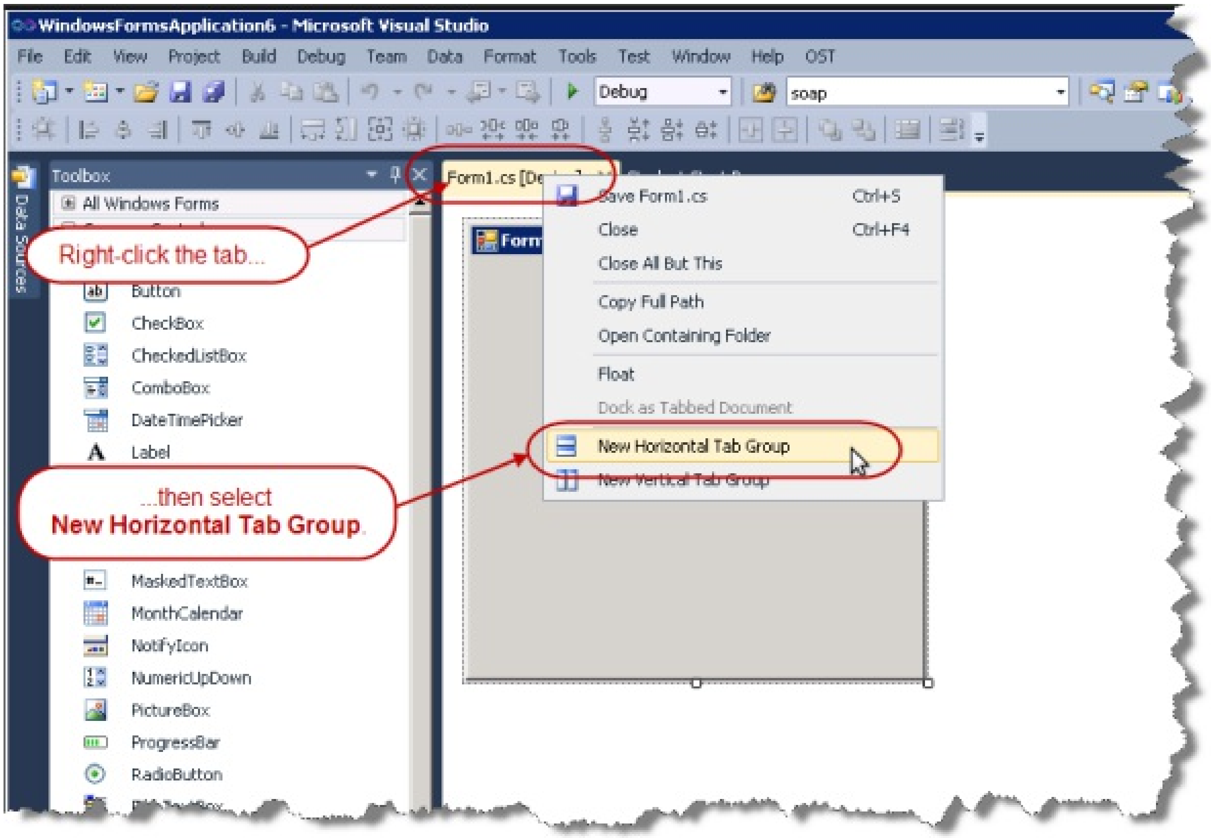
Okay, enough talk—let's create our first C# program! Traditionally, a first program is one that simply displays the text "Hello, World!" We'll be adhering to that tradition. And according to OST tradition, we want to make a working program right away, so let's get going and create this first program. Here's an overview of the steps we'll be taking to make it:

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.

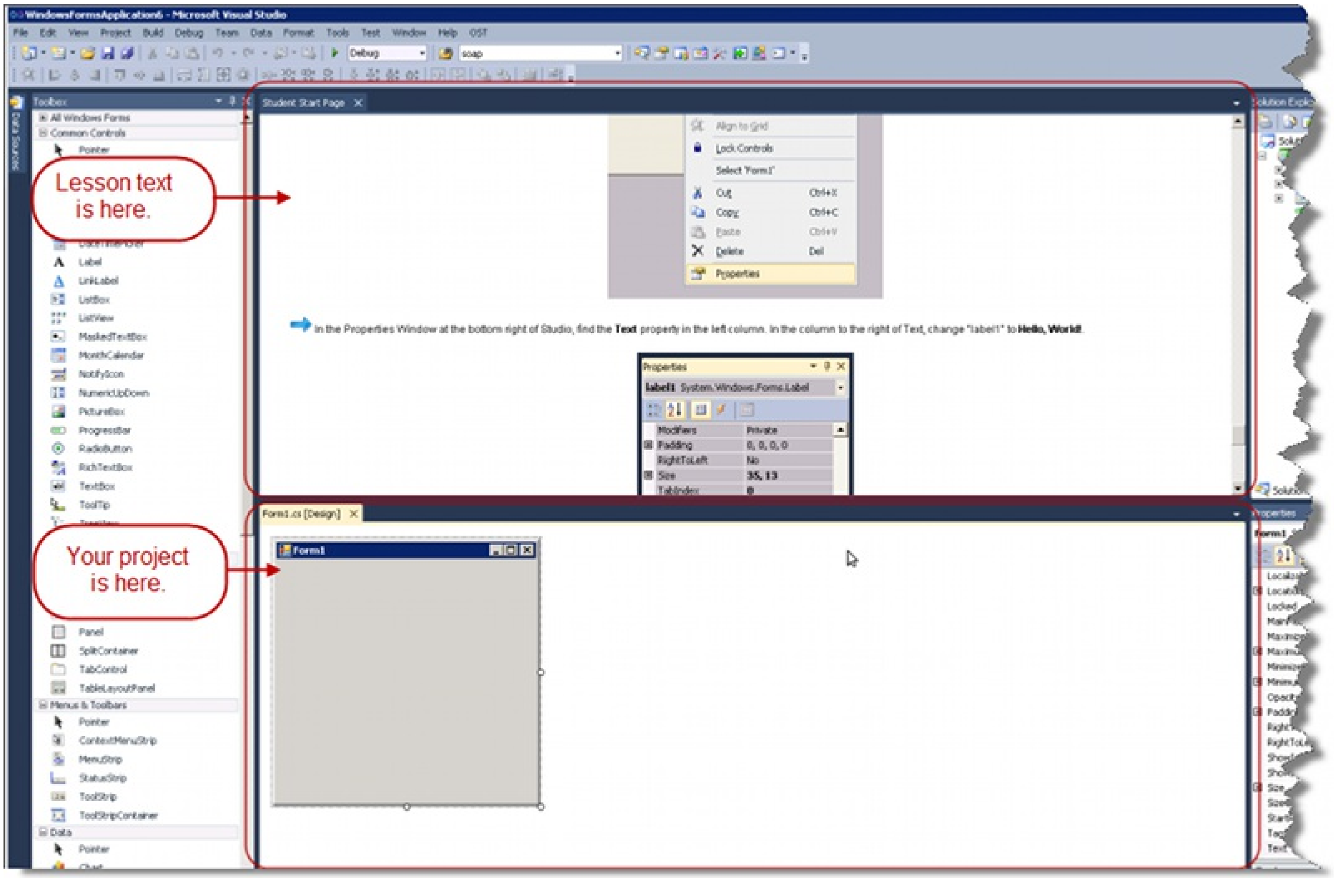
Let's get started!

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

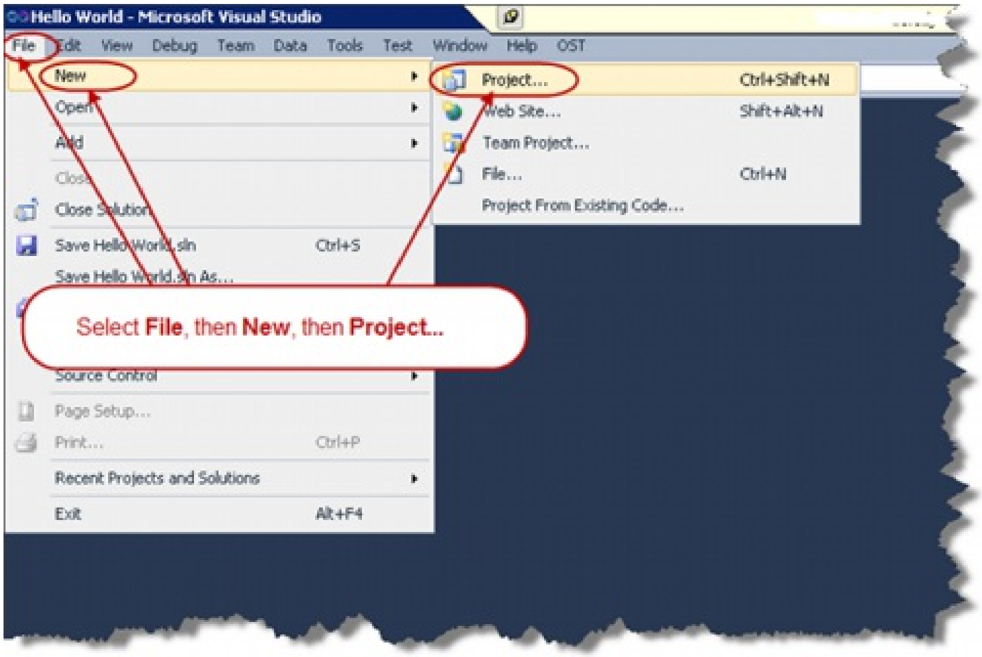


When you finish, you should see something like this.



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!**.

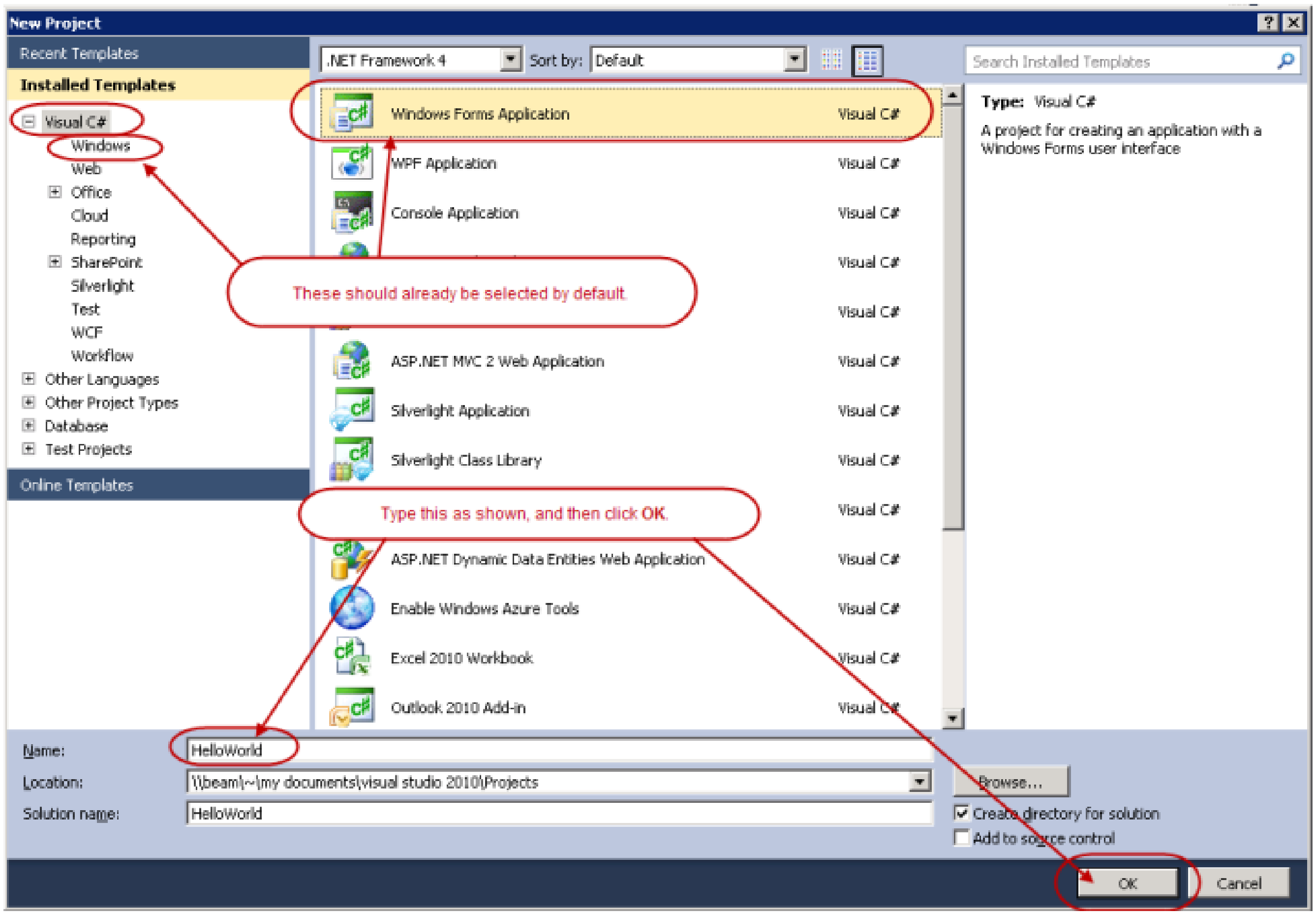
Now, remembering those steps, let's create our project. Select the **File** menu, then **New**, then **Project** (in the future, we'll show this kind of menu sequence like this: **File | New | Project**):



Hello World - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help OST
New
Open
Add
Close
Close Solution
Save HelloWorld.sln Ctrl+S
Save Hello World.sln As...
Source Control
Page Setup...
Print... Ctrl+P
Recent Projects and Solutions
Exit Alt+F4
Project... Ctrl+Shift+N
Web Site... Shift+Alt+N
Team Project...
File... Ctrl+N
Project From Existing Code...

Select **File**, then **New**, then **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.



New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Visual C#
Visual C#
ASP.NET MVC 2 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
Enable Windows Azure Tools Visual C#
Excel 2010 Workbook Visual C#
Outlook 2010 Add-in Visual C#
Type: Visual C#
A project for creating an application with a Windows Forms user interface
Browse...
OK
Cancel

Name: HelloWorld

Location: \\beam\~\my documents\visual studio 2010\Projects

Solution name: HelloWorld

☑ Create directory for solution

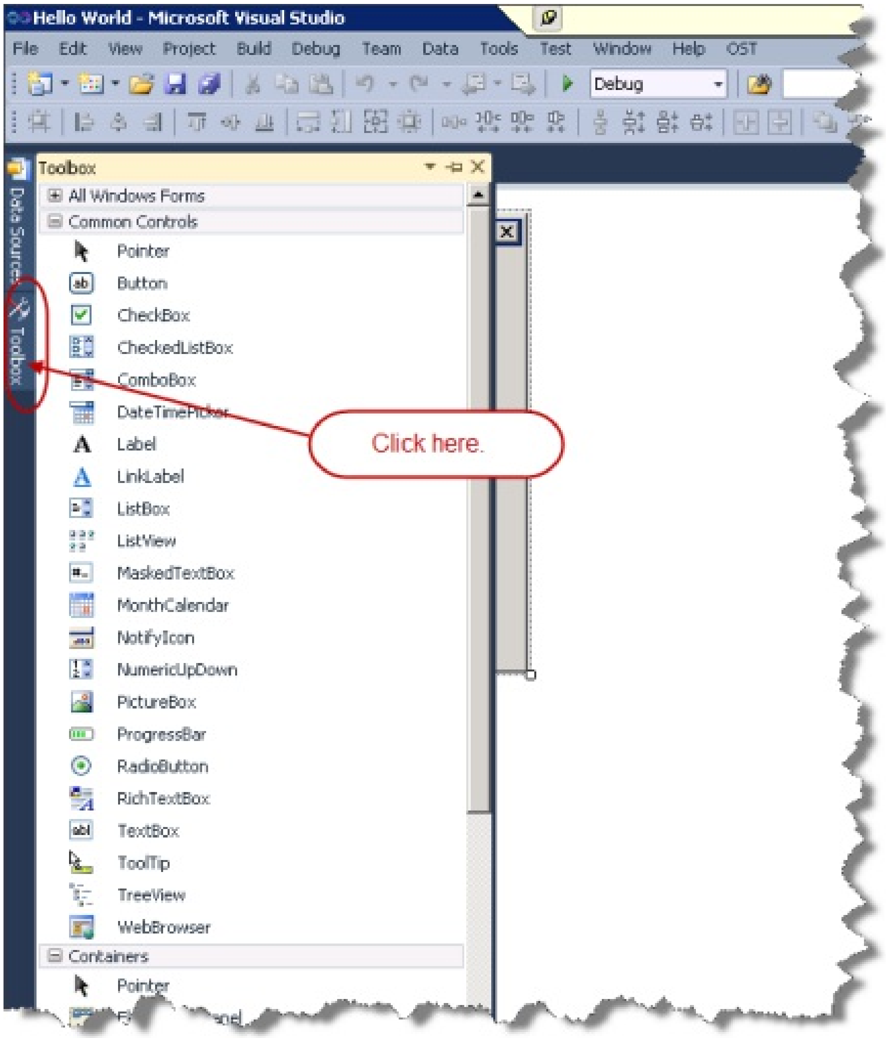
□ Add to source control

These should already be selected by default.

Type this as shown, and then click OK.

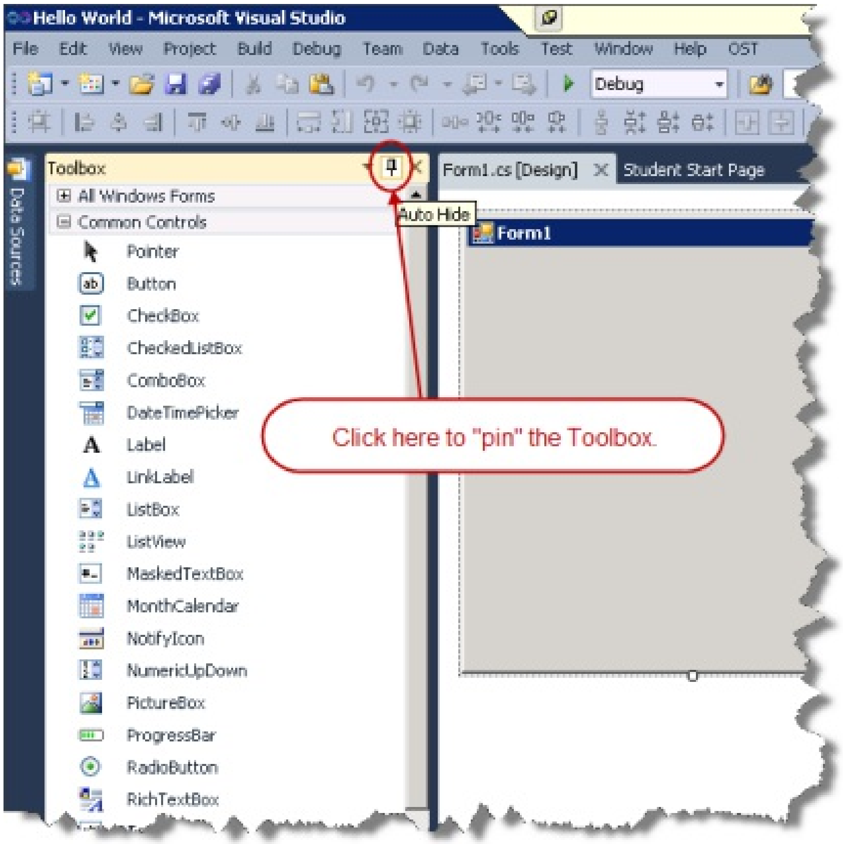
At this point, perform the steps described earlier to get the lesson text back.

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

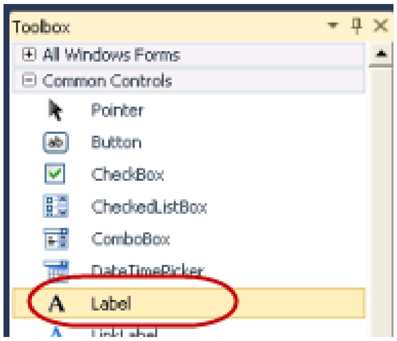


Click here.

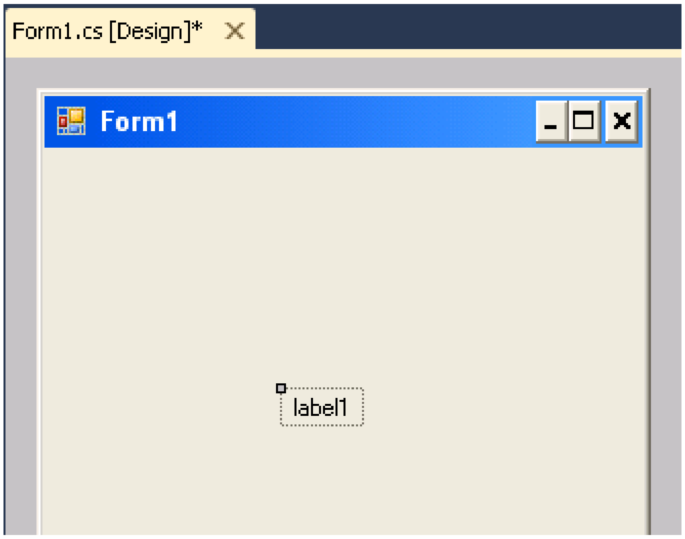
When the Toolbox expands, click on the **Auto Hide** "pin" icon at the top of the Toolbox window so that the pin is pointing down rather than horizontal. The Toolbox will now stay displayed.



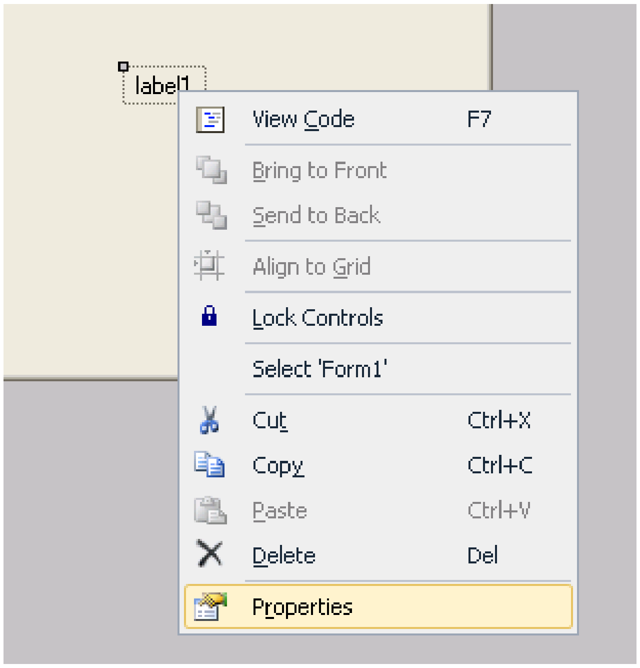
Find the **Label** control under the Common Controls option.



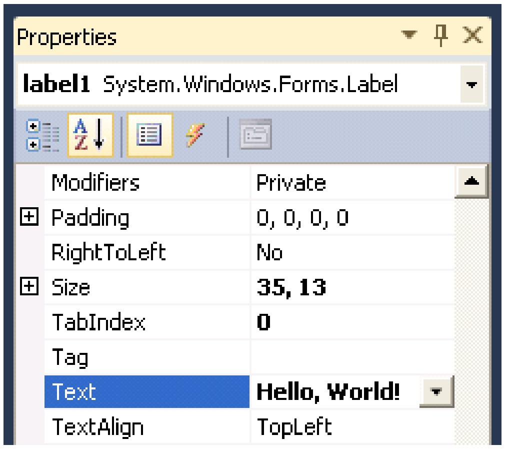
Click and drag the **Label** onto the grey area under the box named Form1 in the middle of Visual Studio.



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!**.

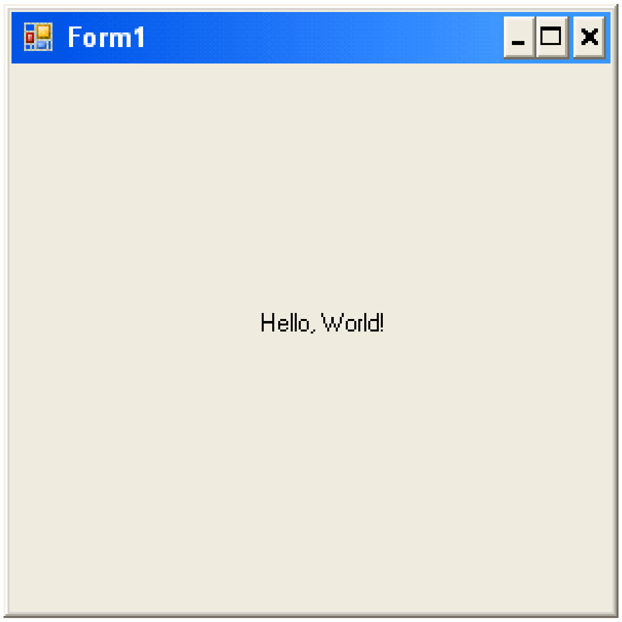


label1 System.Windows.Forms.Label

Modifiers	Private
⊞ Padding	0, 0, 0, 0
RightToLeft	No
⊞ Size	**35, 13**
TabIndex	**0**
Tag	
Text	**Hello, World!**
TextAlign	TopLeft

💾 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!



Hello, World!

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!!!

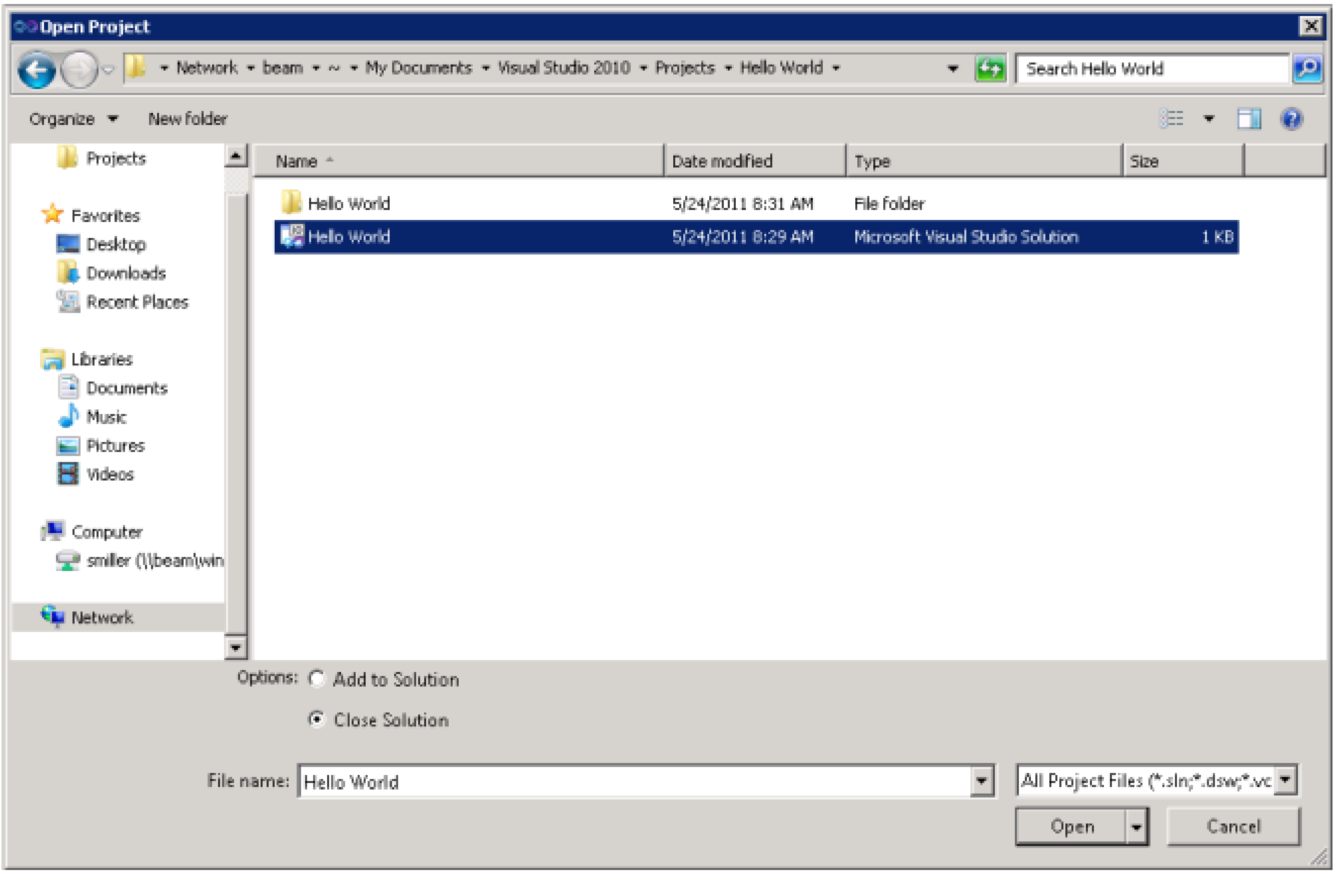
Closing and Reopening a Solution

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.



Name	Date modified	Type	Size
Hello World	5/24/2011 8:31 AM	File folder	
Hello World	5/24/2011 8:29 AM	Microsoft Visual Studio Solution	1 KB

Options: ○ Add to Solution

● Close Solution

File name: Hello World

All Project Files (*.sln;*.dsw;*.vc

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

Next

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!

Getting Started

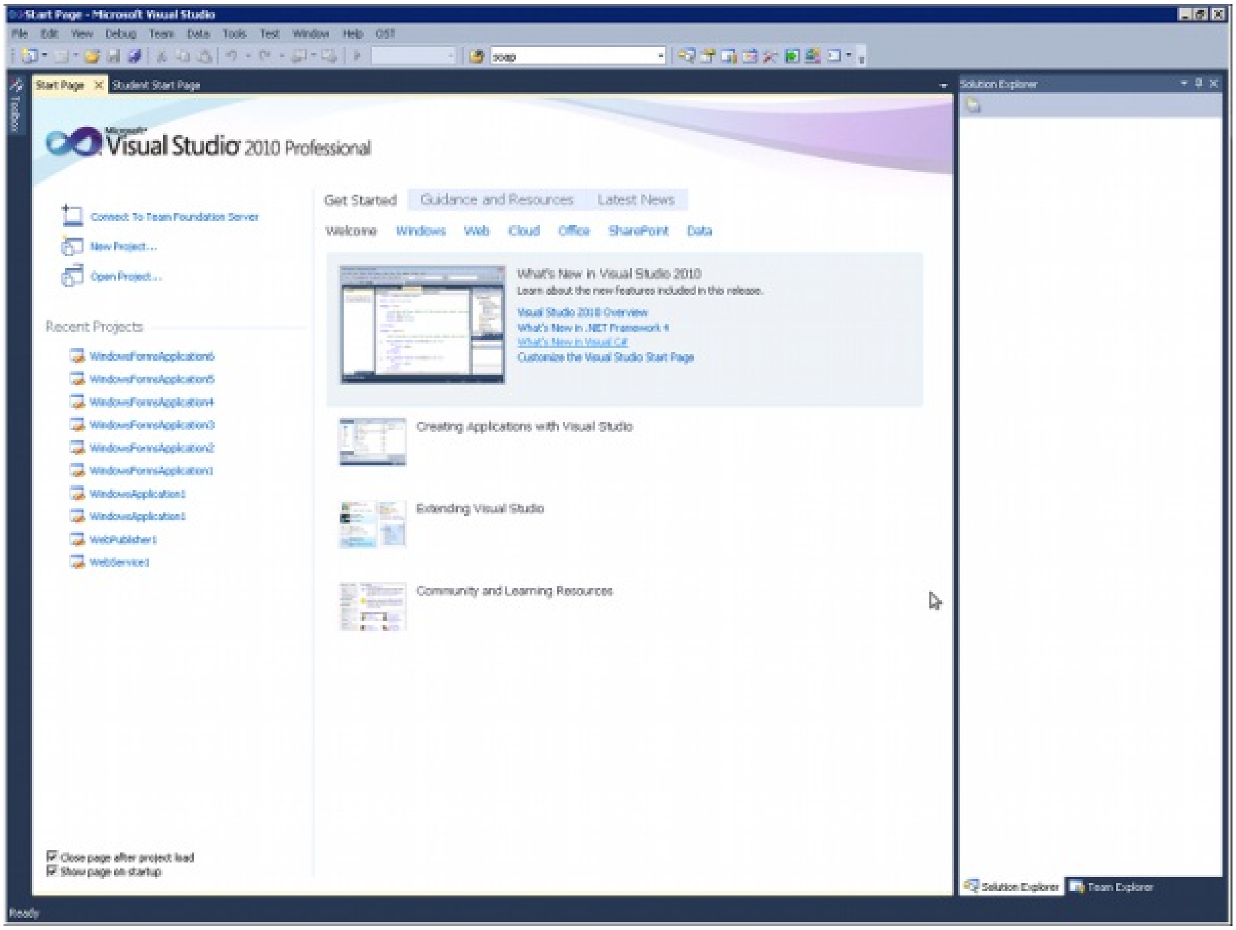
Introduction

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 Integrated Development Environment 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 how to use those features and components to begin creating your own software.

A Quick Tour

The Studio IDE initial Start Page is shown below. When you first visit this page, 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.

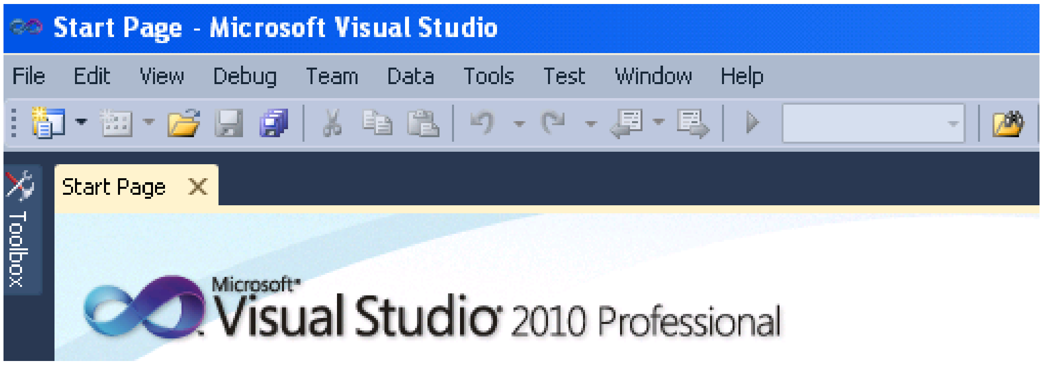


The Start Page is a convenient presentation of many initial tasks you may want to perform when first opening Studio.

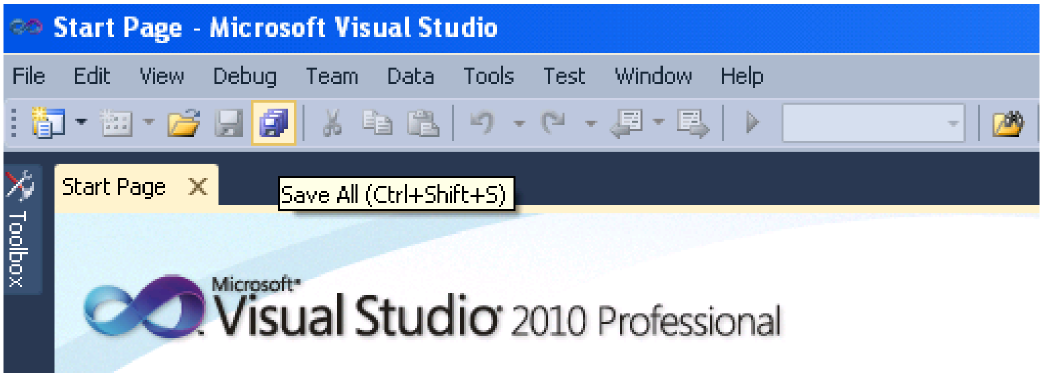
Close this page by clicking the **X** on the **Start Page** tab.

Tip You can view the Start Page again at any time by selecting **View | Start Page** in the menu at top.

The Studio IDE includes a command menu along the top of the screen (including File, Edit, View, etc.), icons just below the menu, and components below the icons. All Studio IDE elements are context-sensitive, which means that certain elements may be disabled or unavailable depending on what you are doing. In the image below from the Start Page, notice that many of the icons look dim, or gray. These icons are disabled.


Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Start Page
Toolbox
Microsoft Visual Studio 2010 Professional

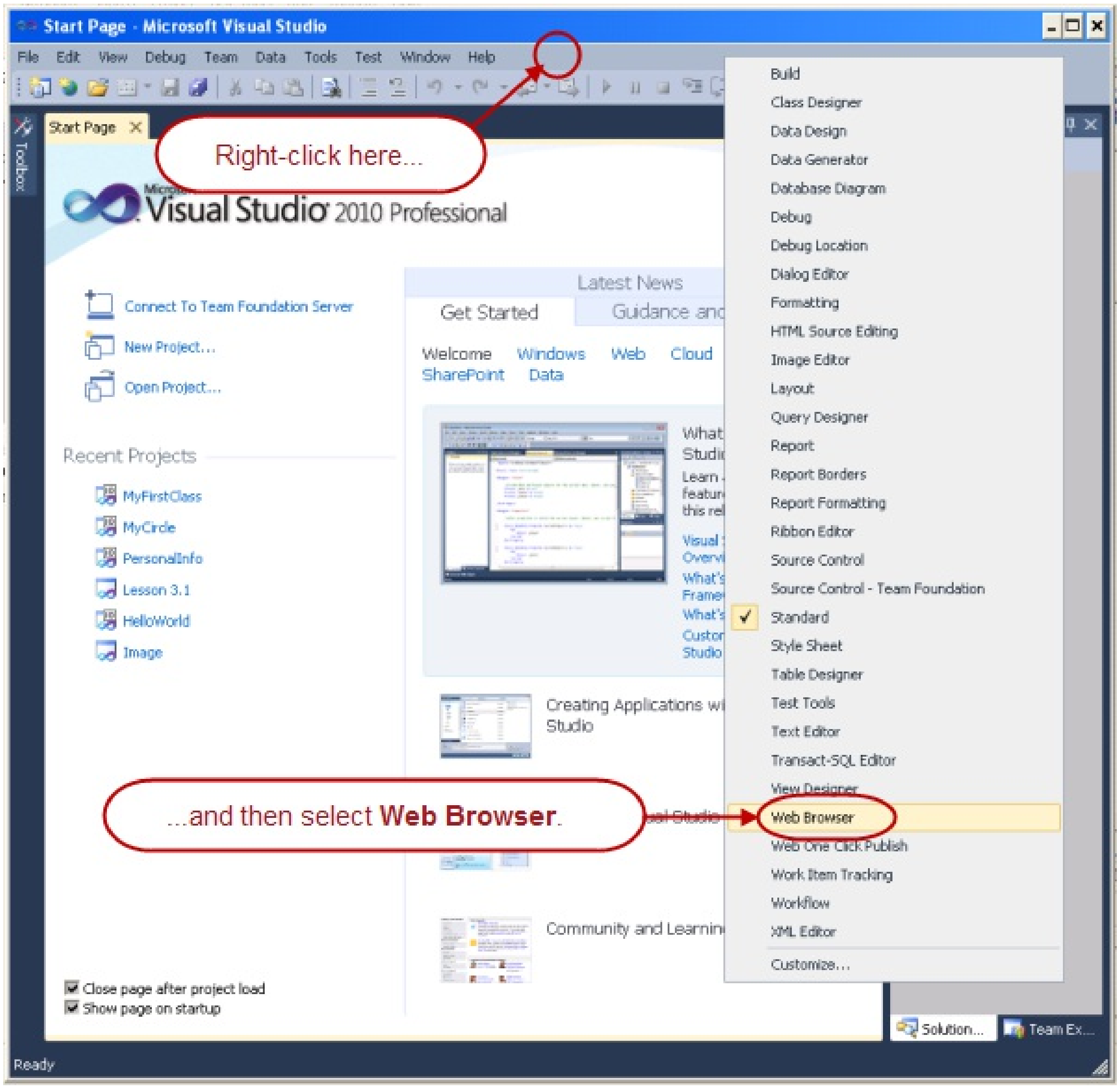
Move your mouse pointer over one of the icons, and pause until a popup appears. The image below shows the **Save All** popup.


Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Start Page
Save All (Ctrl+Shift+S)
Toolbox
Microsoft Visual Studio 2010 Professional

These popups display the icon function, and keyboard shortcut, if available. The icon function will also correspond to the same functionality available through the menu. For example, the **Save All** function is also available from the **File** menu.

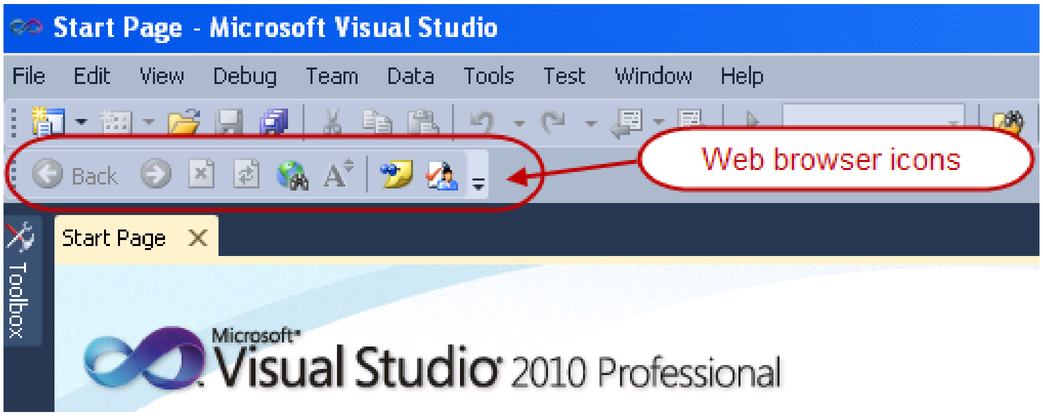
The Studio IDE contains many more elements than initially appear, and as you work more with Studio, you will learn to add the elements that you prefer.

Right-click in the blank area at the top of Studio next to the menu items or icons, and select **Web Browser**. Icons appropriate to a web browser appear in a new toolbar below the default icons. The following two images show the popup menu of choices, and the new Web Browser icons added to the top left of the IDE below the existing icons.



Right-click here...

...and then select **Web Browser**.

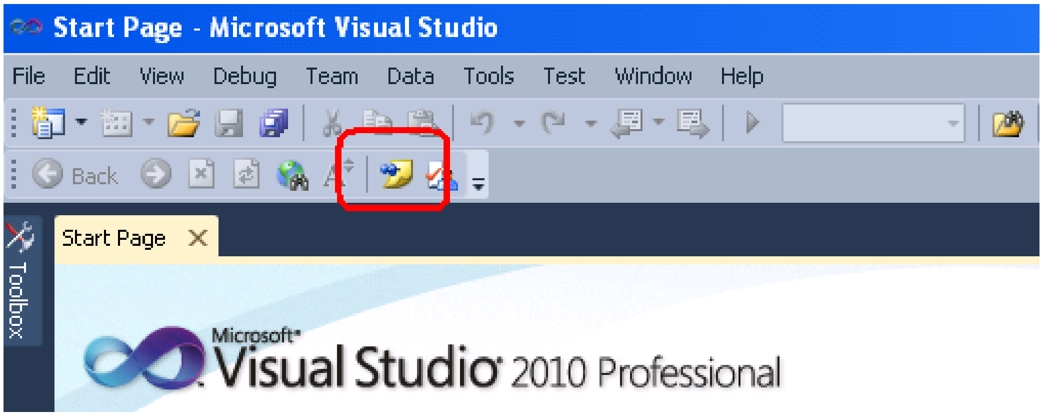


Web browser icons

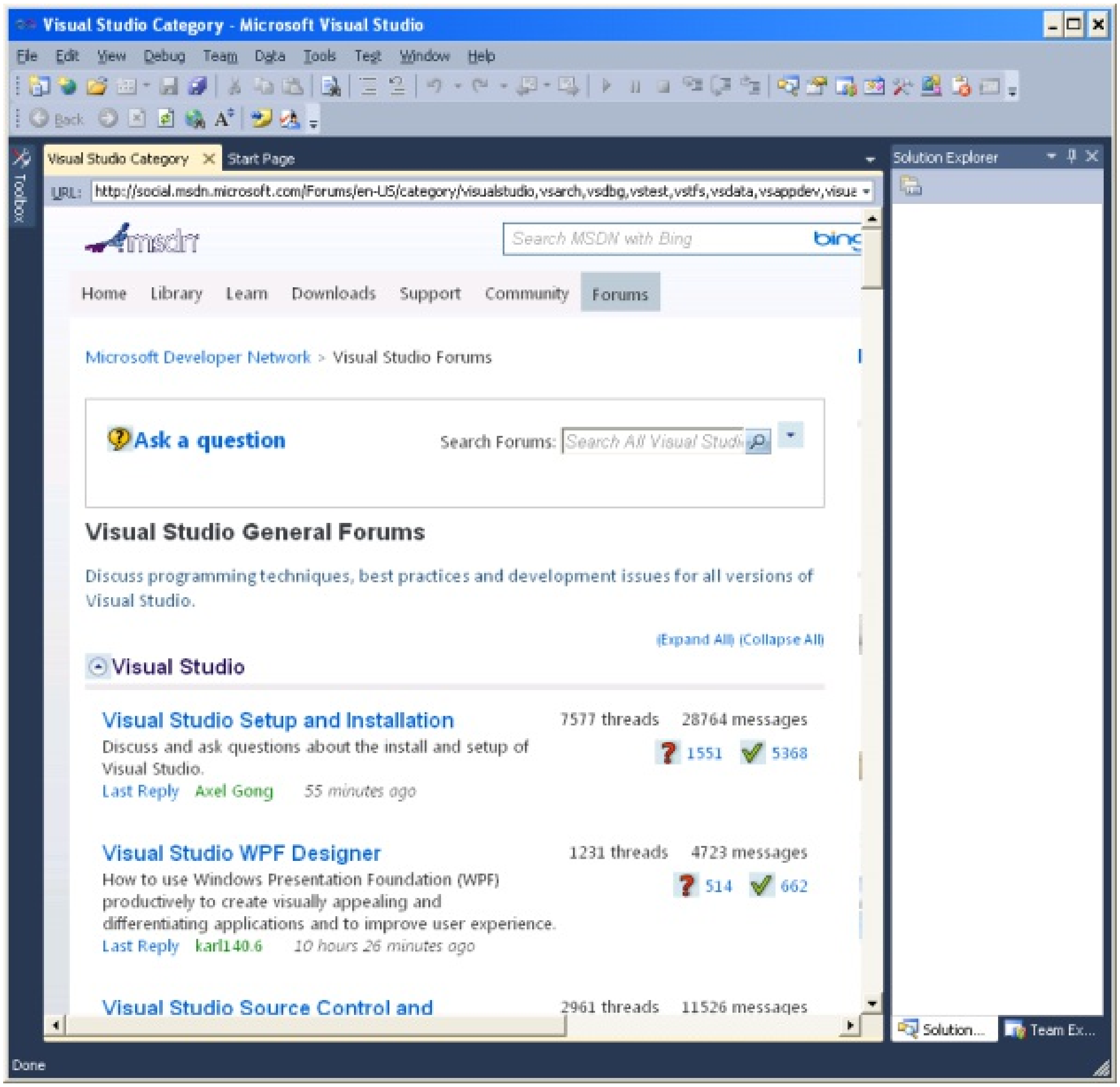
Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Back
Start Page
Toolbox
Microsoft
Visual Studio 2010 Professional

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:


Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Back
Toolbox
Start Page
Microsoft® Visual Studio® 2010 Professional

You will see content similar to this:


Visual Studio Category - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Back
Toolbox
Visual Studio Category
Start Page
Solution Explorer
URL: http://social.msdn.microsoft.com/Forums/en-US/category/visualstudio,vsarch,vsdbg,vstest,vstfs,vsdata,vsappdev,visua
msdn
Search MSDN with Bing
bing
Home Library Learn Downloads Support Community Forums
Solution...
Team Ex...
Done

Microsoft Developer Network > Visual Studio Forums

Ask a question

Search Forums: Search All Visual Studi

Visual Studio General Forums

Discuss programming techniques, best practices and development issues for all versions of Visual Studio.

(Expand All) (Collapse All)

Visual Studio

Visual Studio Setup and Installation — 7577 threads 28764 messages
Discuss and ask questions about the install and setup of Visual Studio.
? 1551 ✓ 5368
Last Reply Axel Gong 55 minutes ago

Visual Studio WPF Designer — 1231 threads 4723 messages
How to use Windows Presentation Foundation (WPF) productively to create visually appealing and differentiating applications and to improve user experience.
? 514 ✓ 662
Last Reply karl140.6 10 hours 26 minutes ago

Visual Studio Source Control and — 2961 threads 11526 messages

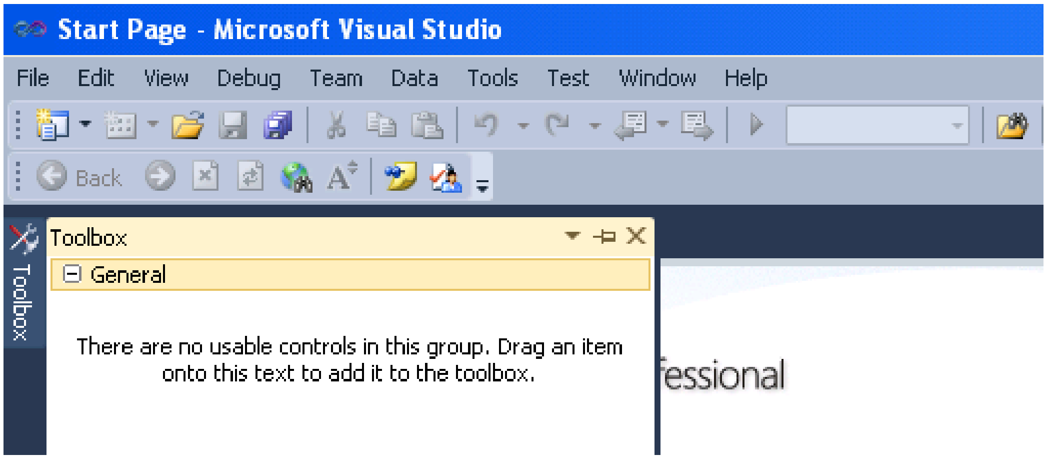
This is a valuable resource for learning more about developing software using Visual Studio—we'll visit it again later.

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:

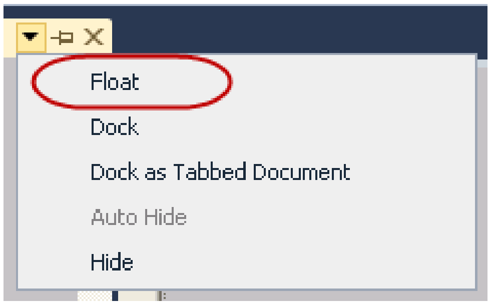


There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

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**.

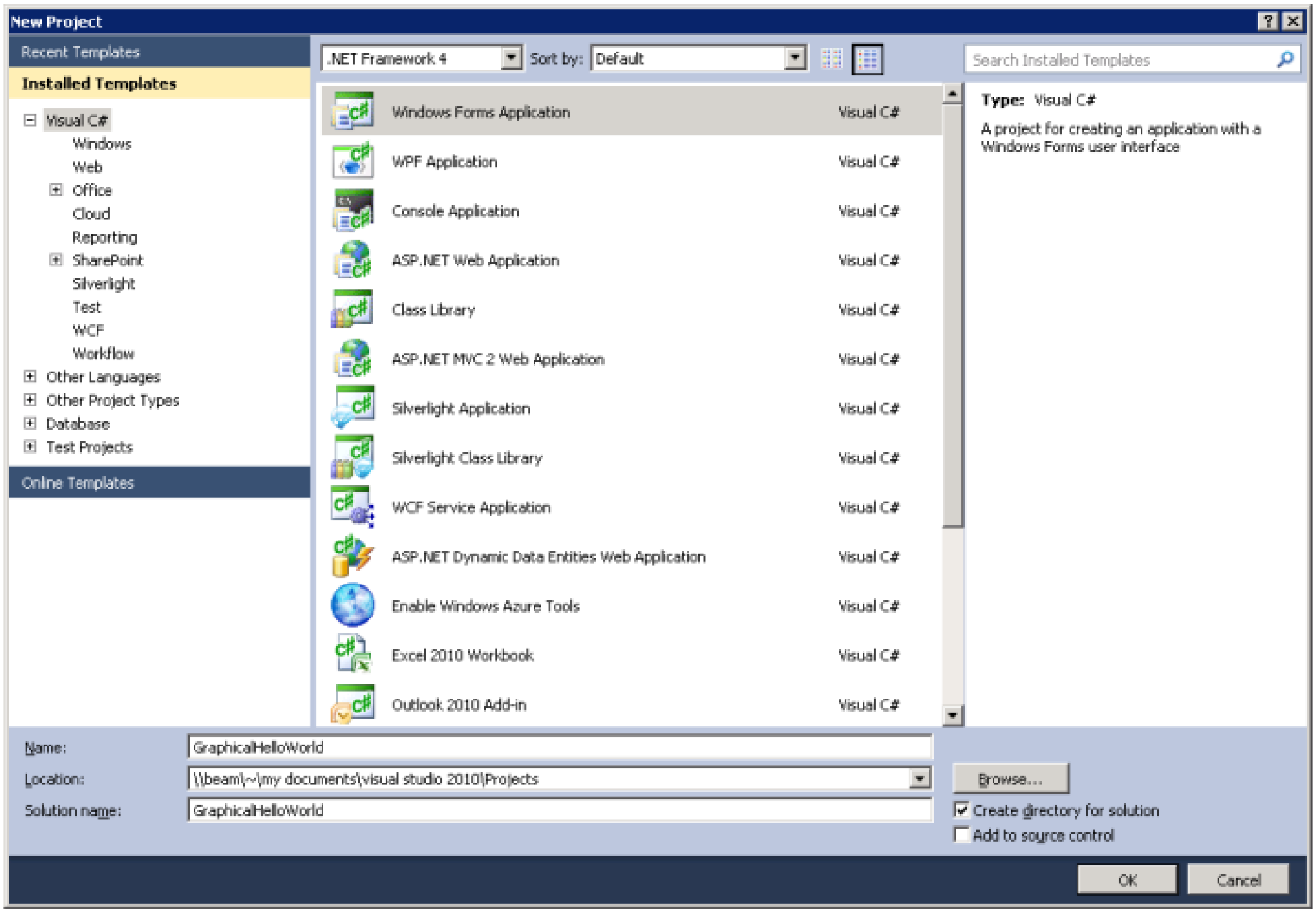
Toggle the Auto Hide pushpin and note the behavior when the pushpin is vertical versus when it is horizontal. When you finish, make sure it is vertical so the Toolbox remains onscreen.

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**:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
ASP.NET MVC 2 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
WCF Service Application Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
Enable Windows Azure Tools Visual C#
Excel 2010 Workbook Visual C#
Outlook 2010 Add-in Visual C#
Browse...
OK
Cancel

.NET Framework 4 Sort by: Default

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: GraphicalHelloWorld

Location: \\beam\~\my documents\visual studio 2010\Projects

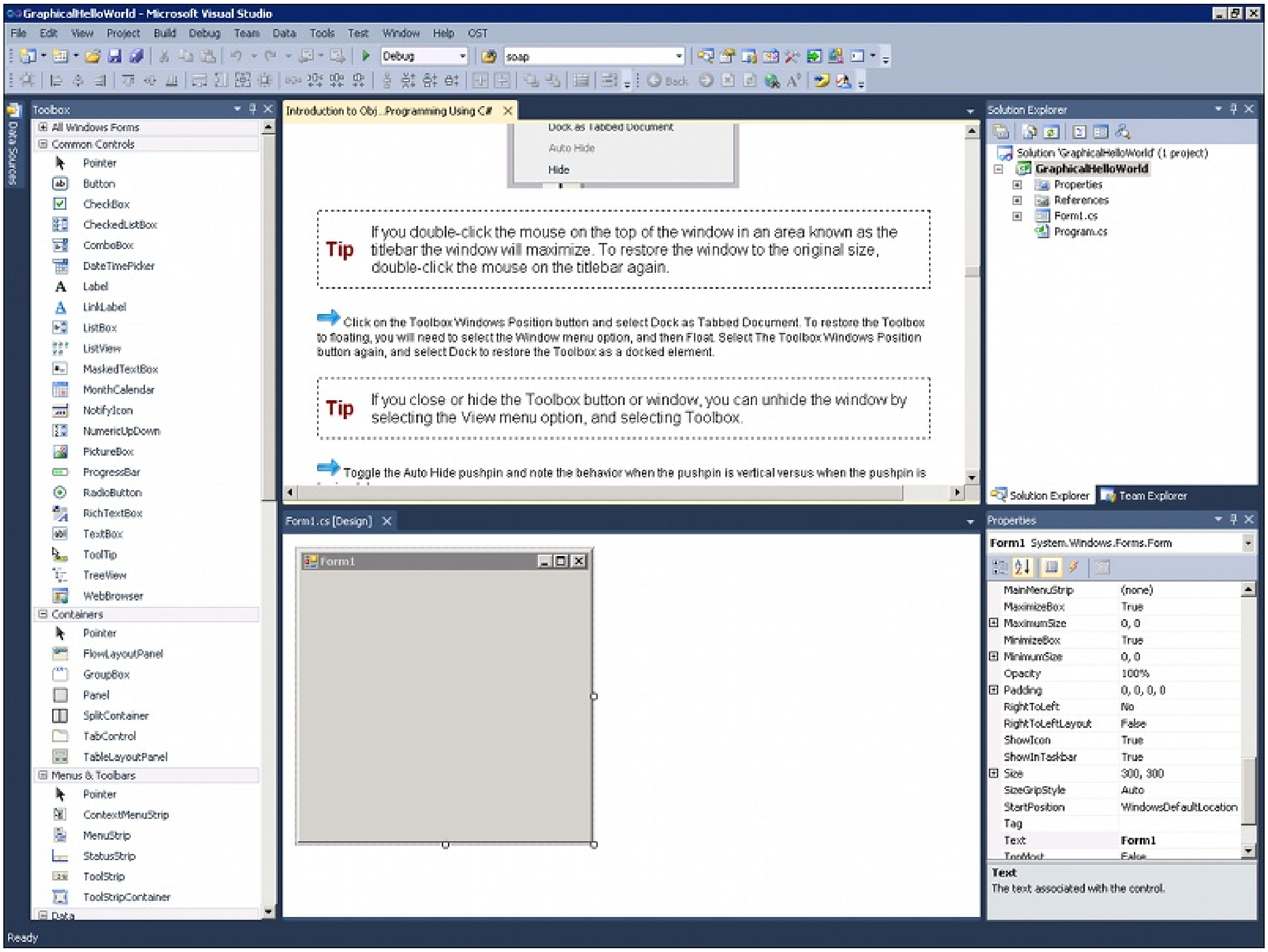
Solution name: GraphicalHelloWorld

☑ Create directory for solution

□ Add to source control

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.



Tip If you double-click the mouse on the top of the window in an area known as the titlebar the window will maximize. To restore the window to the original size, double-click the mouse on the titlebar again.

Click on the Toolbox Windows Position button and select Dock as Tabbed Document. To restore the Toolbox to floating, you will need to select the Window menu option, and then Float. Select The Toolbox Windows Position button again, and select Dock to restore the Toolbox as a docked element.

Tip If you close or hide the Toolbox button or window, you can unhide the window by selecting the View menu option, and selecting Toolbox.

Toggle the Auto Hide pushpin and note the behavior when the pushpin is vertical versus when the pushpin is

Form1 System.Windows.Forms.Form

MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	Form1
TopMost	False

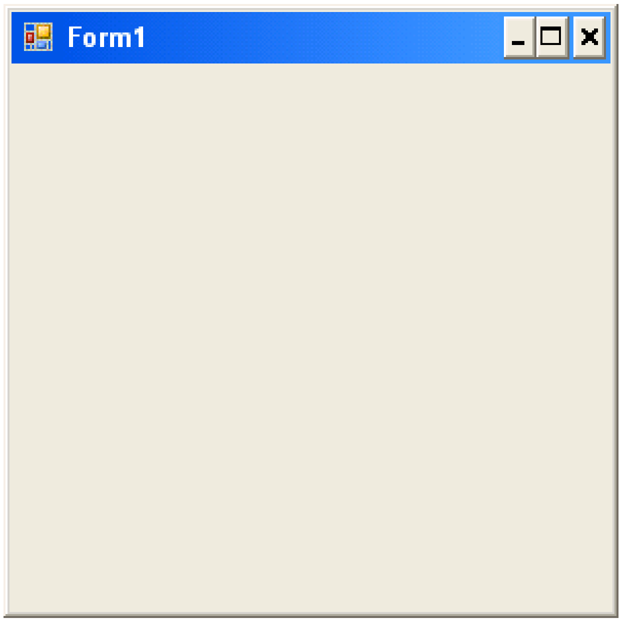
Text
The text associated with the control.

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:



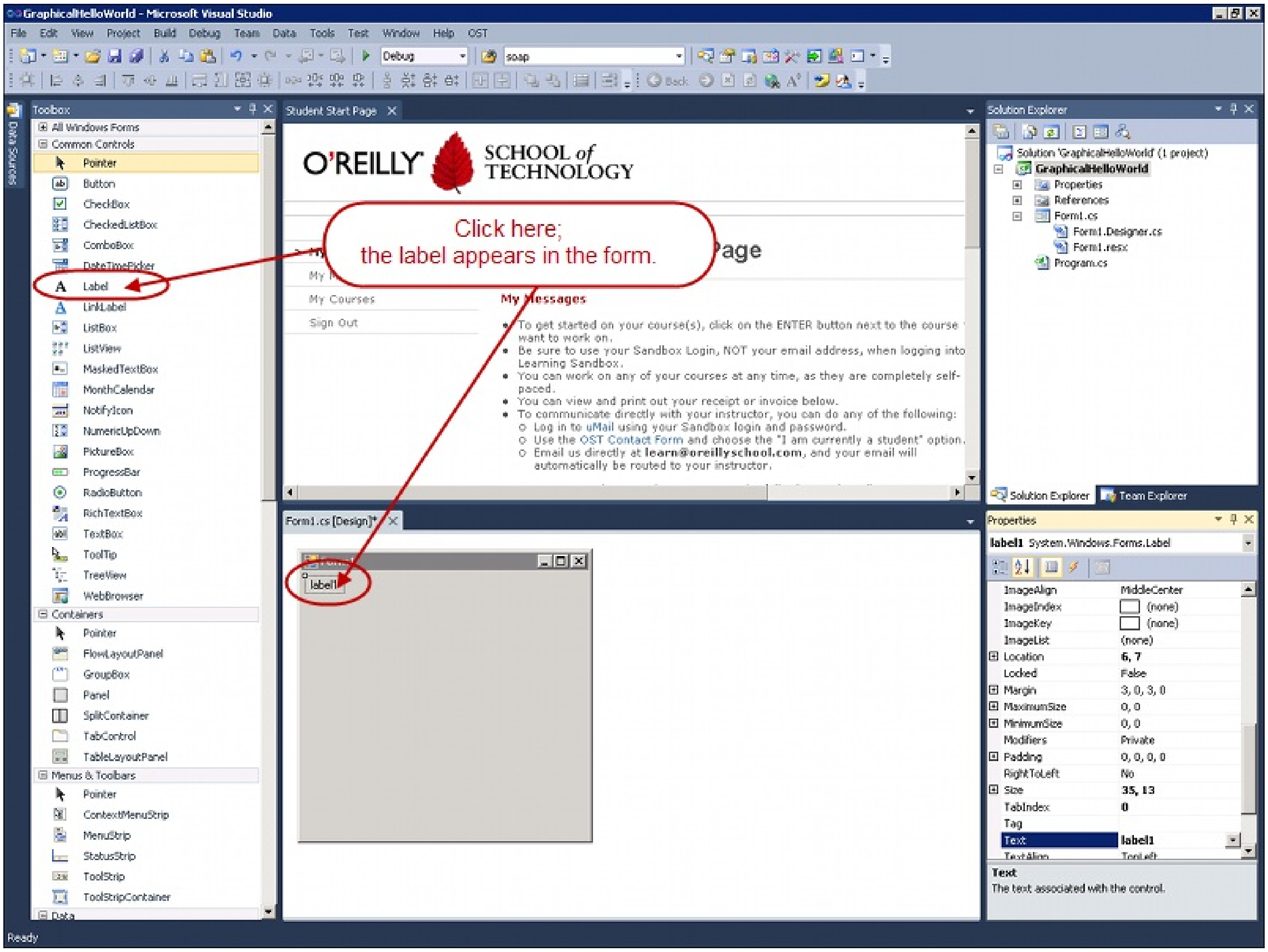
Although this form looks like an independent application, Studio is still connected to it. You can see how Studio is still linked to the running GraphicalHelloWorld application by the choices under the Debug menu.

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.



Click here;
the label appears in the form.

My Messages

- To get started on your course(s), click on the ENTER button next to the course want to work on.
- Be sure to use your Sandbox Login, NOT your email address, when logging into Learning Sandbox.
- You can work on any of your courses at any time, as they are completely self-paced.
- You can view and print out your receipt or invoice below.
- To communicate directly with your instructor, you can do any of the following:
  - Log in to uMail using your Sandbox login and password.
  - Use the OST Contact Form and choose the "I am currently a student" option.
  - Email us directly at **learn@oreillyschool.com**, and your email will automatically be routed to your instructor.

Properties

label1 System.Windows.Forms.Label

ImageAlign	MiddleCenter
ImageIndex	□ (none)
ImageKey	□ (none)
ImageList	(none)
⊞ Location	**6, 7**
Locked	False
⊞ Margin	3, 0, 3, 0
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	Private
⊞ Padding	0, 0, 0, 0
RightToLeft	No
⊞ Size	**35, 13**
TabIndex	**0**
Tag	
Text	**label1**
TextAlign	TopLeft

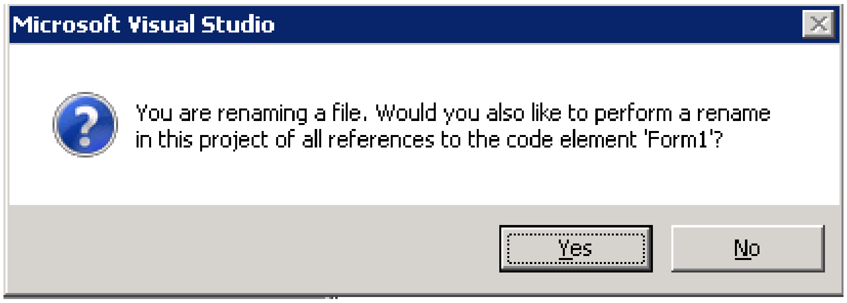
Text
The text associated with the control.

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, GraphicalHellowWorld.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:


Microsoft Visual Studio

You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'Form1'?

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 ☒.

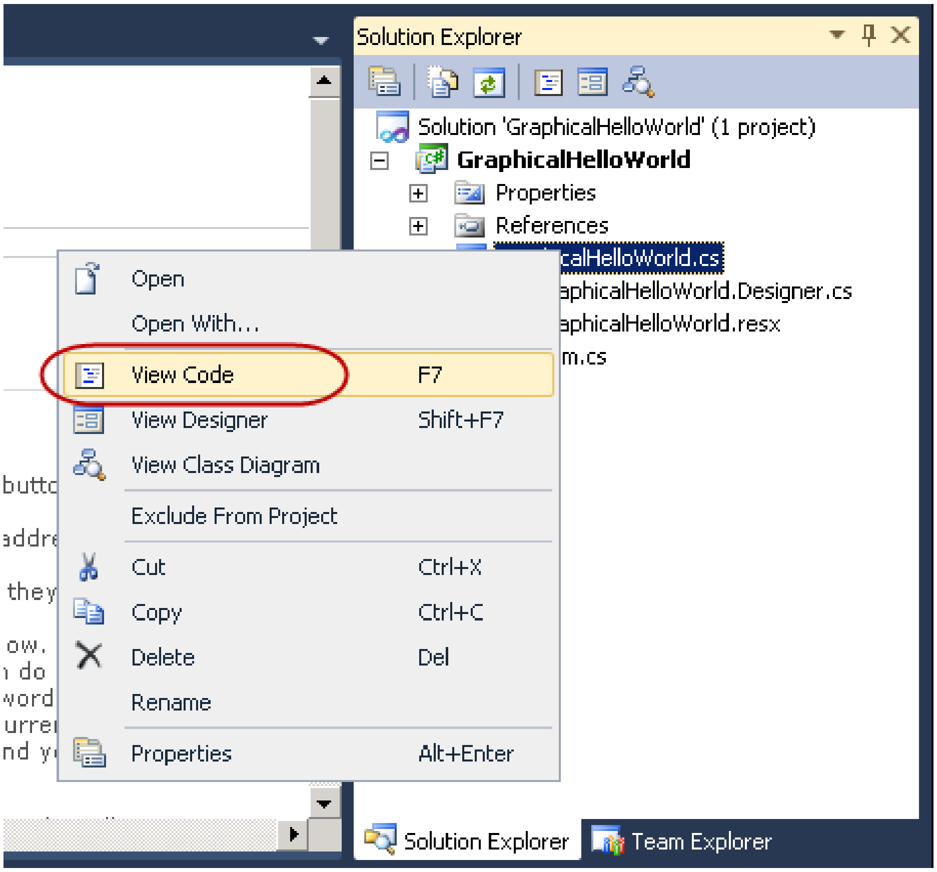
Tip Get in the habit of saving your work. Often, we will pause while we think about doing something. Train yourself to click **Save** or **Save All** during these idle times.

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

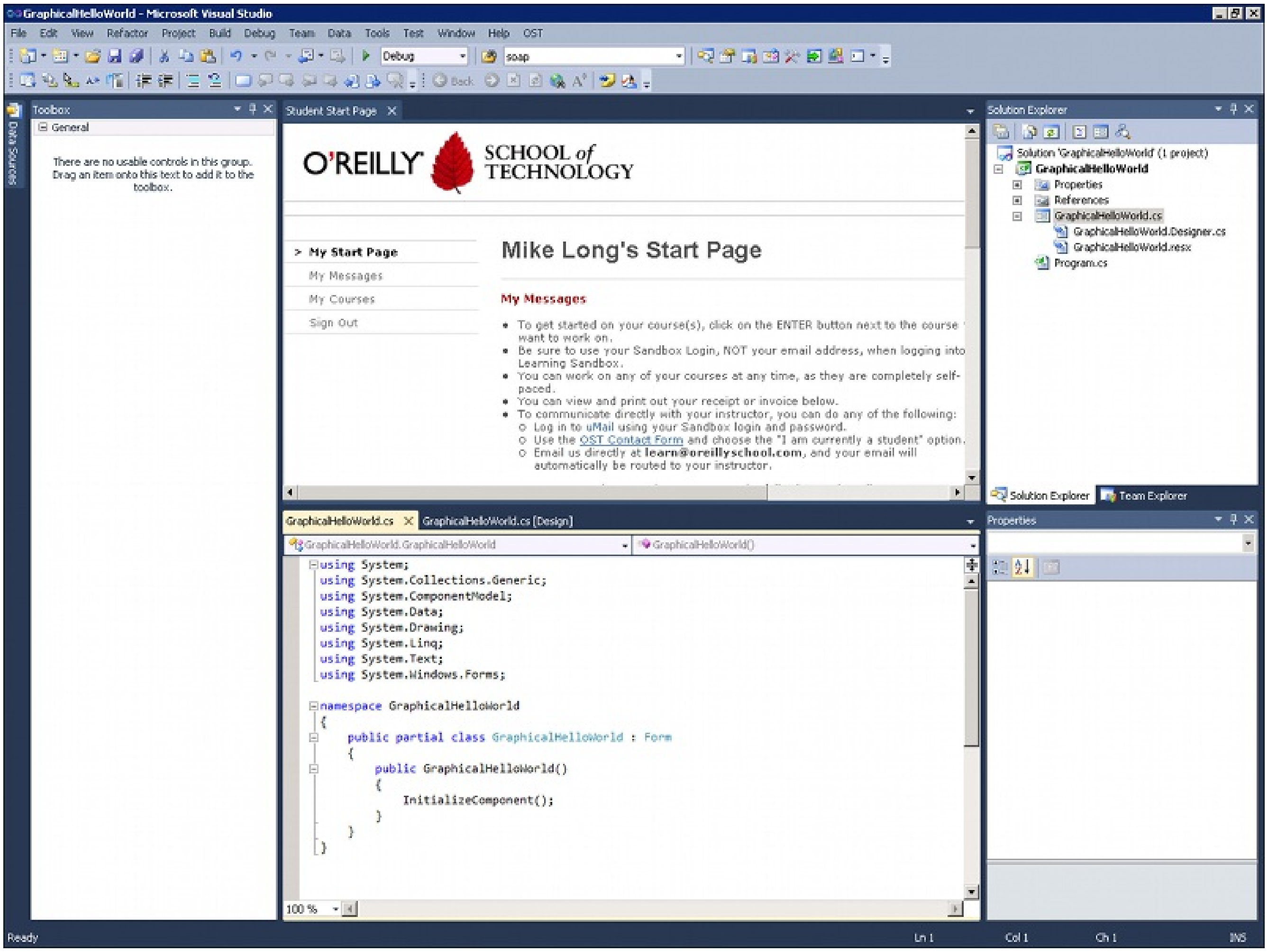
Programming Code

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.



GraphicalHelloWorld - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Test Window Help OST
Debug
soap
Toolbox
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
Student Start Page
Solution Explorer
Solution 'GraphicalHelloWorld' (1 project)
GraphicalHelloWorld
Properties
References
GraphicalHelloWorld.cs
GraphicalHelloWorld.Designer.cs
GraphicalHelloWorld.resx
Program.cs
Solution Explorer
Team Explorer
Properties
GraphicalHelloWorld.cs
GraphicalHelloWorld.cs [Design]
GraphicalHelloWorld.GraphicalHelloWorld
GraphicalHelloWorld()
100 %
Ready
Ln 1
Col 1
Ch 1
INS

> My Start Page
My Messages
My Courses
Sign Out

Mike Long's Start Page

My Messages

- To get started on your course(s), click on the ENTER button next to the course want to work on.
- Be sure to use your Sandbox Login, NOT your email address, when logging into Learning Sandbox.
- You can work on any of your courses at any time, as they are completely self-paced.
- You can view and print out your receipt or invoice below.
- To communicate directly with your instructor, you can do any of the following:
  - Log in to uMail using your Sandbox login and password.
  - Use the OST Contact Form and choose the "I am currently a student" option.
  - Email us directly at **learn@oreillyschool.com**, and your email will automatically be routed to your instructor.

```
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();
        }
    }
}
```

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

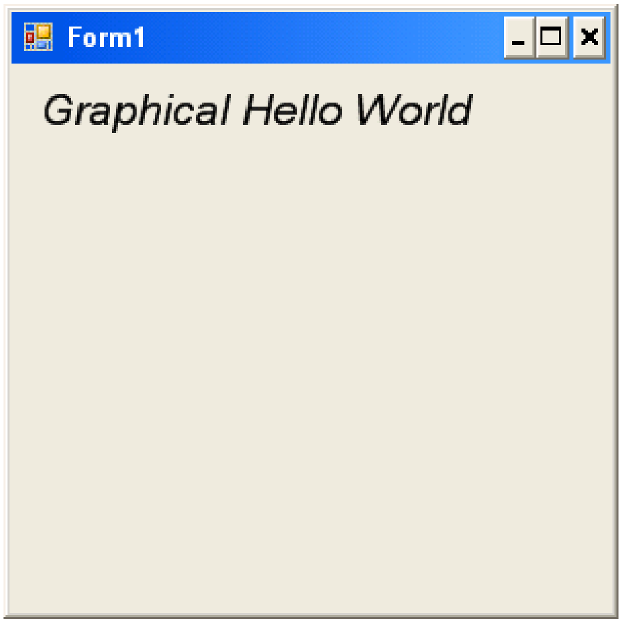
Modify the form code to look exactly like the code shown below (remember: type the highlighted code yourself rather than cutting and pasting!). Note the addition of the line of code below InitializeComponent().

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.

Next

Now that you have completed this lesson, complete any associated projects, then quizzes. Remember, the goal is to understand the material, so don't rush through the projects! Feel free to return to the lesson to review what you have learned, and to refresh your memory.

GUI Components

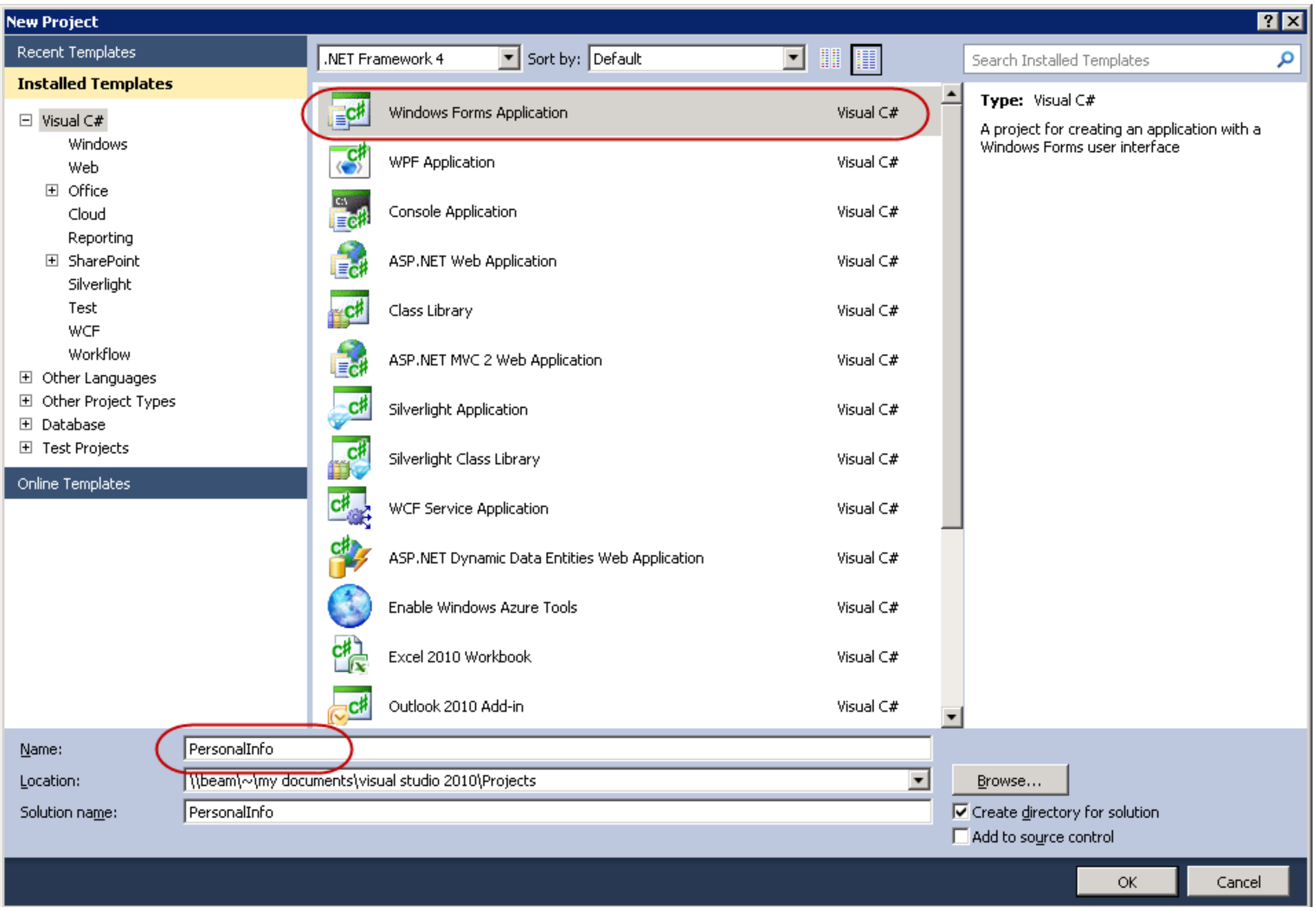
Introduction

Visual Studio, while a software development program, is also a software application. A software application is any computer software designed to help a user perform one or more specific tasks, and typically requires interaction with the user. As we saw in the previous lesson, creating this graphical user interface, or GUI, is an essential step. In this lesson we will learn more about the GUI components available from the Studio Toolbox, how to quickly place and arrange these components, a few basic visual design standards, and finally adding code to make the program do something.

Forms and Controls

At the heart of any Visual Studio application is the Form object. A Form is a container object for holding other controls that will allow the user to interact with the application. In Studio, the Toolbox contains the controls that may be placed on a Form. The list of controls displayed is those controls recognized by Studio on your computer. By default, Studio comes with an extensive list of such controls, and others may be added, including your own created controls.

If you have a project open, select the File menu, then Close Solution to close it. Create a new project (select **File | New | Project**), selecting **Windows Forms Application**, changing the Name to **PersonalInfo** as shown:



.NET Framework 4 | Sort by: Default

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
ASP.NET Web Application	Visual C#
Class Library	Visual C#
ASP.NET MVC 2 Web Application	Visual C#
Silverlight Application	Visual C#
Silverlight Class Library	Visual C#
WCF Service Application	Visual C#
ASP.NET Dynamic Data Entities Web Application	Visual C#
Enable Windows Azure Tools	Visual C#
Excel 2010 Workbook	Visual C#
Outlook 2010 Add-in	Visual C#

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: PersonalInfo

Location: \\beam\~\my documents\visual studio 2010\Projects

Solution name: PersonalInfo

☑ Create directory for solution

☐ Add to source control

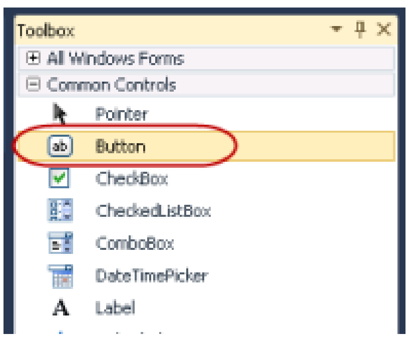
After creating a new Windows Forms Application, Studio presents a view of the four key components: Toolbox, Windows Forms Visual Designer, Solution Explorer, and Properties. We saw this configuration before when creating the HelloWorld application. For this lesson, we will become more familiar with placing controls on the Windows Form object using the Toolbox, and working with the Properties Window.

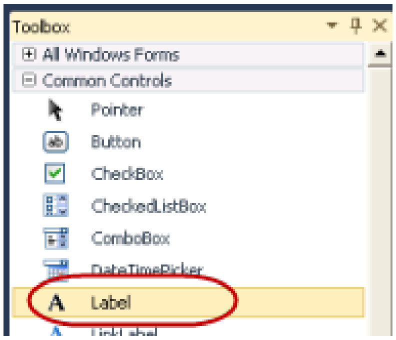
Take a moment to navigate through the Toolbox. Make sure it's "pinned" to the menu by clicking the **Auto Hide** pin icon. Collapse and Expand the Common Control category of the lists by clicking on the minus/plus symbol.

Adding Labels and Buttons

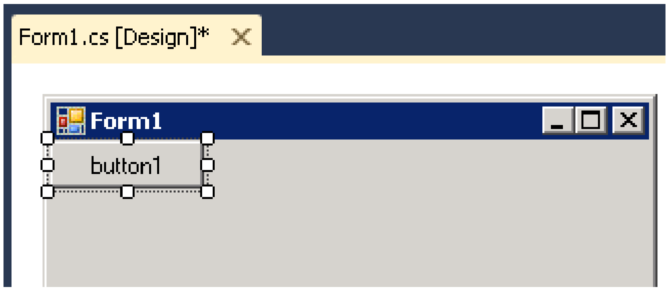
When placing controls on a Windows Form, two primary methods are used: drag-and-drop and double-clicking. Dragging a control from the Toolbox and placing it on the form allows you to place the control exactly where you want it, while double-clicking on a control will place it in a default position on the form in a cascading arrangement. The double-clicking method provides a technique of adding multiple controls quickly.

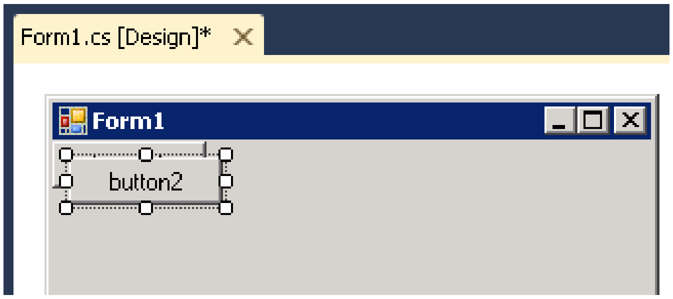
Double-click the **Button** Common Control. Double-click the **Button** control again, and then double-click the **Label** Common Control.

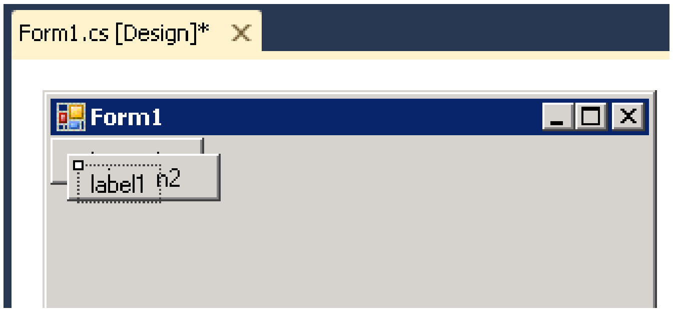




Note that each control is placed at the top-left corner, on top of the preceding control. See images below.







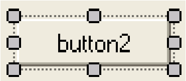
When adding controls using the double-click technique, adding the same control a number of times works well, provided that you then move the controls to a position away from the top left position where new controls are added.

Click and drag the **label1** Label control away from the button2 Button control.

While dragging the label1 Label control, note that the cursor changes to a mouse pointer, and that blue alignment bars appear to assist you in aligning the label1 control with other visual elements. The label1 control will also have a dashed box around it indicating that this control has the focus. When the label1 control is not being moved, floating the cursor over the control will display a four-direction cursor ✥ indicating that if you click on the control you will be able to move the control.

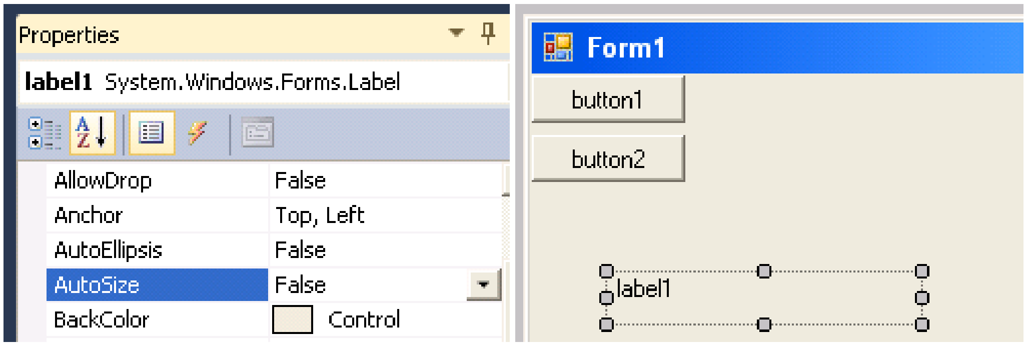
Click the button2 Button control and move it down, aligning it with the button1 Button control.

When selected, a Button control displays eight gray resize points:



Most Common Controls may be resized to a desired size. The Label control may also be resized, but did not have any visible resize points because by default a Label control has an AutoSize property set to True, which automatically expands the control to display the Text property contents.

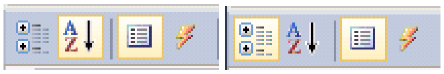
Select the label1 Label control. Navigate the Properties scrollbar in the lower right of Studio to locate the AutoSize property. Click the True property to the right of the AutoSize property, and select **False** from the drop-down. Move the mouse cursor over the different gray resize points. The cursor will change depending on which resize point you pass over. Click the center-right resize point and drag it right to resize the label:


Properties

label1 System.Windows.Forms.Label

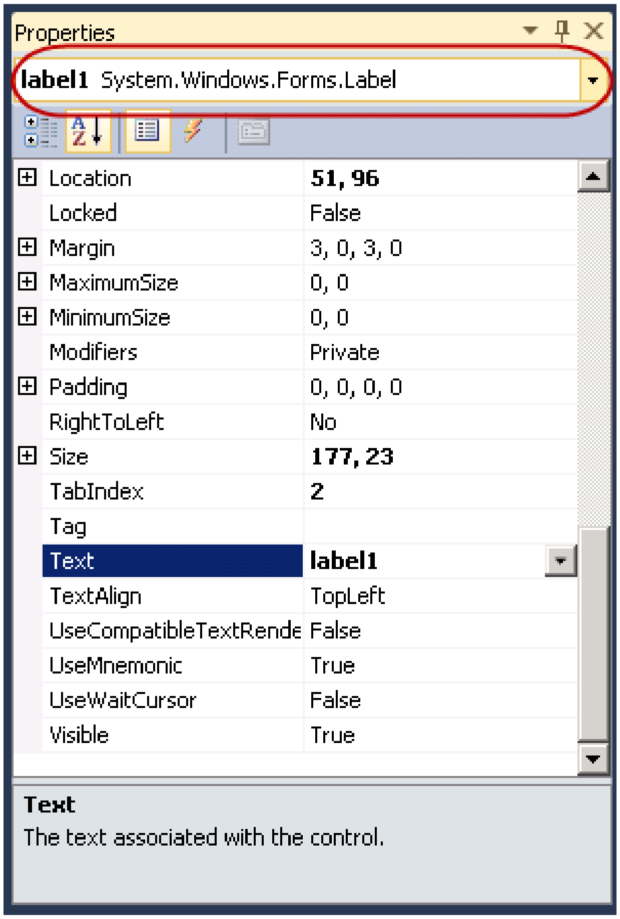
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	False
AutoSize	False
BackColor	Control

At the top of the Properties Window are a number of buttons that control what is displayed in the Properties Window: the order and grouping of the properties or events, and whether the Properties Page or Events Page is displayed.



You can sort Properties in the Properties Window alphabetically or by category. The first button selects the Categorized view of the Properties. The second button selects the Alphabetical (default) view of the Properties. To see a reminder of the buttons' functions, hover the mouse pointer over each button. In the left image above, the Alphabetical view is selected; in the right image, the Categorized view is selected. The third and fourth buttons are toggles. The third button selects Properties. The fourth button, a lightning bolt, selects Events—we'll cover this later.

Tip Remember that Studio is context-sensitive. If you do not see the properties for the control you want to change, make sure you have selected the correct control. Above the Properties Window button is a drop-down that indicates the currently selected control or component. You may also use this drop-down to select a different control or component.


Properties

label1 System.Windows.Forms.Label

⊞ Location	**51, 96**
Locked	False
⊞ Margin	3, 0, 3, 0
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	Private
⊞ Padding	0, 0, 0, 0
RightToLeft	No
⊞ Size	**177, 23**
TabIndex	**2**
Tag	
Text	**label1**
TextAlign	TopLeft
UseCompatibleTextRende	False
UseMnemonic	True
UseWaitCursor	False
Visible	True

Text
The text associated with the control.

Make sure the **label1** control is selected. In the Properties Window, use the Active Element drop-down to select the **button2** control. The button2 control will now have the focus in the form.

Save your project.

Naming

When creating a new Windows Forms Application, by default the new form is named Form1. Most applications will have multiple forms, in addition to many controls and other elements we will discuss later, and selecting appropriate names for these objects will greatly assist in identifying the element and in writing your code. The rules we use to name elements are known as naming conventions, guidelines, or standards. We will also learn later about user interface design standards, as well as programming standards.

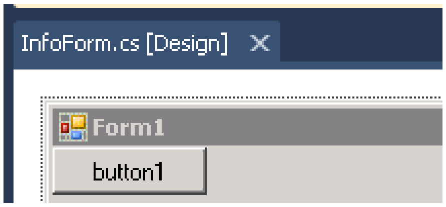
Primarily, two different naming conventions for forms and controls are used: *Pascal* and *Camel*. The Pascal convention capitalizes each significant word and omits spaces. For example, for an element that defines a last name using Pascal, you would use **LastName**. The Camel convention is identical to Pascal except that the first letter is lower-case, so rather than **LastName**, the element would be **lastName**.

As an additional naming convention, when working with controls and other visual components, such as forms, indicating the type of control or component is also very helpful. So far, we have used a Form, a Label, and Button. We can indicate the type of control we are dealing with by adding the type to the end of the variable name, such as: **InfoForm**, **firstNameLabel**, and **exitButton**, respectively. For example, to name a Button that updates an image would use the Camel convention and be named **updateImageButton**. With Intellisense, the need for this has decreased, but we will use these naming conventions to help clarify the elements you will be using. Also, it is a commonly accepted naming convention for most C# programmers.

Select the Form1.cs entry in the Solution Explorer. In the Properties Window, find the File Name property, and change the value from Form1.cs to **InfoForm.cs**. A refactor dialog box may appear asking you if you want to change all references to the code element Form1. Click **Yes** to update the Form1 references to now reference InfoForm.

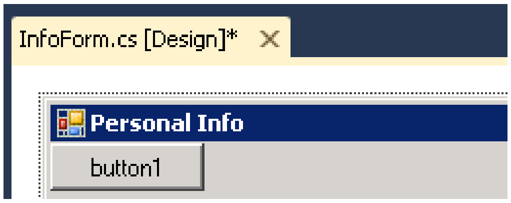
Note InfoForm.cs is the C# backing file that holds the code for the form. A form is a *class* in C#, so we name it using the Pascal case, or *Title case*, naming convention.

Following the Form name change, the Visual Editor tab shows the new InfoForm.cs:



We do have more renaming to do, but first, notice the blue area at the top of the Form that has the words Form1. This blue area, or title bar, can be changed.

Select the Form1 title bar to make sure you have the entire Form selected. In the Properties Window, locate the Text property, and change the value to **Personal Info**. Note that it changes in the title bar in the Visual Editor:

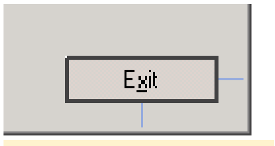


Changing the title bar of a Form to something meaningful is important to help users understand the purpose of a Form, and to distinguish one Form from another. Now, let's continue renaming the other controls.

We have changed the Text property a number of times, for a Label and a Form. The Text property is the text displayed on an element. This visual text differs from the name of the element. The names you choose should be meaningful. Oh, and no spaces in the names! To set the name of an element, you change the Name Property.

In the Visual Editor, select button1. The Properties Window should be displaying the Properties alphabetically. Scroll to the very top of the properties list, and change the Name property to **exitButton**. Scroll down to the Text property, and change button1 to **E&xit**. Now, move this Exit button on the form to the lower-right corner, leaving a margin around the

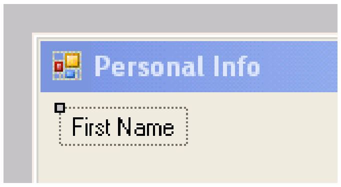
Button by using the blue spacing bars that appear as you get close to the corner of the form. See the image below.



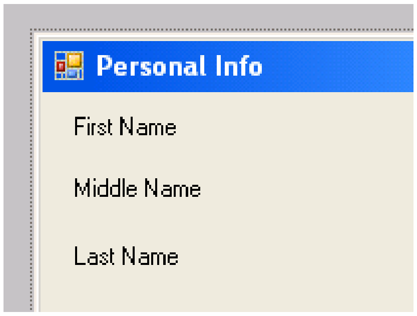
The **&** in the exitButton Text Property is not an error. The & indicates which letter of the Text Property will act as a keyboard accelerator, or shortcut. Accelerators are typically enacted using the **Alt** key with the indicated letter. In the case of the exitButton, the accelerator is **Alt+x**. Visually, the lowercase x in the Button Text property is underlined.

Note The underlining may not be visible on all operating systems. Windows Vista and 7 may suppress the keyboard accelerators, but you may be able to restore them using the Ease of Access Center in the Windows Control Panel.

Select the button2 control, and press the **Delete** key to remove it from the form. Move the label1 Label to the top-left of the form, again using the blue bars to leaving a margin around the Label. Change the label1 Name property to **firstNameLabel**, and the Text property to **First Name**. Change the AutoSize property back to **True**:

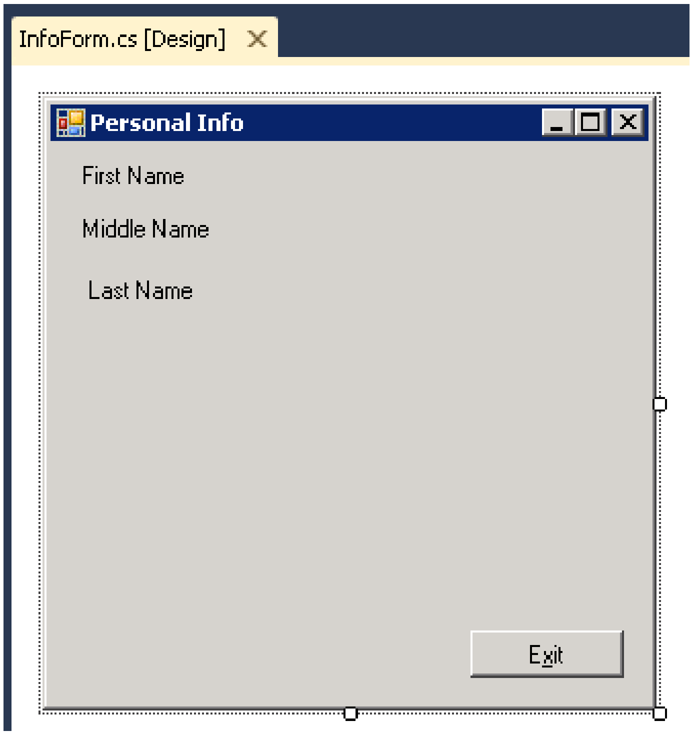


Add a new Label under the firstNameLabel, aligning the left side of the Label using the blue alignment bars. Change this new Label Name property to **middleNameLabel**, and Text property to **Middle Name**. Add a third new Label under middleNameLabel and align it, changing the Name property to **lastNameLabel** and Text property to **Last Name**.



Have you saved lately?

The complete form so far is shown in the image below.



Tip As you work with Studio elements such as forms and controls, and learn to write C# code, try to follow the commonly accepted naming conventions to make your code more readable to other programmers. However, you may be required to follow specific corporate standards of your employer. Whichever pattern you develop or follow, be consistent. Whenever you create or add new Forms and add controls, name the Forms and controls before you begin working with them and making modifications.

Formatting Controls

While the blue alignment and spacing bars are extremely helpful when placing controls and other elements on a Form, you may need to further adjust element placement. You may always drag a user interface (UI) element, but this technique may be tedious or inexact. The Format menu will help, containing aids to aligning, spacing, and sizing one or more UI elements. Some aids will only work when you have two or more elements selected.

Tip To select multiple controls, use the **Ctrl** key. You may also click and drag to create an expanding (also known as rubber banding) selection box. When you release the mouse button, any item within the box will be selected. Yes, you may select more than one item. You may even change item properties of all of the items selected, although not all Properties will be displayed, but only those that overlap between all of the selected elements.

Select the firstNameLabel Label, and then select **Format | Center in Form | Horizontally** in the top menu. The Label will now be centered. Select the middleNameLabel Label, then, holding down the **Ctrl** key, select the lastNameLabel. Both controls should now be selected. Center both controls horizontally in the Form simultaneously using the **Format** menu.

Tip Just as with most Windows applications, Studio includes the ability to Undo previous actions using **Edit | Undo** in the top menu. You can also **Redo**.

Select all three Label controls by clicking above and to the left of the firstNameLabel, holding the mouse button down, and dragging the selection window until all three Label controls are at least partially within the rubber banding box. Select the **Format | Align | Middles**. All of the Label controls are stacked on top of each other, not what we wanted, so select **Edit | Undo** (or press **Ctrl+Z**, the standard shortcut. Select **Format | Align | Centers**. The labels are now centered in the form. Finally, align the labels using **Format | Align | Lefts**, then click and drag any one of the selected labels back near the left side of the form until the blue spacing bar appears.

Note When selecting multiple controls for formatting, the first control selected is the anchor control. In other words, if you were left-aligning controls, the left margin of the first control selected is the left margin that would be used for the other selected controls.

The Labels are now left-aligned to each other, but may not be evenly spaced (the vertical space between each Label may be different). Let's fix that.

Select all three Labels again, if they are not still selected. Select **Format | Vertical Spacing | Make Equal**.

Note Most of the options in the **Format | Align** menu are also available as icons in the **Layout** toolbar.

Feel free to experiment with the other control formatting options, remembering you can always **Undo**, or just move the controls to where you want them.

Remember to save your form!

Adding Input Textboxes

Previously, we used Label controls to hold information, but Label controls are typically used to describe other input controls, display information that users cannot change, or display a status. To gather textual information from a user, we may use a TextBox control.

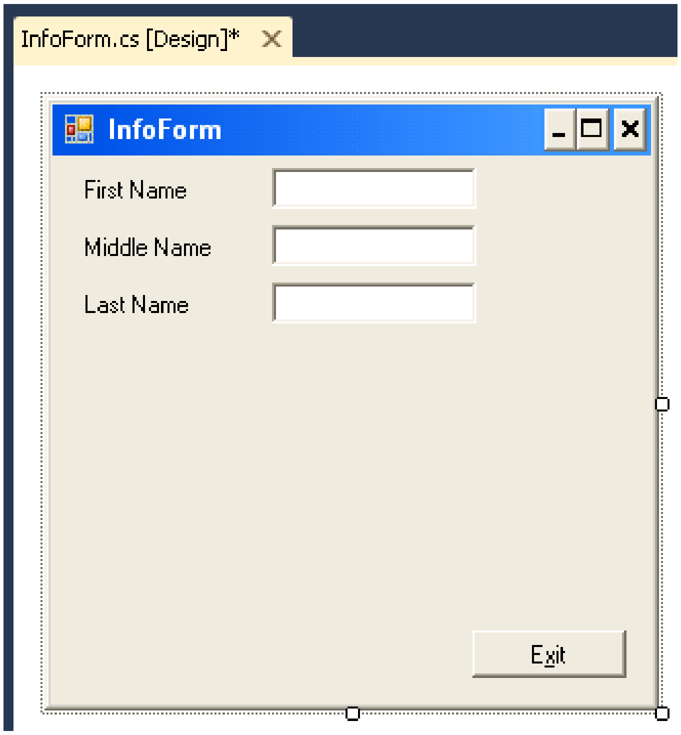
Add three TextBox controls using the drag-and-drop method from the Toolbox, placing each to the right of each of the First, Middle, and Last Name labels. Select all three TextBox controls and left-align them if you did not align them using the blue alignment bars.

Tip You may have noticed additional purple alignment bars. You may use those bars to align the bottom of the TextBox text with the Label text. Otherwise, you may use the Format menu to select a Label and the corresponding Textbox, and select Align Middles, although the Align Middles refers to the middle of the controls. The purple Text alignment bars ensure the Text of one control is bottom-aligned with the Text of another control, and may not correspond to the actual middle of each control.

As with the Label controls and Form, we need to change the names of the TextBox controls.

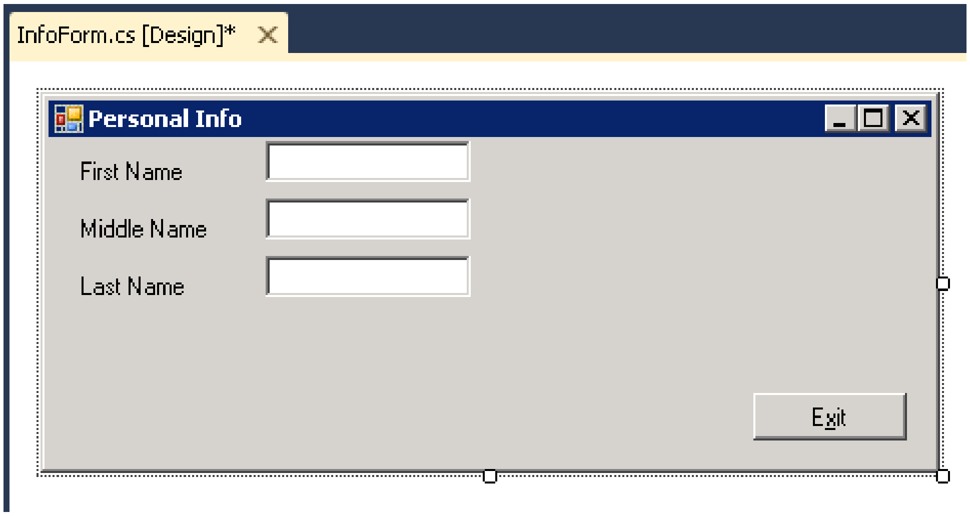
Change each TextBox Name property to match the Label Name property, changing the Label suffix in each Name to **TextBox**. When you finish, you should have TextBox controls named **firstNameTextBox**, **middleNameTextBox**, and **lastNameTextBox**.

In the image below, note the asterisk next to InfoForm.cs [Design]. Yes, it's time to save your changes again!



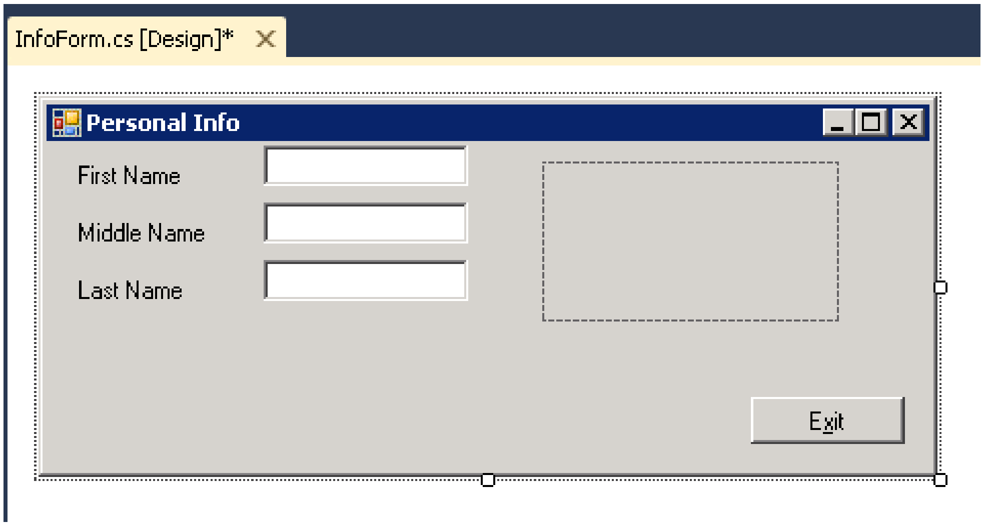
The size of a form may be changed, making it larger to hold more controls, or smaller if the form contains only a few controls.

Move the mouse over the right edge of the form, and click and drag the form edge to make it slightly wider. Move the Exit Button up, and then click and drag the bottom edge of the form to reduce its height, as shown:

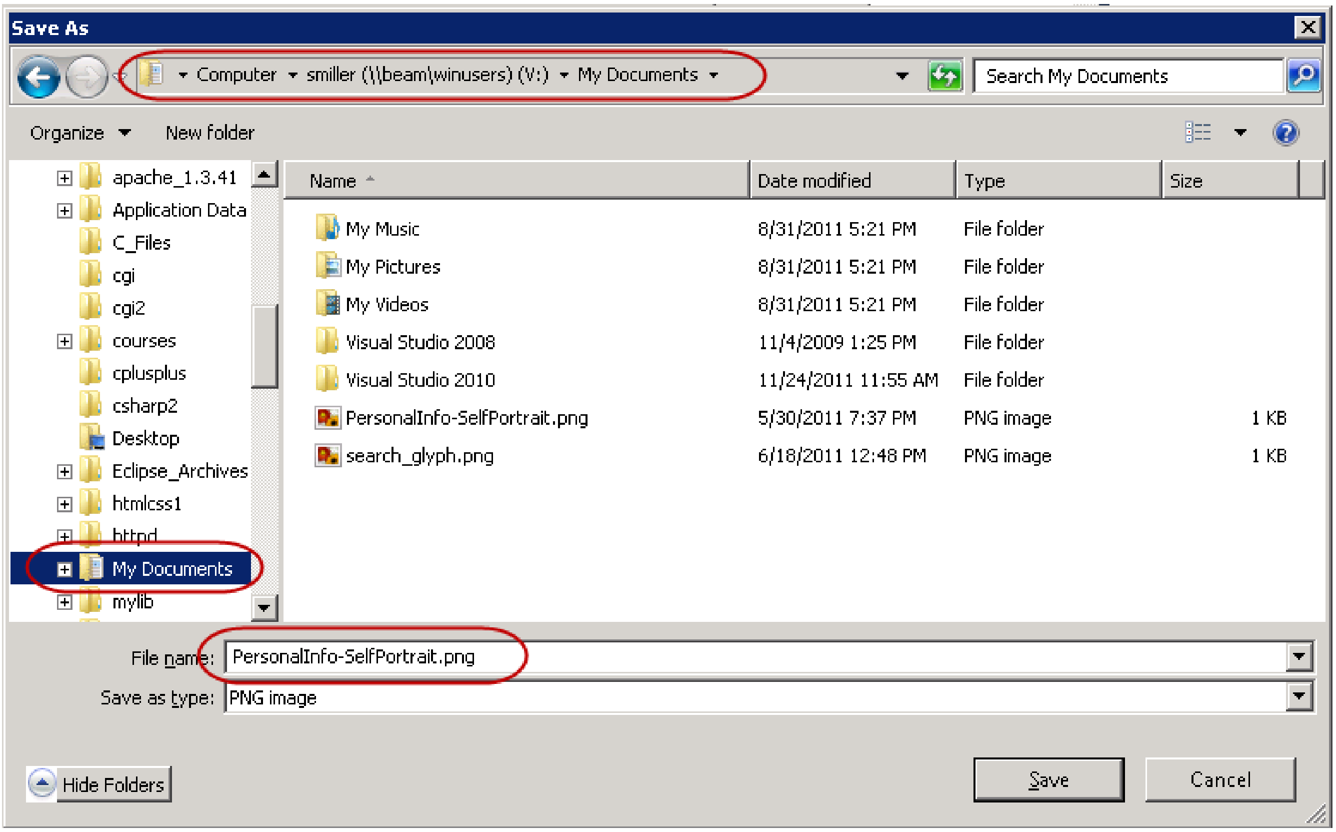


Why the strange size? Next, we will add the course author's self-portrait using a PictureBox control.

Drag a PictureBox control from the Toolbox, and position it to the right of the three Textbox controls. Change the size of the PictureBox control using the Size Property, entering **145, 78** (for the width and height, respectively). Set the Name of the PictureBox control to **myPictureBox**:



We need to display an image in the PictureBox. We have created a lovely portrait, **PersonalInfo-SelfPortrait.png**, for you to use. To get the image, right-click this link and select **Save Target As...**. Save the file in your **My Documents** directory:



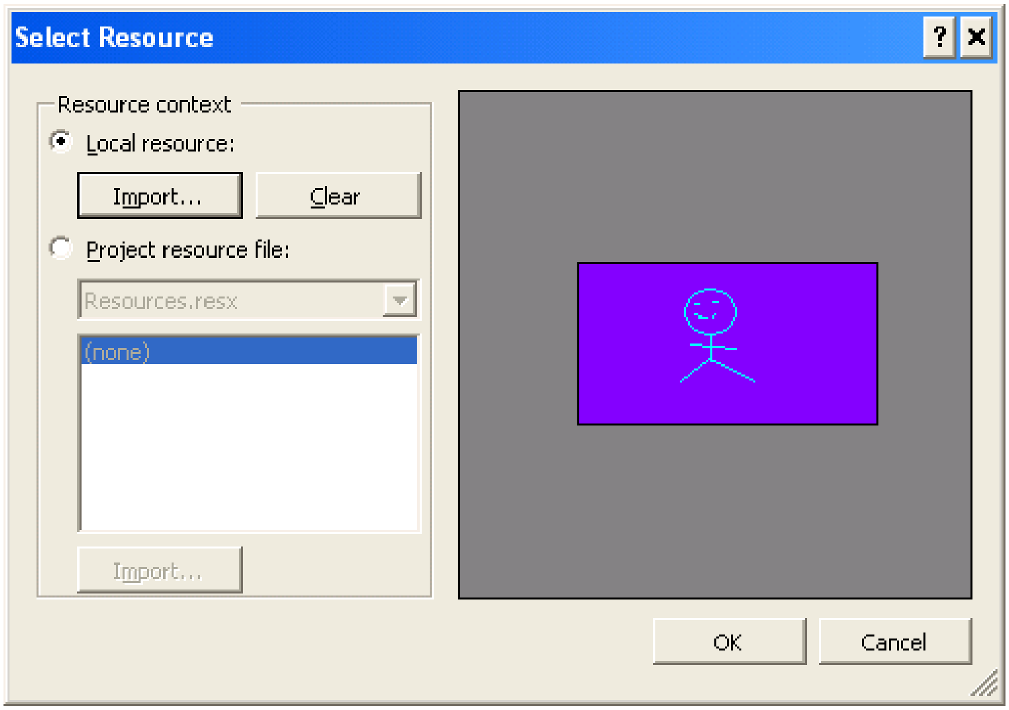
Name	Date modified	Type	Size
My Music	8/31/2011 5:21 PM	File folder	
My Pictures	8/31/2011 5:21 PM	File folder	
My Videos	8/31/2011 5:21 PM	File folder	
Visual Studio 2008	11/4/2009 1:25 PM	File folder	
Visual Studio 2010	11/24/2011 11:55 AM	File folder	
PersonalInfo-SelfPortrait.png	5/30/2011 7:37 PM	PNG image	1 KB
search_glyph.png	6/18/2011 12:48 PM	PNG image	1 KB

File name: PersonalInfo-SelfPortrait.png

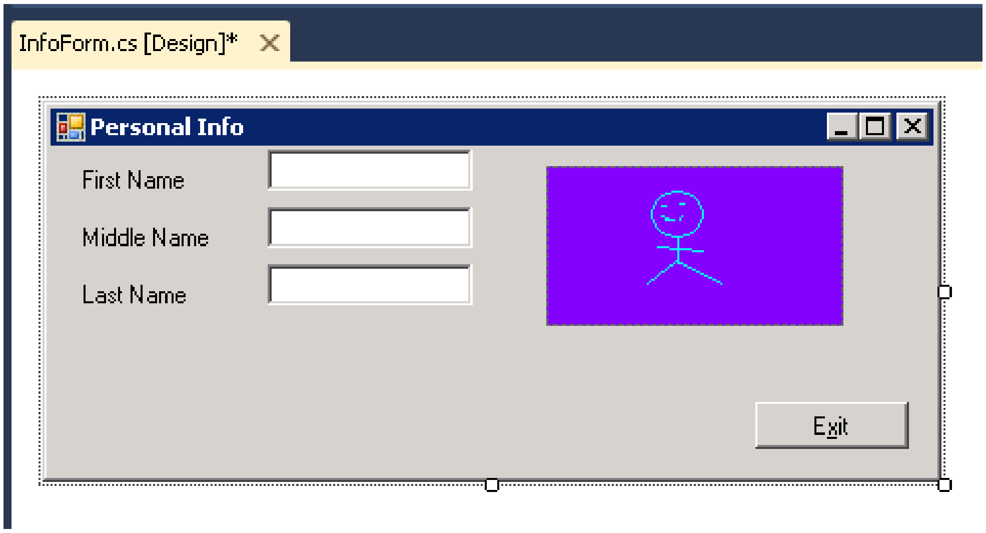
Save as type: PNG image

Note In other lessons, we'll have you do this same thing with some other images. Refer back to this lesson if you need a reminder on how to do this.

Select the myPictureBox control, then select the Image Property of the PictureBox, and click on the **...** (or ellipsis), to bring up the Select Resource dialog box. Select the **Local Resource** radio button, then click **Import**. In the Open File dialog box that appears, select the file **PersonalInfo-SelfPortrait.png**, and click **Open**:



Click **OK** in the Select Resource dialog box to import the author's self-portrait into the myPictureBox control. Your completed form should look like this:



An uncanny likeness, no? Remember to save your masterpiece!

Standards

Previously, we discussed two naming conventions for controls and forms. These naming conventions are part of the concept of standards. Standards represent an agreed-upon and repeatable way for doing just about anything. For software, standards include how we program, how the graphical user interface appears, how a user moves from one screen to another, how to describe what a program is supposed to do, and more.

One of the most interesting facts about standards is the simple fact that while many standards remain the same, many change. While this lesson, and future lessons, do not intend to create a comprehensive list of standards, just as with the naming conventions mentioned earlier, we will discuss standards as they apply to the lesson material. Different organizations require adherence to their standards (academic institutions, employers, open source groups). Also, Microsoft as a company recommends standards when using their development environment, and for the appearance of a product targeted for different versions of the Windows operating system.

As we dig more into programming, code standards will be discussed. For now, let's introduce four basic principles that should be considered when designing a user interface. By the way, you can read this information yourself online at Microsoft: The Basic Principles of Proper UI.

Spacing and Positioning

As we saw earlier, when placing components on a Form, the Visual Editor prompts you for proper spacing using blue lines, and alignment with light purple lines. In general, provide sufficient spacing between controls, align controls with other related controls, make spacing equal between different elements, and make the overall design balanced.

Size

Just as with spacing and positioning, Studio helps you with determining the size of a control or element. If you drag a control from the Toolbox onto the Visual Editor, the control or element appears at the recommended default size.

Grouping

Modern applications require an abundance of controls. Grouping these controls based on related function is essential. Also, we'll learn about other controls offered by Studio to assist in grouping these related controls, such as Tabs and Panels. These grouping controls can be found collected on the Container tab in the Toolbox.

Intuitiveness

Great, a section on intuition? Well, no, this section is about making your user interface intuitive to the user. How? Using well-matched colors—such as green for starting, yellow for warning, and red for stopping or critical elements—can help. Also, selecting wording that makes sense to the user, and not just to a technical programmer, and using a common placement for familiar controls and buttons (such as OK, or Cancel).

For now, we'll leave this discussion of standards, but we'll return to it as the lessons continue.

Making the Program Do Something

So far, we've added a number of controls to our Personal Info Form, and even added an image of the course author (he needs a shave!). Now, we'll add C# code that enables the **Exit** button to exit the program, and that animates the image.

Select the **InfoForm.cs [Design]** Visual Editor tab, then double-click the **Exit** button.

Double-clicking a Button in the Visual Editor automatically opens the C# Code Editor, and places your cursor inside an *event handler* for the Button. This event handler is the code that will execute or "fire" when the Button is clicked.

Modify the code in the exitButton_Click event handler as indicated in blue below.

CODE TO TYPE: exitButton_Click

```
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 PersonalInfo
{
    public partial class InfoForm : Form
    {
        public InfoForm()
        {
            InitializeComponent();
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Call the Form Close method
            this.Close();
        }
    }
}
```

Let's discuss how this code works. We'll omit some of the code in this OBSERVE box so we can focus on the code we're most interested in.

OBSERVE: Adding Exit Code

```
.
.
.
namespace PersonalInfo
{
    public partial class InfoForm : Form
    {
        public InfoForm()
        {
            InitializeComponent();
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Call the Form Close method
            this.Close();
        }
    }
}
```

We have added an event handler that handles the click of the Exit button by adding the **exitButton_Click** method (we will discuss methods in the next lesson), that will tell the Form to close. The **this** refers to the **InfoForm** Form, and is used to call the **Close()** method. We include a **comment** that helps explain what the code is doing.

Save your project.

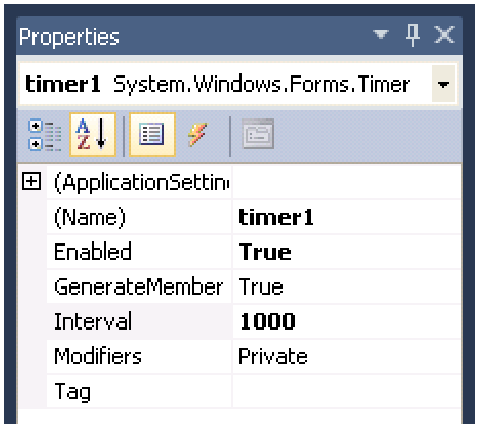
Select the green Run/Debug button from the Studio button bar to test your changes. Click the **Exit** button rather than the upper-right "X" to close the program and return to Studio.

Next, we'll animate the image using a Timer. A Timer is a non-visual control (you add it to the Form, but it's not visible to users) that creates a predictable event that can be used to make something happen. In our case, we will make the image "flip" 180 degrees around the y-axis every second. The Timer control measures time in milliseconds; 1000 milliseconds is equal to 1 second.

Select the **InfoForm.cs [Design]** tab to select the Visual Editor. Expand the **Components** section of the Toolbox, and double-click the **Timer** control .

The Timer control appears at the bottom of the Visual Editor, named timer1. Now we need to turn "on" the Timer, and change how often the Timer will "fire."

Select the **timer1** control, change the Interval Property in the Properties Window to **1000** (for 1000 milliseconds), and the Enabled Property to **True**, as shown below:



timer1 System.Windows.Forms.Timer

⊞ (ApplicationSettin	
(Name)	**timer1**
Enabled	**True**
GenerateMember	True
Interval	**1000**
Modifiers	Private
Tag	

Note After our discussion on naming conventions, why didn't we rename the Timer control? Pure laziness (and to be able to make a point)! Often, controls aren't referenced (such as Label controls), or you may only have a single control (in the case of the Timer), so leaving the default Studio name isn't unusual. It's a good programming practice to rename the Timer control. What name would you give it? We'll leave the default **timer1** name for now—to see what a better name for the Timer control might be, you'll have to keep reading.

Now that we've added and enabled the Timer control, we need to add code to respond to the Timer "firing" every second. We need to create a handler for this "firing" event, and within this handler, add code to do the image "flip" to create the animation.

Double-click on the timer1 Timer control to create an event handler for the firing of the Timer. Studio will take you to the Code Editor for this event. Modify the existing code as shown below.

Note Here's where the Intellisense autocompletion feature can really save you a lot of typing!

CODE TO TYPE: InfoForm

```
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 PersonalInfo
{
    public partial class InfoForm : Form
    {
        Bitmap tempBitmap;    // Temporary bitmap object for flipping

        public InfoForm()
        {
            InitializeComponent();

            // Set the temporary bitmap initial image to the
            // image in the PictureBox
            tempBitmap = (Bitmap)myPictureBox.Image;

        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Call the Form Close() method
            this.Close();
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            // Make sure the temporary bitmap image is not empty
            if (tempBitmap != null)
            {
                // Flip the temporary image 180 degrees on the y-axis
                tempBitmap.RotateFlip(RotateFlipType.Rotate180FlipY);
                // Update the PictureBox image with the flipped temporary image
                myPictureBox.Image = tempBitmap;
            }
        }
    }
}
```

-->

Save your project.

Select the green Run/Debug button from the Studio button bar. Observe the cool changing image! To close the program and return to Studio, click the **Exit** button.

Let's discuss how this code works.

OBSERVE: Adding Image Animation Code

```
.
.
.
namespace PersonalInfo
{
    public partial class InfoForm : Form
    {
        Bitmap tempBitmap;    // Temporary bitmap object class variable for flipping

        public InfoForm()
        {
            InitializeComponent();
            // Set the temporary bitmap initial image to the
            // image in the PictureBox, using Bitmap cast to ensure
            // same type
            tempBitmap = (Bitmap)myPictureBox.Image;
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Call the Form Close() method
            this.Close();
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            // Make sure the temporary bitmap image is not empty
            if (tempBitmap != null)
            {
                // Flip the temporary image 180 degrees along the y-axis
                tempBitmap.RotateFlip(RotateFlipType.Rotate180FlipY);
                // Update the PictureBox image with the flipped temporary image
                myPictureBox.Image = tempBitmap;
            }
        }
    }
}
```

To animate the author image self-portrait, we added code that flips the image 180 degrees on the y-axis. To flip the image, we use a temporary Bitmap (Bitmap **tempBitmap**). When the **InfoForm** is loaded, the **InfoForm** method is called. Within this method, we initialize the **tempBitmap** temporary Bitmap object with the image stored in the **PictureBox**. When the Timer "fires," we call the code in the **timer1_Tick** method. Before attempting to "flip" the image in the temporary **tempBitmap** object, we check to make sure that the object is not empty (if (**tempBitmap** != null)). We then "flip" the image calling a method that knows how to manipulate an image (**tempBitmap**.**RotateFlip(RotateFlipType.Rotate180FlipY)**). After flipping the image in **tempBitmap**, we change the image in the PictureBox to this new "flipped" image (myPictureBox.Image = **tempBitmap**). Note the use again of "//" comments to help explain what is happening.

Note So what was a better name for the Timer control rather than timer1? I'd suggest **animationTimer**.

Coding Mistakes

Have you made a mistake yet while typing in any of the lesson code? If not, congratulations! Nevertheless, making a mistake is normal. In fact, we often learn more from our mistakes than from being perfect. So, let's make a mistake, and see how Visual Studio documents the mistake, and what we can do to fix the problem.

In the Visual Code Editor, double-click on the **Exit** button. The Visual Editor for the InfoForm Form opens, with the cursor at the first line of code of the exitButton_Click event handler for the Exit button. Change the code below in the LISTING block by removing the parentheses after this.Close:

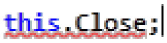
CODE TO EDIT: InfoForm exitButton_Click

```
.
.
.
  private void exitButton_Click(object sender, EventArgs e)
  {
      // Call the Form Close() method
      this.Close~~()~~;
  }
.
.
.
```

The code should now look like this:

OBSERVE: Changed exitButton_Click method

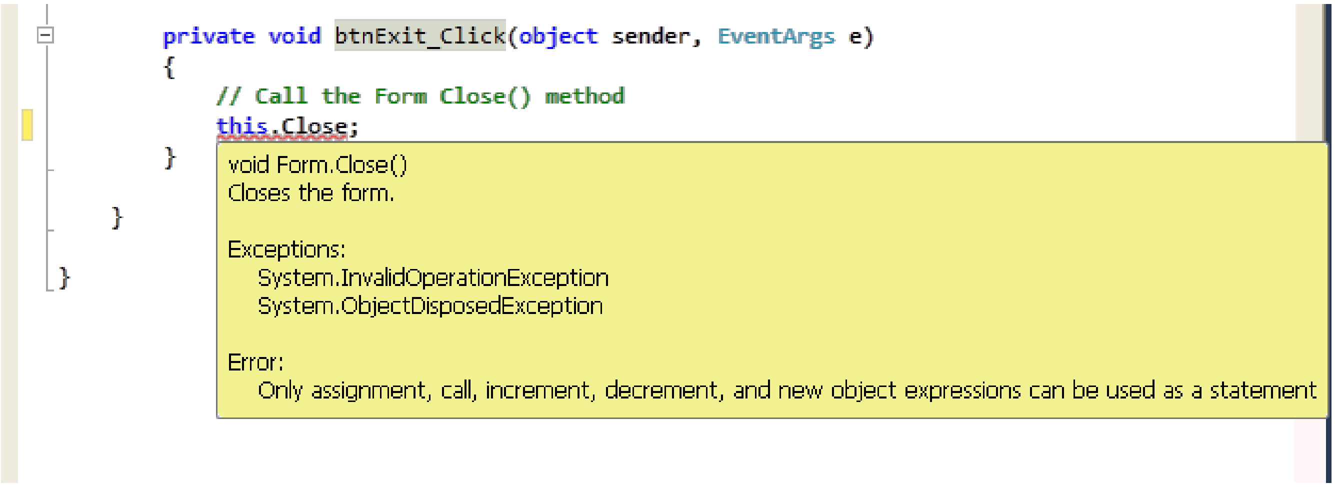
```
.
.
.
  private void exitButton_Click(object sender, EventArgs e)
  {
      // Call the Form Close() method
      this.Close;
  }
.
.
.
```

Notice the wavy red line that appears below this.Close: 

```
this.Close;
```

This red wavy underline is called an **error marker**. Whenever you code, Studio is evaluating what you type for errors. You can find out about the errors by either "hovering" with the mouse pointer over the error marker, or by selecting **View | Error List Window**. (By default, the Error List Window appears at the bottom of the Studio IDE, below the Visual Code Editor.)

Move the mouse pointer over the error marker. You will see a popup message like this:



```
        private void btnExit_Click(object sender, EventArgs e)
        {
            // Call the Form Close() method
            this.Close;
        }

    }

}
```

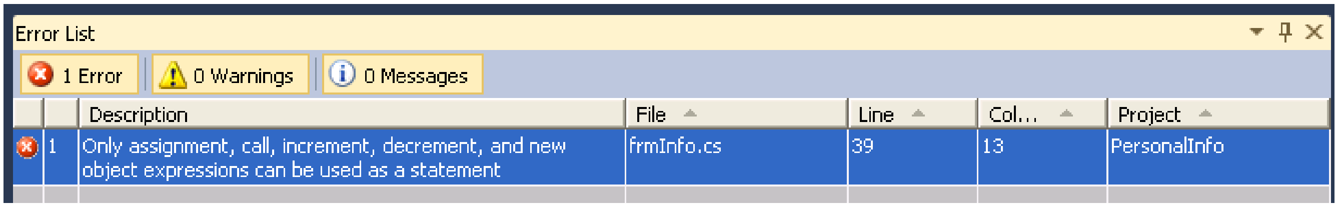
void Form.Close()
Closes the form.

Exceptions:
    System.InvalidOperationException
    System.ObjectDisposedException

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

The actual error message is located at the bottom of the error popup, indicating that **this.Close**, without the parentheses, is not a valid statement. We will discuss other error messages as they occur, and as your experience with Visual C# increases.

Selecting **View | Error List Window**. The Error List Window lists all of the errors (or warnings or other messages) related to your program. Although in this instance there is only a single error, you may have more than one error. The Error List Window will show each error on a separate row. You can double-click on a row to jump to the error in your code.



Error List

1 Error | 0 Warnings | 0 Messages

		Description	File	Line	Col...	Project
☒	1	Only assignment, call, increment, decrement, and new object expressions can be used as a statement	frmInfo.cs	39	13	PersonalInfo

Click away from the error marker, then find the Error List Window (as shown in the image above), and double-click the error row. Modify the code, replacing the missing parentheses:

CODE TO EDIT: InfoForm exitButton_Click

```
.
.
.
  private void exitButton_Click(object sender, EventArgs e)
  {
      // Call the Form Close() method
      this.Close();
  }
.
.
.
```

Next

That's it for this lesson! Now that you have completed this lesson, complete any associated projects, then quizzes.

Classes

Class Blueprint and Object Instances

Have you ever seen a blueprint? A blueprint is a document that describes in complete detail how to build something, such as a house, or a car. In modern programming, object blueprints are known as classes, and may contain some or all of the components we've discussed so far, such as properties, methods, and events.

A class, just like a blueprint, is not an actual object, but a definition for an object. In order to create an actual object, you have to create an instance of an object using the class. In this lesson, we will begin learning how to create class blueprints, and object instances.

We'll create a C# class that contains properties and methods to create a visual circle. In order to show the class working, we'll create a Windows Form with a Label and TextBox to specify the size of the circle, and a Button to create an instance of the object using the class and draw the circle. We will then create our new class, then modify the Form to work with the new class. You will notice that the Form code itself is also a class, so we will be modifying an existing class (the Form), and adding a new class (the Circle).

As we work through creating a class, we will be using variables defined using the C# reserved word *private*. These variables will hold information about our class, and will be accessed using methods called accessors. These accessor methods use the C# reserved words *get* and *set*. Don't worry too much if these terms are unfamiliar. In a later lesson, we will cover these terms, and the C# syntax, in more depth. For now, remember that just as objects in the real world have properties, so will our C# classes. Also, just as real world objects can do something, so too will our class, using public methods.

Note The pattern we will follow for creating projects and adding components is: create a new project, rename the Form filename to something descriptive, save the project, change the Form title bar, add any controls, rename any controls, change what the controls display, add code, and test. Remember to save your work early and often!

The Circle Class, Part 1

Creating the Project

First, we create the project.

Create a new Visual Studio Windows Forms Application by selecting **File | New | Project** from the menu. Make sure under the C# tree item, you have selected **Windows**. In the center section of the New Project dialog box, select **Windows Forms Application**. In the New Project dialog box, change the name of the project to **MyCircle**. and all other dropdowns and checkboxes take their default settings. Click **OK**.

You should now have a Windows Form Application showing. Click the **Save All** icon to save the Project and the Form.

Tip Whenever you start a new project, use the **Save All** icon to make sure you save the project! Simply using **Save** only saves what you are currently working on, but the **Save All** option will save all modified components of the current project.

Renaming the Default Form

Next, we need to rename our Form.

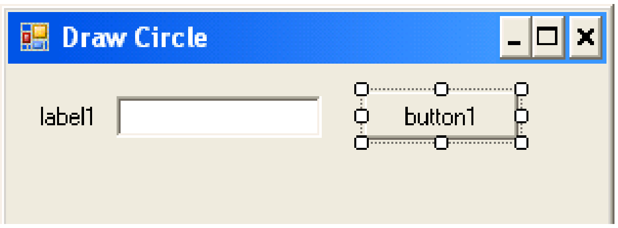
Tip When renaming your form, always be sure to include the filename extension (suffix). In the example below, the extension is .cs. If you forget the extension, you may need to remove the misnamed item from the project and add it back with the full correct name.

Locate and click on **Form1.cs** in the Solution Explorer. In the Properties Window, change the File Name to **MyCircle.cs**. Click Save or Save All to save your changes (if you change more than one item, it's a good idea to Save All).

Setting the Title Bar and Adding Controls

We now need to change the Form title bar, and add the Label, TextBox, and Button controls.

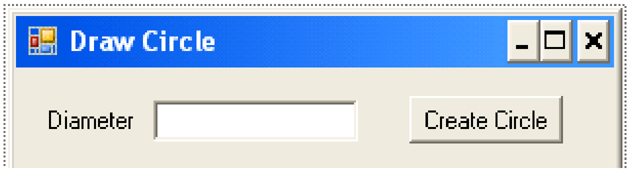
Click on the Form1 Form title bar, and find the Text property in the Properties Window. Change the Text property from Form1 to **Draw Circle**. Drag a Label, TextBox, and Button control onto the Form, aligning them along the top of the Form.



Save your changes 💾.

Next, we need to rename each control.

Click the **label1** control, and in the Properties Window, change the Name property to **circleDiameterLabel**, and the Text property to **Diameter**. Click the **TextBox** control and change the Name property to **circleDiameterTextBox**. Click the **button1** control, and change the Name to **createCircleButton** and the Text property to **Create Circle**. You may have to move the controls around so all are fully visible and evenly spaced, something like this:

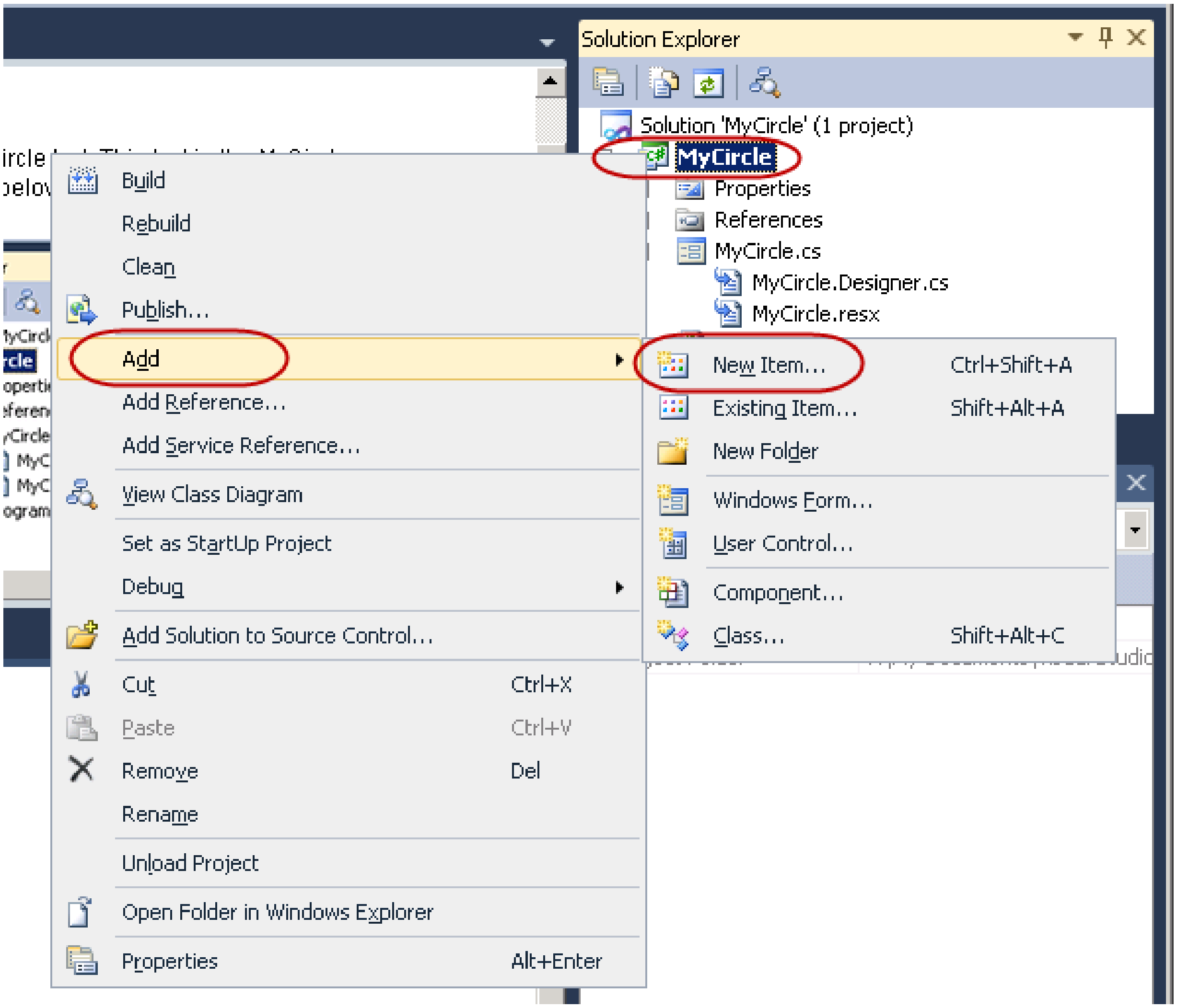


Save your changes 💾.

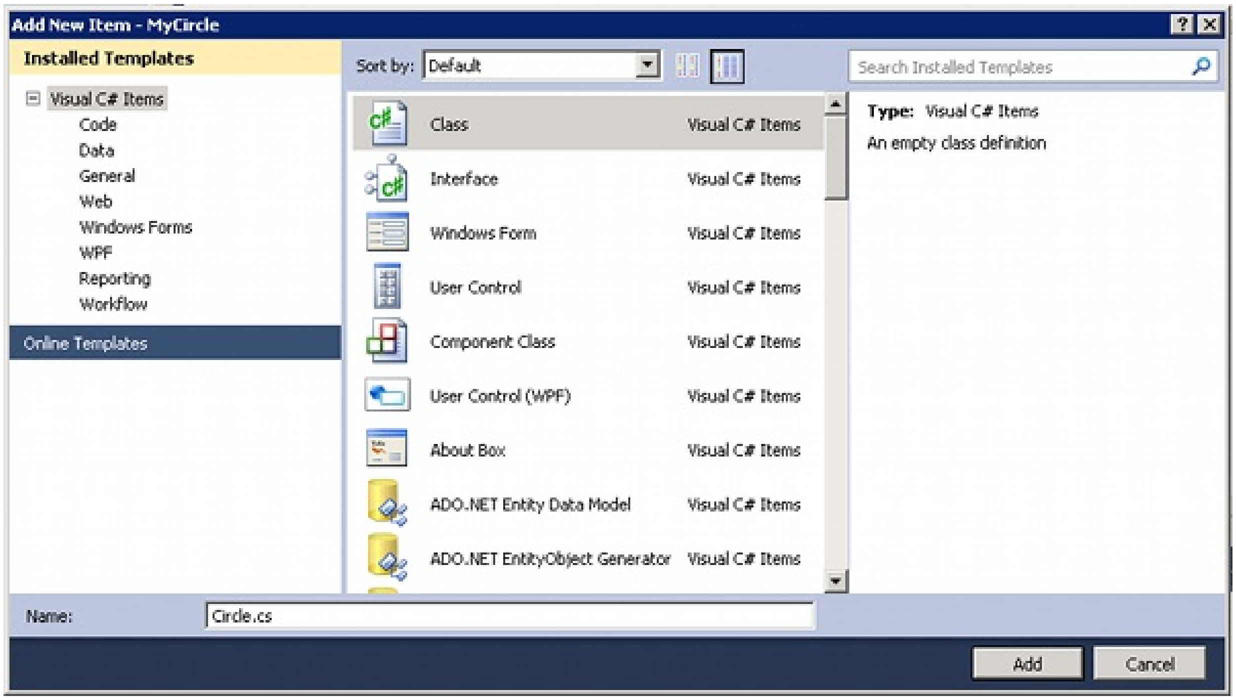
Creating a Class

Finally, we get to create our first C# class! To add a class, we use menu options to add the class from the Solution Explorer, then edit the C# code in the Code Editor.

In the Solution Explorer, just below the Solution MyCircle, right-click on the **MyCircle** text. This text is the MyCircle project entry. In the popup menu, select **Add** and then select **New Item**:



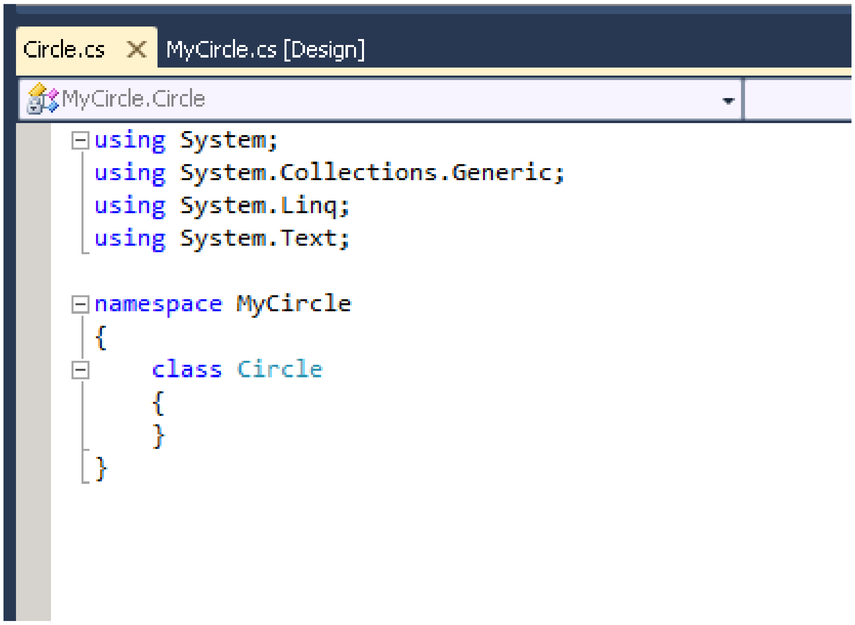
The Add New Item dialog box appears, with the Class template selected. In the Name field at the bottom of this dialog, type **Circle.cs** (replacing the default Class1.cs), and click **Add**:


Add New Item - MyCircle
Installed Templates
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Online Templates
Sort by: Default
Search Installed Templates
Class Visual C# Items
Interface Visual C# Items
Windows Form Visual C# Items
User Control Visual C# Items
Component Class Visual C# Items
User Control (WPF) Visual C# Items
About Box Visual C# Items
ADO.NET Entity Data Model Visual C# Items
ADO.NET EntityObject Generator Visual C# Items
Type: Visual C# Items
An empty class definition
Add
Cancel

Name: Circle.cs

Right-click on the **Circle.cs** tab and select **Move To Next Tab Group** to move the editor tab to the bottom of the screen.

The Circle.cs file is added to the Solution Explorer, and the Code Editor now displays the class code created by Studio:



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

namespace MyCircle
{
    class Circle
    {
    }
}
```

Let's look at the default class code provided by Studio.

OBSERVE: Circle Default Code

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

namespace MyCircle
{
    class Circle
    {
    }
}
```

Note the three unique sections: a series of **using** statements, a **namespace** statement, and a **class Circle** statement. The **using** section indicates *namespaces* of physical code groups known as *assemblies* that are included in the class. The **namespace** statement declares a logical code grouping within our code that allows your code to be clearly distinguished from other possible classes of the same name. Assemblies and namespaces will be covered more later. Finally, **class Circle** declares a class blueprint.

Tip

The curly braces {} in the code establish a container, or in programming verbiage, a block. Any opening curly brace must have a closing curly brace, and vice versa. When you have multiple matched curly braces, it can be difficult to tell which opening brace belongs to which closing brace, so the editor provides assistance: if you put the text cursor to the left of an opening curly brace, or to the right of a closing curly brace, both the brace you are near and the matching brace are highlighted to assist you. Go ahead, try it.

Adding a Class Property

Modify the Circle.cs code as shown below to add the diameter property for the Circle class, including a default diameter of 0. As mentioned before, don't cut and paste! You should always type the code from LISTINGs manually, so you become intimate with the C# language.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        // Private member property _circleDiameter, with default of 0.
        private int _circleDiameter = 0;
    }
}
```

Save your changes 💾.

Let's discuss this code.

OBSERVE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        // Private member property _circleDiameter, with default of 0.
        private int _circleDiameter = 0;
    }
}
```

Note throughout that we include **comments** at every step.

The code includes the line **using System.Drawing;** to add the necessary assemblies to support drawing. We'll access functionality of this assembly to create a Pen object that will enable us to draw a circle.

Within the **Circle** class, we will need a programming variable to hold the diameter of the circle. The code **int _circleDiameter** declares such a variable with an integer datatype. As such, the **Circle** class will only support diameters with integer, or whole, numbers. The C# reserved word **private** indicates that methods outside the **Circle** class cannot directly access this variable. Including **= 0** assigning the initial value 0 to this variable.

Note The **_circleDiameter** variable begins with an *underscore* and uses *camel case*. Although it is not required, we usually use the underscore to indicate a private class member property.

Tip Most datatypes have a default value. For integers, the default value is zero even if we don't include the **= 0** code. However, it's a good practice to declare an explicit default value this way, to make it obvious what the value is, and to make it easier in case you ever want to change the default value.

Adding Class Property Accessors

Modify the Circle.cs code as listed below to allow access to the diameter property for the Circle class.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        // Private member property _circleDiameter, with default of 0.
        private int _circleDiameter = 0;

        // ** Accessors **
        // Public accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
            // Sets the circle diameter. The value variable is the
            // value the user specifies when calling the accessor.
            set
            {
                _circleDiameter = value;
            }
            // Returns the current value of the circle diameter.
            get
            {
                return _circleDiameter;
            }
        }
    }
}
```

Save your changes 💾.

Let's discuss how this code works.

OBSERVE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        // Private member property _circleDiameter, with default of 0.
        private int _circleDiameter = 0;

        // ** Accessors **
        // Accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
            // Sets the circle diameter. The value variable is the
            // value the user specifies when calling the accessor.
            set
            {
                _circleDiameter = value;
            }
            // Returns the current value of circle diameter.
            get
            {
                return _circleDiameter;
            }
        }
    }
}
```

Unlike the private **_circleDiameter** variable, the new code for **Diameter** uses public, which provides access to the **_circleDiameter** variable for any code that uses the **Circle** class. Member property values can either be retrieved or set to a value. In computers, retrieving a property value is termed *reading*, and setting a property value is *writing*. In C#, the accessor syntax for writing a value uses the C# reserved word **set**, and the accessor syntax for reading a value uses the C# reserved word **get**. The **Diameter** is the actual name of the accessor to be used.

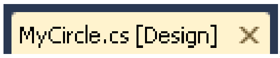
We will talk more about accessors as the lessons progress, but suffice it to say that the **set** syntax sets the value, and the **get** syntax returns the value, of the **_circleDiameter** member property. Note also that **value** variable contains the value the user specified when calling the accessor.

Note A property that may only be read is known as a *read-only* property; a property that may only be written is known as a *write-only* property. A property that may be both read and written—as is the case with the Circle member property **_circleDiameter**—is known as a *read-write* property.

So far, we have a circle class, with a read-write diameter property. Eventually, we want to use the circle class to create a circle, but before we add more code to the circle class, let's see how we would use what we have so far. To use the circle class, we need to discuss how a class, a blueprint, becomes an object, and that requires a discussion of object instances.

Object Instances and Exceptions

Remember that a class is only a blueprint—to actually use a class, we must create an *instance* of it. In order to use the class, we'll go back to the Circle Form we created, and add code to allow us to create and use an instance of the Circle class.

Click on the MyCircle.cs [Design] tab 

, and double-click on the **createCircleButton**

control. The code for the MyCircle.cs Form appears. Add the code shown below.

Note

If the createCircleButton_Click method is missing, go back to the Circle Form and double-click the **createCircleButton** object again. If you still cannot find the createCircleButton_Click method, look at your code for another method that ends in _Click, as you may have named the Button incorrectly.

CODE TO TYPE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        // Circle object.
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            // Get the entered text, convert to integer, and assign result to
            // Circle diameter property using class set accessor.
            _myCircle.Diameter = Convert.ToInt32(circleDiameterTextBox.Text);

            // Output text to Studio console so we can see what happened.
            Console.WriteLine("New diameter is: {0}", circleDiameterTextBox.Text
);
        }
    }
}
```

Save your changes 💾.

Let's discuss how this code works.

OBSERVE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        // Circle object.
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            // Get the entered text, convert to integer, and assign result to
            // Circle diameter property using class set accessor.
            _myCircle.Diameter = Convert.ToInt32(circleDiameterTextBox.Text);

            // Output text to Studio console so we can see what happened.
            Console.WriteLine("New diameter is: {0}", circleDiameterTextBox.Text
);
        }
    }
}
```

We've essentially added three lines of code related to creating and using the **Circle** class, some comments, and a line of code to output information to the Studio Output window to verify that something happened (the Output window is also called the Console window).

Note You probably noticed that the code layout for the Windows Form mimics the code layout for a class, and in fact the MyCircle Form includes the C# reserved word **class** because Forms *are* classes.

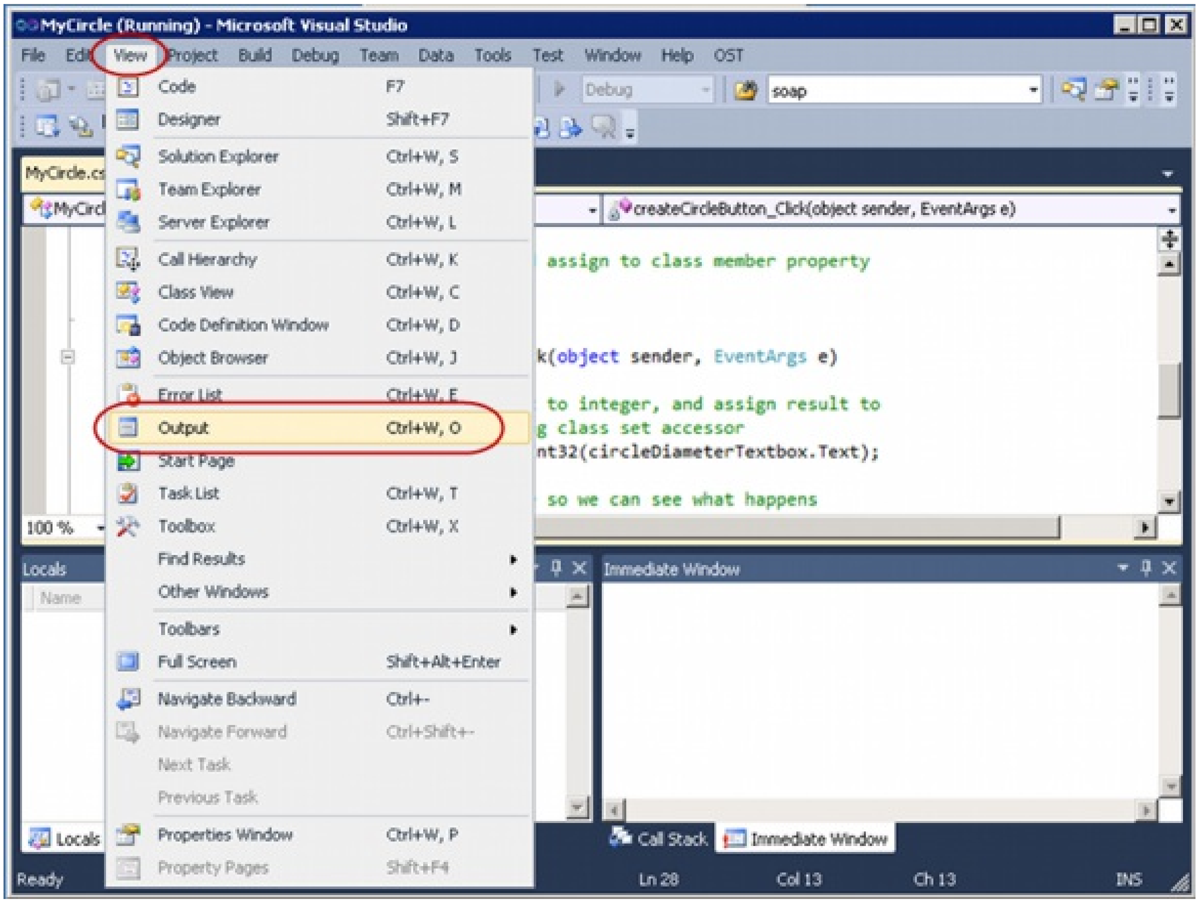
The first added line of code creates **_myCircle** as a variable of datatype **Circle**. **_myCircle** is an **instance variable** of the **Circle** class, although initially it is empty, or *null* (another C# reserved word). The second line of code uses the C# reserved word *new*, which creates an instance of a class, in this case **Circle**. This new instance of the **Circle** class is then assigned to the **MyCircle** Form member property **_myCircle**.

We still need to set the diameter of the circle, which is what the third line of code accomplishes. The user entered the diameter in the circleDiameterTextBox object, but the Textbox Text property only stores a string datatype, so we need to convert this string value into an integer value acceptable to the **Diameter** accessor of the **Circle** class. To convert the string to an integer, we use the **ToInt32** method of the **Convert** object. This object, and many other objects (and their methods and properties) is part of the System assembly, one of the assemblies loaded as part of the MyCircle.cs code.

The final code uses the **Console** method, also of the System assembly, to output information to the Studio Output window. Outputting information during execution of your code in the Studio IDE can be very helpful, and is a common method of software developers. The Console method will output standard unformatted text, or formatted text as we're using. The **{0}** represents a replaceable parameter; it will be replaced by the first variable following the comma, **circleDiameterTextBox.Text**. You can use multiple replaceable parameters by using different numbers ({0}, {1}, {2}), but for each replaceable parameter, you must also include a variable in the list that follows. If that's confusing, don't worry, we'll provide many more examples of replaceable parameters during the lessons.

So, enough explanation; let's run the code and create an instance of the Circle class!

First, one more thing: because our program is sending text to the output console, we want to make that visible before we run it. Select **View | Output** from the menu:



```
 assign to class member property

k(object sender, EventArgs e)

 to integer, and assign result to
g class set accessor
nt32(circleDiameterTextbox.Text);

 so we can see what happens
```

Click **Debug** to execute the program from the Studio IDE. When the form appears, enter a number in the textbox and click **Create Circle**. Try different numbers.

You should see the text "New diameter is:" in the Output Window in Studio; otherwise, not too much happens, but we'll change that shortly!

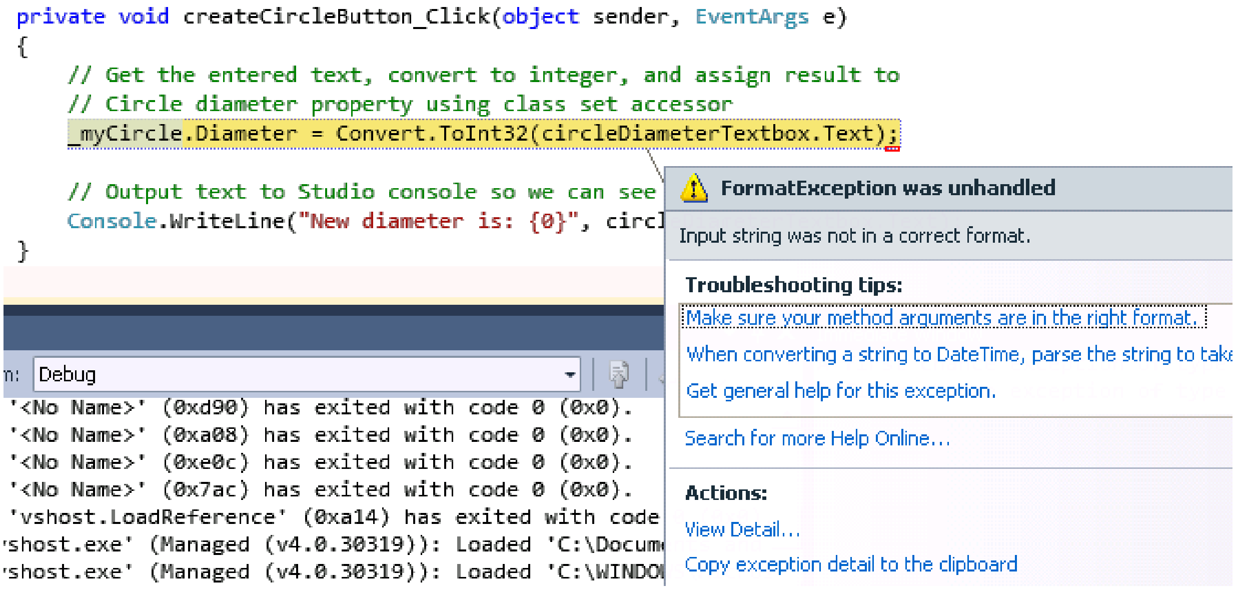
Exceptions

We know a diameter should be an interger, but users often have minds of their own! Will our code handle non-integer entries? Let's find out!

Note

When a program is running in Visual Studio, this lesson content will disappear while other data is displayed. The lesson information should reappear when the program finishes. In the next exercise, we'll cause an error condition and the program won't end gracefully, so you'll need to get the lesson content back by terminating the program manually. After creating and reviewing the error conditions below, select **Debug | Terminate All** to stop the program and return to the lesson.

Enter a non-integer value such as **ABC**; what happens? Something like this, probably:



```
private void createCircleButton_Click(object sender, EventArgs e)
{
    // Get the entered text, convert to integer, and assign result to
    // Circle diameter property using class set accessor
    _myCircle.Diameter = Convert.ToInt32(circleDiameterTextbox.Text);

    // Output text to Studio console so we can see
    Console.WriteLine("New diameter is: {0}", circl
}
```

n: Debug

```
'<No Name>' (0xd90) has exited with code 0 (0x0).
'<No Name>' (0xa08) has exited with code 0 (0x0).
'<No Name>' (0xe0c) has exited with code 0 (0x0).
'<No Name>' (0x7ac) has exited with code 0 (0x0).
'vshost.LoadReference' (0xa14) has exited with code
'shost.exe' (Managed (v4.0.30319)): Loaded 'C:\Docum
'shost.exe' (Managed (v4.0.30319)): Loaded 'C:\WINDO
```

FormatException was unhandled

Input string was not in a correct format.

Troubleshooting tips:

Make sure your method arguments are in the right format.

When converting a string to DateTime, parse the string to take

Get general help for this exception.

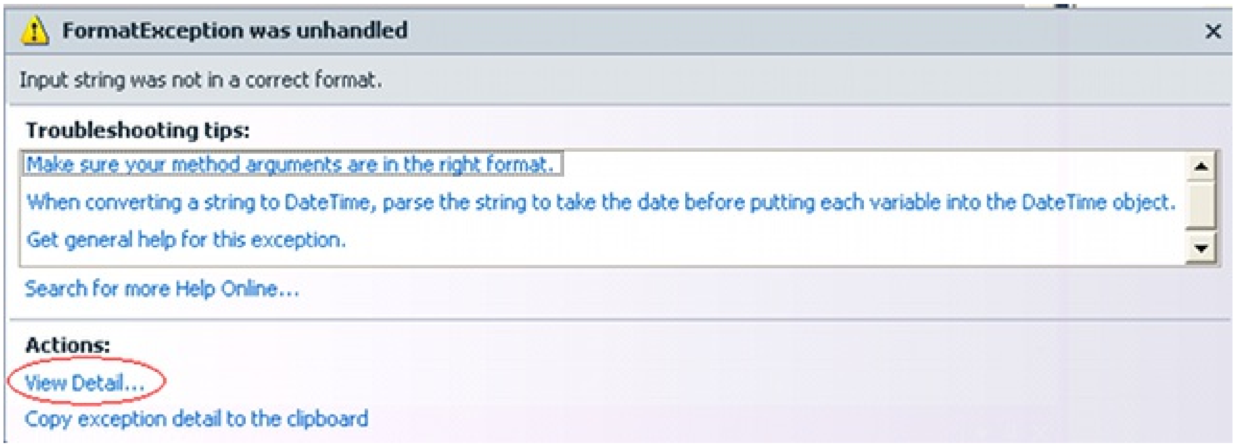
Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

If you look closely at the error message, you'll see that the error popup dialog box indicates a "FormatException was unhandled." Click the **View Detail** link in this box:



FormatException was unhandled

Input string was not in a correct format.

Troubleshooting tips:

Make sure your method arguments are in the right format.

When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.

Get general help for this exception.

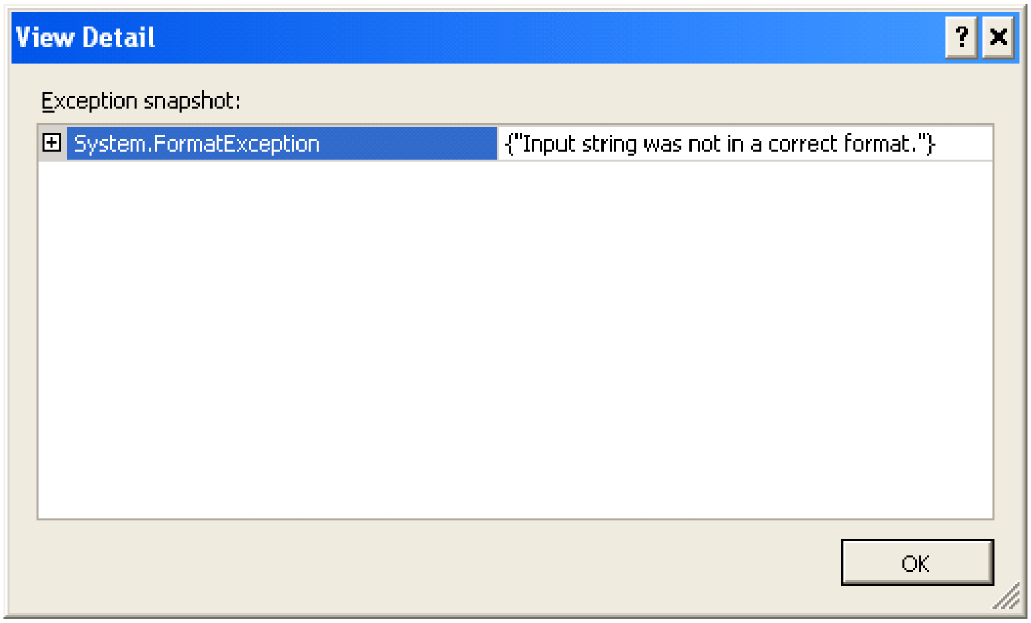
Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

The dialog box indicates "Input string was not in a correct format":



View Detail

Exception snapshot:

⊞ System.FormatException {"Input string was not in a correct format."}

You've probably guessed that C# had a problem converting the string **ABC** into a integer—exactly the problem. So let's change the code to be more robust and not raise an error when the user enters non-integer text.

Note For errors in object-oriented programs, we say they are *raised* or *thrown*.

Click **OK** to close the View Detail box, and then click the upper-right **X** to close the FormatException notification. Then (as noted earlier), select **Debug | Terminate All** to stop the program and return to the lesson.

Now let's fix the error, using the TryParse method of the System Int32 object.

Change the MyCircle.cs code as shown below, deleting the code that looks like ~~this~~ and adding the code that looks like this:

CODE TO TYPE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        // Circle object.
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            ~~// Get the entered text, convert to integer, and assign result to~~
            ~~// Circle diameter property using class set accessor.~~
            ~~_myCircle.Diameter = Convert.ToInt32(circleDiameterTextBox.Text);~~

            ~~// Output text to Studio console so we can see what happened.~~
            ~~Console.WriteLine("New diameter is: {0}", circleDiameterTextBox.Text);~~

            // Variable for TryParse result.
            int diameter = 0;

            // Try to convert the entered text to an integer, and assign result to
            // Circle diameter property using class set accessor.
            // If the TryParse method fails, diameter will be 0.
            if (Int32.TryParse(circleDiameterTextBox.Text, out diameter) == true)
            {
                _myCircle.Diameter = diameter;
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("New diameter is: {0}", diameter);
            } else {
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("Non-integer entry: {0}", circleDiameterTextBox.Text);
            }
        }
    }
}
```

Save your changes .

Let's run the revised code, and see if the exception is "handled."

Click **Debug** to execute the program from the Studio IDE. When the form appears, enter a non-integer value in the textbox and click **Create Circle**. Try different non-integer characters, and even mixed numeric and non-numeric values.

Yes, it works! The **_myCircle.Diameter** member property is only updated with numbers! Did you try numbers with decimals? Ah, those are flagged as non-integer, as numbers with a decimal are not integers!

Let's discuss how this code works (we'll save space and only show the modified createCircleButton_Click() method.

OBSERVE: MyCircle.cs

```
.
.
.
private void createCircleButton_Click(object sender, EventArgs e)
{
    // Variable for TryParse result.
    int diameter = 0;

    // Try to convert the entered text to an integer, and assign result to
    // Circle diameter property using class set accessor.
    // If the TryParse method fails, diameter will be 0.
    if (Int32.TryParse(circleDiameterTextBox.Text, out diameter) == true)
    {
        _myCircle.Diameter = diameter;
        // Output text to Studio console so we can see what happened.
        Console.WriteLine("New diameter is: {0}", diameter);
    } else {
        // Output text to Studio console so we can see what happened.
        Console.WriteLine("Non-integer entry: {0}", circleDiameterTextBox.Text);
    }
}
```

The **if** reserved word is part of the C# syntax for testing booleans. As mentioned, we use the **TryParse** method of the **Int32** object (of the System namespace), to try to convert the user-entered **circleDiameterTextBox.Text** string into an integer. The **TryParse** method takes two parameters: the first is the **string value to try to convert**, and the second is the **variable that will hold the result** of the attempted conversion. The **TryParse** method returns a *boolean* value: **true** if the conversion succeeds, **false** if it fails. If the conversion succeeds (the entered value is convertible to an integer), the second parameter, **diameter**, is replaced with the result. If the conversion fails (the entered value cannot be converted to an integer), **diameter** remains 0 as it was initialized, and the result of **TryParse** is false. So, we are in effect saying, "If the attempt to convert the user-entered data succeed, return true, and set the **diameter** variable to that converted integer; otherwise, return false, and leave the default value of the **diameter** variable as 0."

Note We will cover the **out** keyword in a future lesson.

Notice the braces {} after the **if** statement. The code in the first set of braces, or block, will execute *only* if the **if** statement returns true. The code in the second set of braces will execute *only* if the **if** statement returns false. So, within the true block following the **if** statement, we send the statement about the new diameter to the Output Window; within the false block following the **else** statement, we send a statement to the Output Window that a non-integer entry was received.

Congratulations on adding code to prevent errors in your software! Next, let's get back to finishing the **Circle** class!

The Circle Class, Part 2

Adding a Class Public Method

As we finish the Circle class, note that so far we have added two sections to the class: a Properties section (delineated by the **// ** Properties **** comments), and the Accessors section (delineated by the **// ** Accessors **** comments). Now we'll add one more section: a Public Methods section.

Add the Public Methods section to Circle.cs as shown below.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        // Private member property _circleDiameter, with default of 0.
        private int _circleDiameter = 0;

        // ** Accessors **
        // Public accessors for getting and setting member property _circleDiame
ter.
        public int Diameter
        {
            // Sets the _circleDiameter. The value variable is the
            // value the user specified when calling the accessor.
            set
            {
                _circleDiameter = value;
            }
            // Returns the current value of _circleDiameter.
            get
            {
                return _circleDiameter;
            }
        }

        // ** Public methods **
        // Public method (behavior) to draw a circle.
        public void DrawCircle(Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid.
            if (drawingArea != null)
            {
                // Call the DrawEllipse method of the drawing area,
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the m_intDiameter
                // member property for both the height and width.
                drawingArea.DrawEllipse(new Pen(Color.Red), x, y, _circleDiamete
r, _circleDiameter);
            }
        }
    }
}
```

Save your changes 💾.

Let's discuss how this code works.

OBSERVE: Circle.cs

```
.
.
.
        // ** Public methods **
        // Public method (behavior) to draw a circle.
        public void DrawCircle(Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid.
            if (drawingArea != null) {
                // Call the DrawEllipse method of the drawing area,
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the _circleDiameter
                // member property for both the height and width.
                drawingArea.DrawEllipse(new Pen(Color.Red), x, y, _circleDiameter, _circleDiameter);
            }
        }
```

This new code adds a public method named **DrawCircle** that may be called to actually draw a circle! The circle drawn will have a diameter according to the **_circleDiameter** class member property. The circle will be drawn starting at the **x** and **y** position specified. And what color? Finally, we get to see why we needed that **using** reference to System.Drawing so we could create a Pen object to do the actual drawing! The color, for now, is set to **Red**.

Calling a class public method

Now, we just need to modify MyCircle.cs one last time to call this **DrawCircle** method whenever a user enters a valid diameter and clicks the **Create Circle** Button.

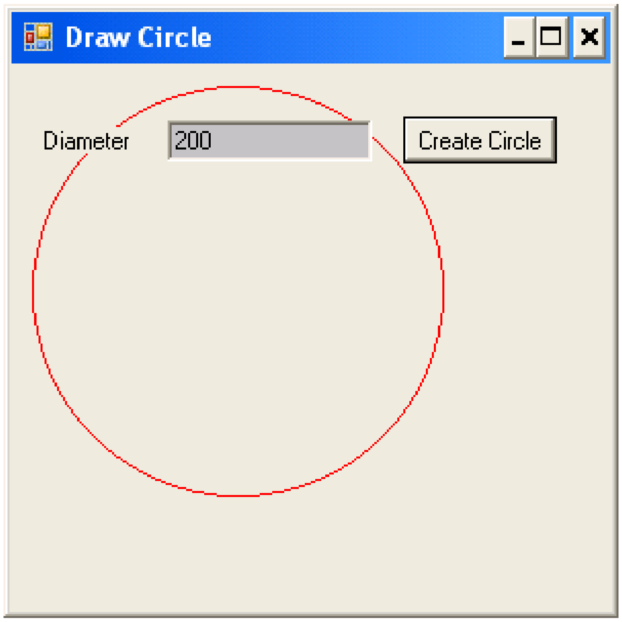
Add the call to the **DrawCircle** method in MyCircle.cs, as shown below:

CODE TO TYPE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        // Circle object.
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            // Variable for TryParse result.
            int diameter = 0;

            // Try to convert the entered text to an integer, and assign result
to
            // Circle diameter property using class set accessor.
            // If the TryParse method fails, diameter will be 0.
            if (Int32.TryParse(circleDiameterTextBox.Text, out diameter) == true
)
            {
                _myCircle.Diameter = diameter;
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("New diameter is: {0}", diameter);
                // Get a graphics canvas to draw on, and call the DrawCircle pub
lic method
                // of the Circle class to draw the actual red circle.
                _myCircle.DrawCircle(CreateGraphics(), 10, 10);
            } else {
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("Non-numeric entry: {0}", circleDiameterTextBo
x.Text);
            }
        }
    }
}
```

Save your changes 💾.

Click **Debug** ▶ to execute the program. When the form appears, enter an integer in the textbox (such as 200), and click **Create Circle** button. A red circle should be drawn!



Diameter 200

Let's discuss how this code works.

OBSERVE: MyCircle.cs

```
.
.
.
          private void createCircleButton_Click(object sender, EventArgs e)
          {
              // Variable for TryParse result.
              int diameter = 0;

              // Try to convert the entered text to an integer, and assign resul
t to
              // Circle diameter property using class set accessor.
              // If the TryParse method fails, diameter will be 0.
              if (Int32.TryParse(circleDiameterTextBox.Text, out diameter) == tr
ue)
              {
                  _myCircle.Diameter = diameter;
                  // Output text to Studio console so we can see what happened.
                  Console.WriteLine("New diameter is: {0}", diameter);
                  // Get a graphics canvas to draw on, and call the DrawCircle p
ublic method
                  // of the Circle class to draw the actual red circle.
                  _myCircle.DrawCircle(CreateGraphics(), 10, 10);
              } else {
                  // Output text to Studio console so we can see what happened.
                  Console.WriteLine("Non-numeric entry: {0}", circleDiameterText
Box.Text);
              }
          }
```

Whenever the user clicks the **Create Circle** button, if the entered diameter is a valid integer, we call the **DrawCircle** method of the **_myCircle** instance variable to draw the red circle. One of the parameters to the **DrawCircle** method is a Graphics object, which represents the "canvas" we will draw the circle on. We get this "canvas" by calling the **CreateGraphics()** method of the System.Windows.Forms namespace.

Naming Conventions

Congratulations on creating your own C# code for drawing a circle!

As you've worked through this lesson, and previous lessons, you've seen us mixing Pascal case and camel case when naming different programming elements, and even adding an underscore prefix. At this point, it might be helpful to present a summary of the rules we've been following. These rules are really just guidelines for adopting a consistent naming standard that is a commonly accepted convention with C# developers. Over time, you may develop your own rules, and most development companies have standards that software developers are required to follow.

- Forms: Pascal case, descriptive, commonly describing the purpose of the Form (MyCircle)
- Controls: Camel case, descriptive, ending with a declaration of the type of control (circleDiameterTextBox, createCircleButton)
- Classes: Pascal case, descriptive (Circle)
- Class properties (variables): Camel case, prefixed with underscore, descriptive (_circleDiameter)
- Class methods: Pascal case, descriptive
- Method variables: Camel case, descriptive (diameter)
- Filenames: Match the contents (MyCircle.cs, Circle.cs)

During our coding we have had one other C# element that also has a naming convention that we've ignored, and will continue to ignore, for a couple more lessons: namespace.

That's it for this lesson! Now that you have completed this lesson, complete any associated projects, then quizzes.

Variables, Types, and Operators

Introduction

Welcome back! Previously, we introduced the concept of a *variable* to hold information within your program. For example, we used **diameter** to store an integer, the diameter of a circle. We will continue to use variables to store and manipulate information, so we need to learn more about variables—the type of information a variable may store, and how to manipulate the data stored in the variables. Also remember that variables are used to store *properties* of an object.

Variables and Data Types

What is a Variable?

From previous lessons, you probably recognize that a variable is a programming construct we use to store information in our software. Variables may store different types of information, including numbers, strings (typically, a sequence of characters such as letters, numbers, symbols), boolean values (true or false), and even objects. We name our variables so we can understand the purpose of the variable, such as: diameterOfCircle (number), isFinished (boolean), firstName (string), myCircle (class), or employeePhoto (image). The type of information that may be stored in a variable depends on its datatype.

Naming a Variable

Every programming language includes restrictions about what may be used for a variable name, and C# is no exception. C# variables:

- **MUST** use only letters, digits, or the underscore character.
- **MUST** start with a letter or underscore.
- **ARE** case-sensitive, so radiusOfCircle is NOT the same as RadiusOfCircle.
- **MUST NOT** use a C# keyword, such as class, integer, etc. (A keyword can be PART of a variable name, such as classSize.

After following the requirements for a variable, you still have a wide choice as to what variable names to use. Here are a few guidelines you should consider:

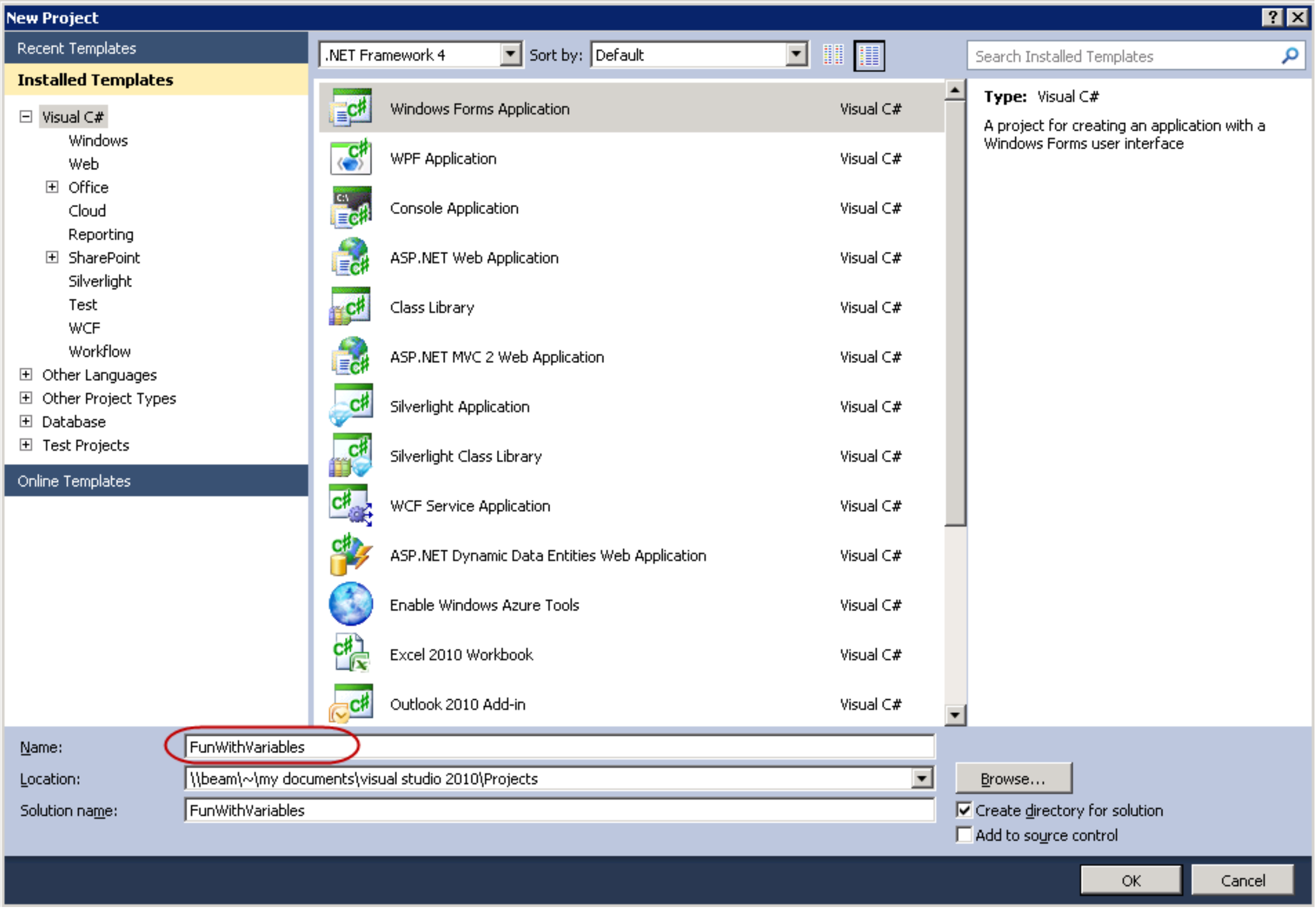
- **DO** follow a consistent naming pattern.
- **DO** use camel case, such as diameterOfCircle, rather than Diameterofcircle.
- **DO** use meaningful, easily readable names, such as horizontalAlignment rather than alignH.
- **DO** favor readability over brevity, such as canScrollHorizontally, rather than scrollableX.
- **DO NOT** use hyphens or other non-alphanumeric characters (also avoid underscores except at the start of a name).
- **DO NOT** use "Hungarian notation" (the practice of including a prefix to describe the variable data type).
- **DO NOT** use abbreviations or acronyms.

For more information, see .NET Framework 4 General Naming Conventions and Visual Studio 2010 - C# Keywords.

Fun With Variables

Enough talk about variables, let's use them! Let's begin by creating a new project.

Create a new Visual Studio Windows Forms Application by selecting **File | New | Project**. In the New Project dialog box, change the Name of the project to **FunWithVariables**, and make sure that Windows is selected under the C# tree item and Windows Forms Application is selected in the center section. Everything else should take their default settings. When you finish, click **OK**:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
ASP.NET MVC 2 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
WCF Service Application Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
Enable Windows Azure Tools Visual C#
Excel 2010 Workbook Visual C#
Outlook 2010 Add-in Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: FunWithVariables

Location: \\beam\~\my documents\visual studio 2010\Projects

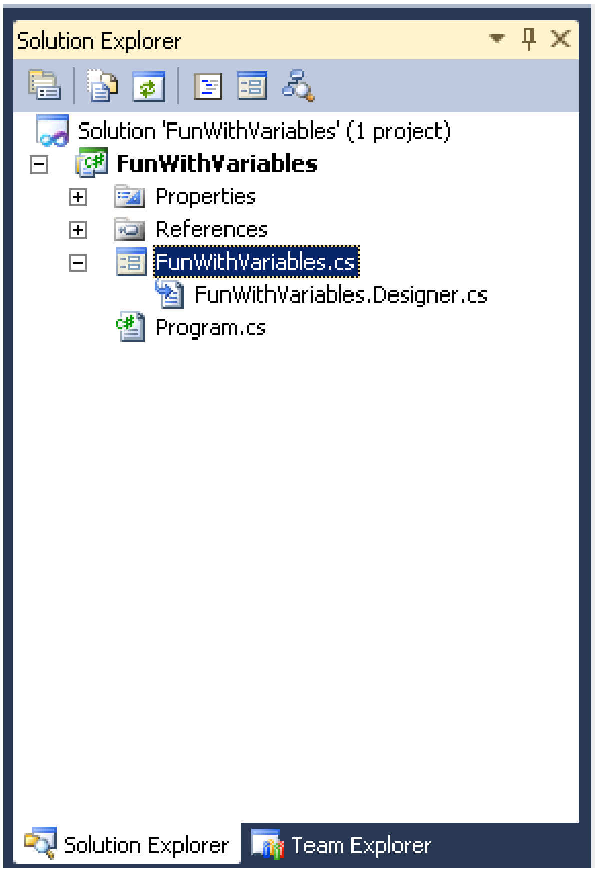
Solution name: FunWithVariables

☑ Create directory for solution

☐ Add to source control

When your new Windows Form Application appears, right-click the **Form1.cs [Design]** tab and select **New Horizontal Tab Group**. Now let's rename the form and save the project.

In the Solution Explorer, locate and select the entry for **Form1.cs**. In the Properties window, change the File Name to **FunWithVariables.cs**.

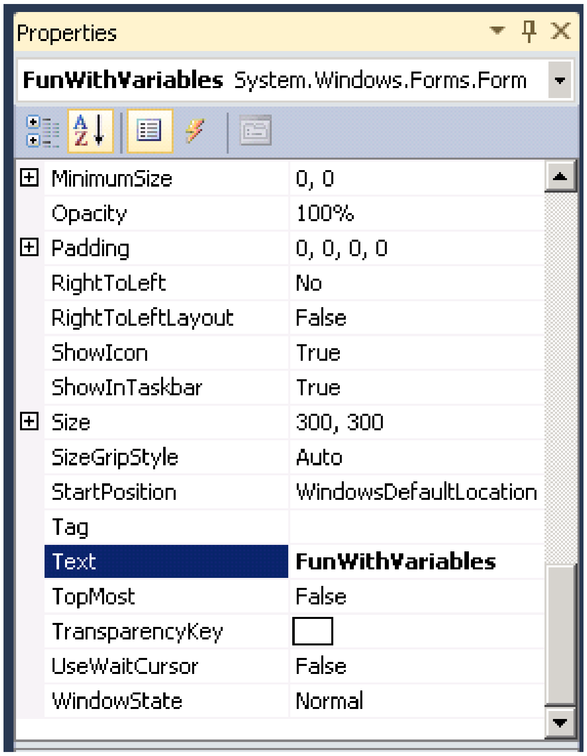


Save the project .

Tip To change the name of Form1.cs, you can also double-click the form name in the Solution Explorer, which will take you to the File Name Property. You can also click once, pause, then click again, and directly change the name of the form in the Solution Explorer. *Always remember to include the .cs extension!*

Next, let's change the Form title, and enter the code section to start working with variables.

Click on the Form in the Windows Forms Designer, and in the Properties window, change the Text property to **Fun With Variables**:

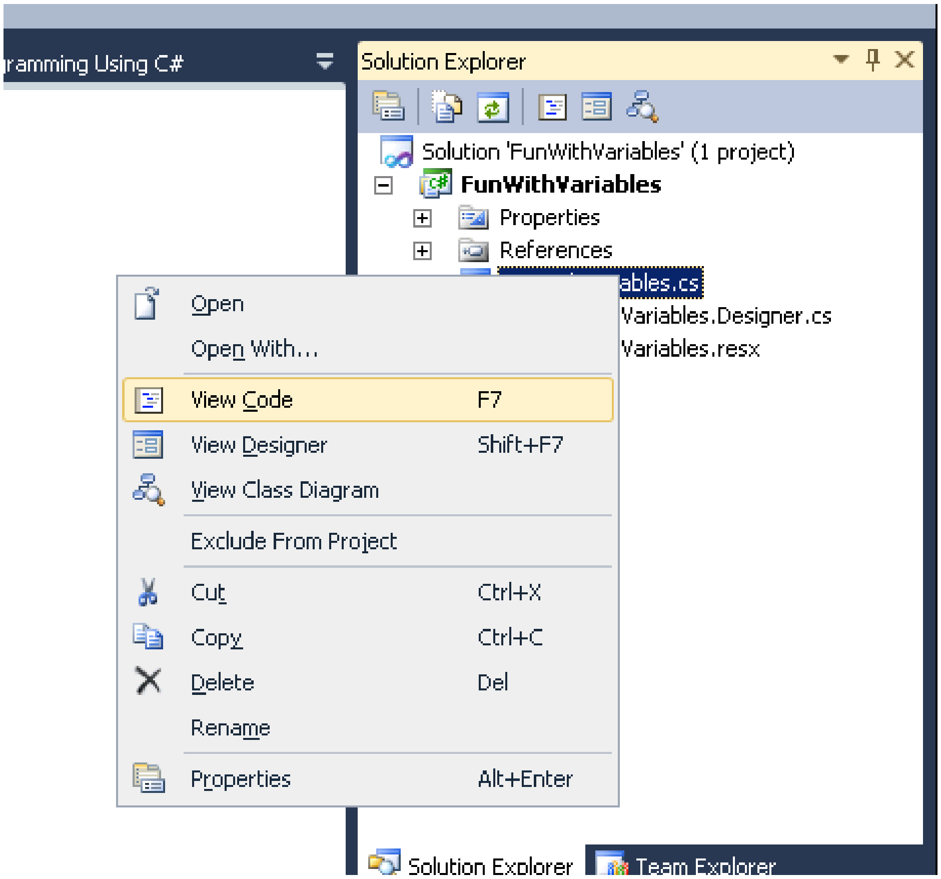


Properties

FunWithVariables System.Windows.Forms.Form

⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**FunWithVariables**
TopMost	False
TransparencyKey	□
UseWaitCursor	False
WindowState	Normal

Right-click the **FunWithVariables.cs** form name in the Solution Explorer, and select View Code to enter the C# Code Editor for the form:



Let's enter a few different types of variables: an integer, a string, a number with a decimal, and an image.

Modify the FunWithVariables.cs source code as shown below. Remember to type the new lines in by hand, and to click **Save All** when you finish!

CODE TO TYPE: FunWithVariables.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 FunWithVariables
{
    public partial class FunWithVariables : Form
    {
        public FunWithVariables()
        {
            string myFirstName = "Jimi";            // Use your own name
            int myAge = 21;                         // Use your own age
            double thePriceOfGasPerGallon = 3.10;   // We can dream, can't we?

            InitializeComponent();
        }
    }
}
```

Let's discuss the code we entered.

OBSERVE: FunWithVariables.cs

```
.
.
.
namespace FunWithVariables
{
    public partial class FunWithVariables : Form
    {
        public FunWithVariables()
        {
            string myFirstName = "Jimi";
            int myAge = 21;
            double thePriceOfGasPerGallon = 3.10;

            InitializeComponent();
        }
    }
}
```

Here, I've used **orange** for data types, **dark blue** for variable names, and **dark green** for the assignment of values to the variables using the equate operator (more on operators shortly).

With this code, we're using one variable of type string, one variable of type int (short for integer), and one variable of type double. We discussed the **string** type earlier: a sequence of characters such as letters, numbers, and symbols. The string variable name is **myFirstName**. We assigned a value of "Jimi" to **myFirstName** using the equate operator (the equals sign) in **= "Jimi"**.

The **int** and **double** data types are both numbers: an **int** stores only whole numbers, whereas a **double** may store numbers with decimals. Why not just use a **double** for both variables? C# is known as a *strongly-typed* language, which means that the operations that may be performed on a variable are determined by the type of the variable, and these operations follow strict rules to ensure the integrity of the data within the variable. Data types are stored in memory, and different data types require different amounts of memory. A particular data type has a range of values it can contain, depending on how much memory the variable uses. For example, an **int** may store a number from -2,147,483,648 to 2,147,483,647, but a **double** may store a number from 5.0 x 10^{-324} to 1.7 x 10^{308}, with up to 16 decimal places. A **double** takes twice as much memory as an **int**. Need we say that that is a very significant difference? OK, we'll say it: that is a very significant difference!

We will introduce more of the data types as we need them, but for now, you can see the use of an **int** for the **myAge** variable, assigned a value of **21**, and a **double** for the **thePriceOfGasPerGallon** variable, assigned a value of **3.10**.

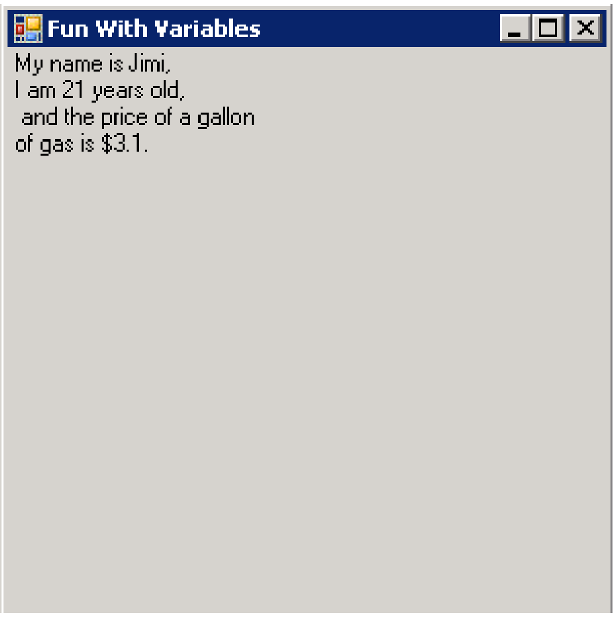
"So," I hear you asking, "where is this 'fun' you spoke of, with these variables?" First, let's introduce some more code.

Modify the FunWithVariables.cs source code as shown below.

CODE TO TYPE: FunWithVariables.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 FunWithVariables
{
    public partial class FunWithVariables : Form
    {
        public FunWithVariables()
        {
            string myFirstName = "Jimi";
            int myAge = 21;
            double thePriceOfGasPerGallon = 3.10;

            InitializeComponent();

            string infoAboutMe = "My name is " + myFirstName + ",\n" +
                   "I am " + myAge.ToString() + " years old, " + "\n" +
                   " and the price of a gallon\n" +
                   "of gas is $" + thePriceOfGasPerGallon.ToString() + ".\n";

            Label myInfoTextLabel = new Label();
            myInfoTextLabel.AutoSize = true;
            myInfoTextLabel.Text = infoAboutMe;
            this.Controls.Add(myInfoTextLabel);
        }
    }
}
```

Click **Save All** ; then, click ▶ to execute the program from the Studio IDE. You should see something like this:



```
My name is Jimi,
I am 21 years old,
 and the price of a gallon
of gas is $3.1.
```

Now let's discuss the code.

OBSERVE: FunWithVariables.cs

```
.
.
.
            InitializeComponent();

            string infoAboutMe = "My name is " + myFirstName + ",\n" +
                  "I am " + myAge.ToString() + " years old, " + "\n" +
                   " and the price of a gallon\n" +
                   "of gas is " + thePriceOfGasPerGallon.ToString() + ".\n";

            Label myInfoTextLabel = new Label();
            myInfoTextLabel.AutoSize = true;
            myInfoTextLabel.Text = infoAboutMe;
            this.Controls.Add(myInfoTextLabel);
.
.
.
```

Again, we've highlighted the variables in **dark blue**, and the data types in **orange**. We added a new string named **infoAboutMe** that uses the other variables to create a **string** that puts together all of the variable contents for display. We've previously seen the equate operator used for assignment, but with **infoAboutMe** we see the concatenation operator (the plus sign) used to concatenate strings together. The **myFirstName** variable is already a string, but for the numeric-type variables **myAge** and **thePriceOFGasPerGallon**, we added calls to a **ToString()** method that returns the numerical information as a string.

Tip When concatenating, C# will try to find the most obvious conversion for a numeric variable, so you can omit the **ToString()** method call for most numbers. For now, we used **ToString()** to show we know that those data types need to be converted.

Previously, we added controls directly to the Form, but in this code, we create a **Label** control dynamically. By *dynamically*, we mean that we created the **Label** after the program was running, rather than as a **static** control—one that was added at design time. To add the **Label** control, we used the C# keyword **new**. We will see that simple data types are easily created using the syntax **dataType variableName** = **defaultValue**, but more complex data types, or objects, require the **new** keyword.

In addition to creating the **Label myInfoTextLabel**, we set the AutoSize property to **true** to make the **Label** stretch. We also set the Text property to the **infoAboutMe** variable. Finally, we used the Add method of the Controls object to actually add the **myInfoTextLabel Label** control to the **FunWithVariables** Form represented by the **this** variable.

So, what other new "stuff" have we introduced? What about that funny "\n"? A string is usually created from a *string literal*, like the **"Jimi"** text, but not all string literals are printable. The "\n" represents a newline character, that wraps the text to a new line. This special syntax uses a backslash character to *escape* the non-printable character, differentiating it from a literal lower-case "n". Other common escape characters you might use are \t (horizontal tab), \' (single quote) and \" (double quote). The quotation marks are useful when you need to use a single or double quote mark within a string literal (itself enclosed in quotation marks).

Operators

To wrap up this lesson, we'll talk a little more about operators. We saw the equate operator (=) and the concatenation operator (+). As mentioned before, the available operations are determined by the type of data. For example, the concatenation operator for a string is also the addition operator for numbers. C# in general uses the standard operators we've seen in mathematics, such as the negative sign (-) for subtraction or changing the sign of a number, the asterisk (*) for multiplication, the forward slash (/) for division, and parentheses for grouping. And, just like with mathematics, operators have a precedence, or order, in which they are processed, so when in doubt, use parenthesis to force the order you want!

So, let's add to our FunWithVariables project!

More Fun With Variables

Modify the FunWithVariables.cs source code using the Code Editor as shown below.

CODE TO TYPE: FunWithVariables.cs

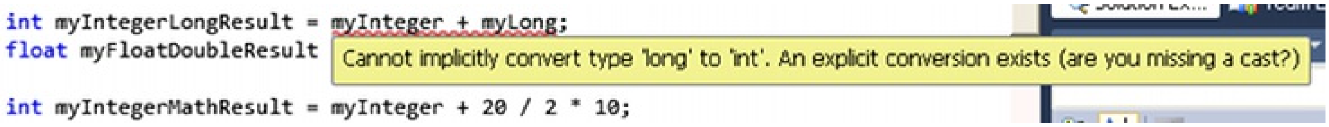
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace FunWithVariables
{
    public partial class FunWithVariables : Form
    {
        public FunWithVariables()
        {
            string myFirstName = "Jimi";
            int myAge = 21;
            double thePriceOfGasPerGallon = 3.10;

            InitializeComponent();

            string infoAboutMe = "My name is " + myFirstName + ",\n" +
                   "I am " + myAge.ToString() + " years old, " + "\n" +
                   " and the price of a gallon\n" +
                   "of gas is " + thePriceOfGasPerGallon.ToString() + "\n";

            Label myInfoTextLabel = new Label();
            myInfoTextLabel.AutoSize = true;
            myInfoTextLabel.Text = infoAboutMe;
            this.Controls.Add(myInfoTextLabel);

            int myInteger = 32;
            long myLong = 32;

            float myFloat = 500F;
            double myDouble = 500;

            int myIntegerLongResult = myInteger + myLong;
            float myFloatDoubleResult = myFloat + myDouble;

            int myIntegerMathResult = myInteger + 20 / 2 * 10;
            float myFloatMathResult = myFloat - 10.5 / (2 * 10);
        }
    }
}
```

Click **Save All**. Yes, there are errors in this code! You should see three red squiggly lines. Let's discuss the first one:

OBSERVE: Error in FunWithVariables.cs

```
int myIntegerLongResult = myInteger + myLong;
```

Move your mouse pointer over the first squiggly red line next to the plus symbol. You should see a popup like this:



```
int myIntegerLongResult = myInteger + myLong;
float myFloatDoubleResult
int myIntegerMathResult = myInteger + 20 / 2 * 10;
```

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

A long is a numeric type like integer that only stores whole numbers (negative and positive), but takes up twice as much memory as an integer, with a range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. When doing math, if you attempt to combine numbers of different data types, C# will try to convert the data to the target datatype. In our code, the **myIntegerLongResult** variable is an **int**. This is known as a *narrowing* conversion, where the resulting data type may not be capable of storing all of the information from the original variables. Comparing the range of information in an integer versus a long, the potential for data loss makes sense. Studio and C# require that you either provide an *explicit* conversion (also known as a *cast*), or make some other change. The opposite of *narrowing* is *widening*, and C# is typically happy to provide an *implicit* in that case, when no data loss would result.

OBSERVE: Errors in FunWithVariables.cs

```
  float myFloatDoubleResult = myFloat + myDouble;
  float myFloatMathResult = myFloat - 10.5 / (2 * 10);
```

We see similar errors for the other two. You might recall that a double has a range from 5.0 x 10^{-324} to 1.7 x 10^{308}. A float, however, only has the range of 1.5 x 10^{-45} to 3.4 x 10^{38}, so just as with converting a long to an integer, converting a double to a float is a narrowing conversion. In fact, when assigning a number with a decimal, C# uses a double data type by default. You will notice that the variable **myFloat** assignment had to use a special designation to explicitly cast the 500 assigned value to a float using **500F** with the F (or f) suffix. Let's verify that.

Modify the FunWithVariables.cs source code, temporarily removing the F suffix following the 500 for the assignment of **myFloat**. You will see another red squiggly error line. Add the F suffix back, and the red squiggly error line will also disappear.

So, how can we fix these conversion problems?

Modify the FunWithVariables.cs source code as shown below.

Note The source code can get long, and rarely do we need to adjust the using block, so we will omit unchanged code in some of our examples. We will always show enough lines of code for you to identify where in the code you need to work.

CODE TO TYPE: FunWithVariables.cs

```
.
.
.
int myInteger = 32;
long myLong = 32;

float myFloat = 500F;
double myDouble = 500;

~~int~~long myIntegerLongResult = myInteger + myLong;
~~float~~double myFloatDoubleResult = myFloat + myDouble;

int myIntegerMathResult = myInteger + 20 / 2 * 10;
float myFloatMathResult = myFloat - 10.5 / (2 * 10);
```

Let's discuss the modified code using the abbreviated OBSERVE block below.

OBSERVE: FunWithVariables.cs

```
long myIntegerLongResult = myInteger + myLong;
double myFloatDoubleResult = myFloat + myDouble;
```

The only changes we made were to change the data type of **myIntegerLongResult** from **int** to **long**, and of **myFloatDoubleResult** from **float** to **double**, effectively changing these conversion scenarios from *narrowing* to *widening*. Previously, C# would have had to implicitly convert **myLong** to an **int**, but now C# implicitly converts **myInteger** to **long**, with no possible loss of data. The same holds true for **myFloatDoubleResult**: **myFloat** is implicitly converted to a **double**, with no possible data loss.

There is still one more error: myFloat - **10.5** / (2 * 10). Do you remember that by default a decimal number is a double? So, 10.5 is really a double, but the variable we're assigning to is **myFloatMathResult**, a **float**. We need to modify 10.5 to be a **float**.

Modify the FunWithVariables.cs source code as shown below:

CODE TO TYPE: FunWithVariables.cs

```
.
.
.
int myIntegerMathResult = myInteger + 20 / 2 * 10;
float myFloatMathResult = myFloat - 10.5F / (2 * 10);
```

Adding the F suffix to 10.5 solved the problem, allowing us to use a floating 10.5 literal, rather than a double 10.5 literal.

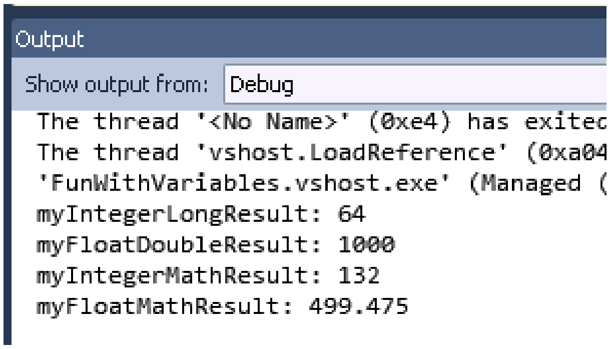
Now that we've finished discussing conversions (for now), what about the operators? We used a variety of mathematic operators, including +, -, *, /, and ( ). Let's add some output to the Studio Output Window using the Console object so we can see the actual values; then, we're finished with this lesson!

Modify the FunWithVariables.cs source code as shown below:

CODE TO TYPE: FunWithVariables.cs

```
.
.
.
int myInteger = 32;
long myLong = 32;

float myFloat = 500F;
double myDouble = 500;

long myIntegerLongResult = myInteger + myLong;
double myFloatDoubleResult = myFloat + myDouble;

int myIntegerMathResult = myInteger + 20 / 2 * 10;
float myFloatMathResult = myFloat - 10.5F / (2 * 10);

Console.WriteLine("myIntegerLongResult: " + myIntegerLongResult);
Console.WriteLine("myFloatDoubleResult: " + myFloatDoubleResult);

Console.WriteLine("myIntegerMathResult: " + myIntegerMathResult);
Console.WriteLine("myFloatMathResult: " + myFloatMathResult);
```

Click **Save All**, and then click **Debug**. Select **View | Output** to see the Studio Output window for the results of the math:



```
The thread '<No Name>' (0xe4) has exitec
The thread 'vshost.LoadReference' (0xa04
'FunWithVariables.vshost.exe' (Managed (
myIntegerLongResult: 64
myFloatDoubleResult: 1000
myIntegerMathResult: 132
myFloatMathResult: 499.475
```

When you finish, close the Output window, and minimize this lesson window to find and close the Fun With Variables form.

That's it for this lesson! Now that you have finished this lesson, complete any associated projects, then quizzes.

Making a Decision, Comparison Operators, Booleans

Introduction

Programming often is about **making a decision**: choosing if one number is greater than another number, if certain text is found in a string, or if what a user typed is valid or not. C#, as with most languages, contains a number of **decision-making structures**. For this lesson, we will cover two such structures: **if** and **if..else**. We'll also discuss **comparison operators**, such as greater than (does one DVD cost more than another DVD?), as well as **Boolean operators** for compound questions.

Note Comparison operators are also known as *relational* operators. Boolean operators are also known as *logical* operators.

Making a Decision

The Ubiquitous If

Every day, you make decisions. Decisions are often expressed as questions: if I leave now, will I make it to the post office before it closes? Programming involves taking such questions, and using code to represent these same questions to make decisions. To begin our discussion of the decision-making structures in C#, we will use a fictitious person, Sally, by defining her name, gender (Female), and age (22), within a C# program. We will determine the answers by asking questions like:

- Is she older than 21?
- Is she younger than 21?
- Is she exactly 21?
- Is she male?
- Is she female?

As you compose questions, sometimes the answer will make you do one thing, or another, or maybe even an entire series of actions. We need to be able to code these same types of actions within C#.

Comparison Operators

When making decisions or asking questions using **if**, you will need C# operators to help with the comparison. Those comparison operators are expressions such as greater than, less than or equal to, not equal to, and even equal to. These operators should look familiar from math.

>	Greater than	The value on the left is greater than the value on the right.
>=	Greater than or equal to	The value on the left is greater than or equal to the value on the right.
<	Less than	The value on the left is less than the value on the right.
<=	Less than or equal to	The value on the left is less than or equal to the value on the right.
==	Equal to	The value on the left is equal to the value on the right.
!=	Not equal to	The value on the left is not equal to the value on the right.

Coding the if

To ask these questions, we will use the C# *if* syntax, the *comparison operators*, and use C# variables to represent information about Sally.

If Statement Flow

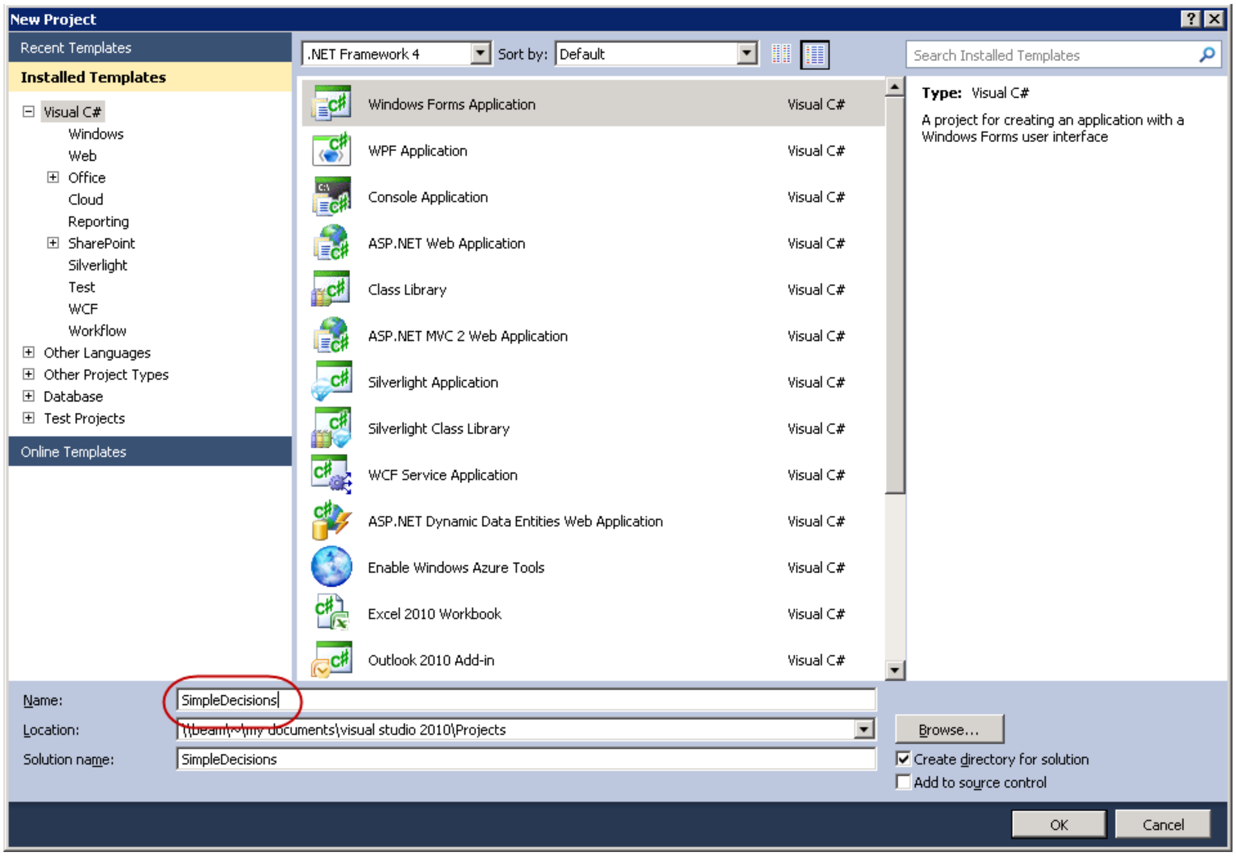
```
if (expression) {
    execute this block if the expression evaluates true
}
if the expression evaluates to false, continue after the braces
```

In the *if* syntax, if the result of the **expression** is true, the **statements within the braces**, or block, immediately following the **if** statement are executed. If the result is false, execution moves to the **code immediately following the block**.

Tip When composing an if statement, remember that the block of code following the expression is only executed if the expression is true, *but* the expression itself can be negative, such as "if gender is not equal to Male".

First, we'll create a new project named **SimpleDecisions**.

Select **File | New | Project**. In the New Project dialog box, change the Name to **SimpleDecisions**, and make sure **Windows** is selected under the Visual C# tree item, and **Windows Forms Application** is selected in the center section. All other options should have the default settings as shown below. Click **OK** when ready.


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
ASP.NET MVC 2 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
WCF Service Application Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
Enable Windows Azure Tools Visual C#
Excel 2010 Workbook Visual C#
Outlook 2010 Add-in Visual C#
Browse...
OK
Cancel

.NET Framework 4

Sort by: Default

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: SimpleDecisions

Location: \\beam\~\my documents\visual studio 2010\Projects

Solution name: SimpleDecisions

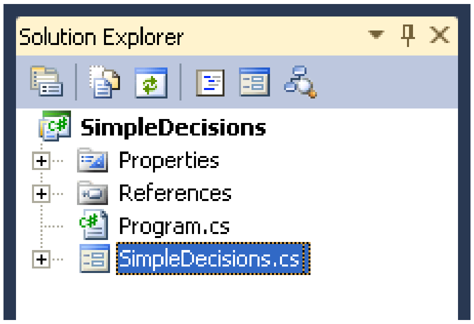
☑ Create directory for solution

☐ Add to source control

The Windows Form Application appears.

Next, we need to rename our Form.

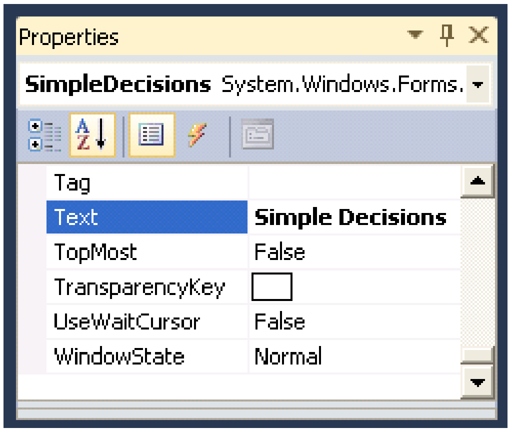
Locate the entry for **Form1.cs** and click on it in the Solution Explorer. Click **Form1.cs** again, and replace it with **SimpleDecisions.cs**. Click 💾 or 💾 to save your changes.



Tip You may be familiar with the **F2** function key as the standard Windows rename key. In Studio, F2 calls the Object Browser. We will discuss the Object Browser later, but if you accidentally press **F2** and get the Browser, simply close the window that appears by clicking ☒.

Now, we'll change the form's title bar, and add a ListBox control ListBox to the form.

Click on the **Form1** form title bar. Find the Text property in the Property Window, change it to **Simple Decisions**, and save your changes 💾.

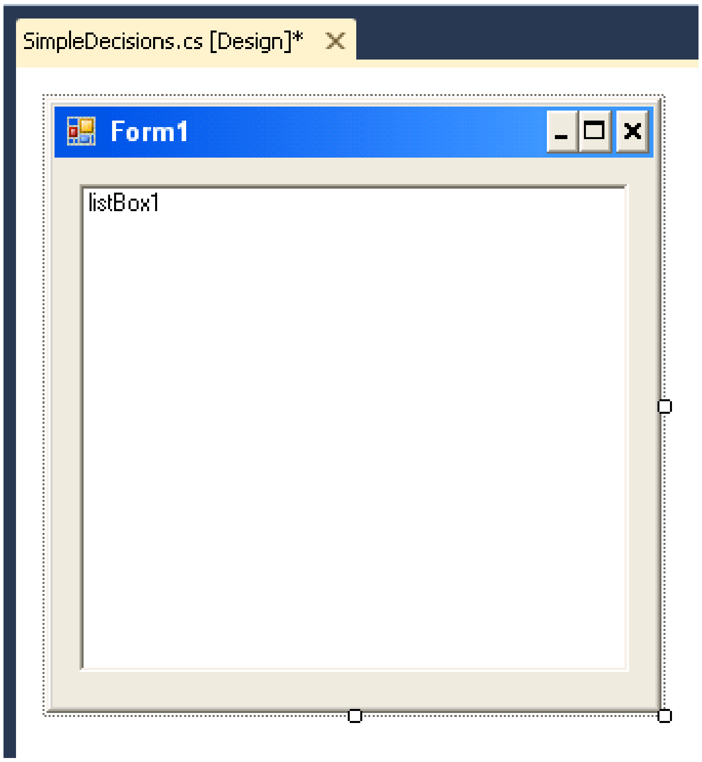


Properties

SimpleDecisions System.Windows.Forms.

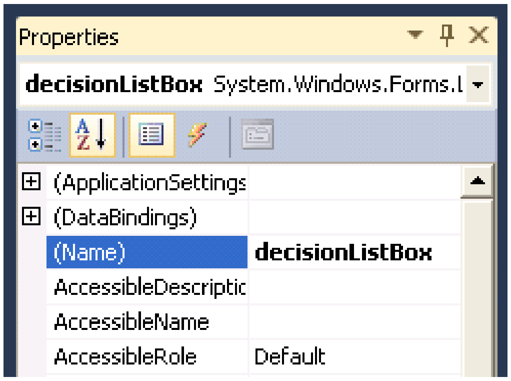
Tag	
Text	**Simple Decisions**
TopMost	False
TransparencyKey	
UseWaitCursor	False
WindowState	Normal

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



Next, rename the ListBox control.

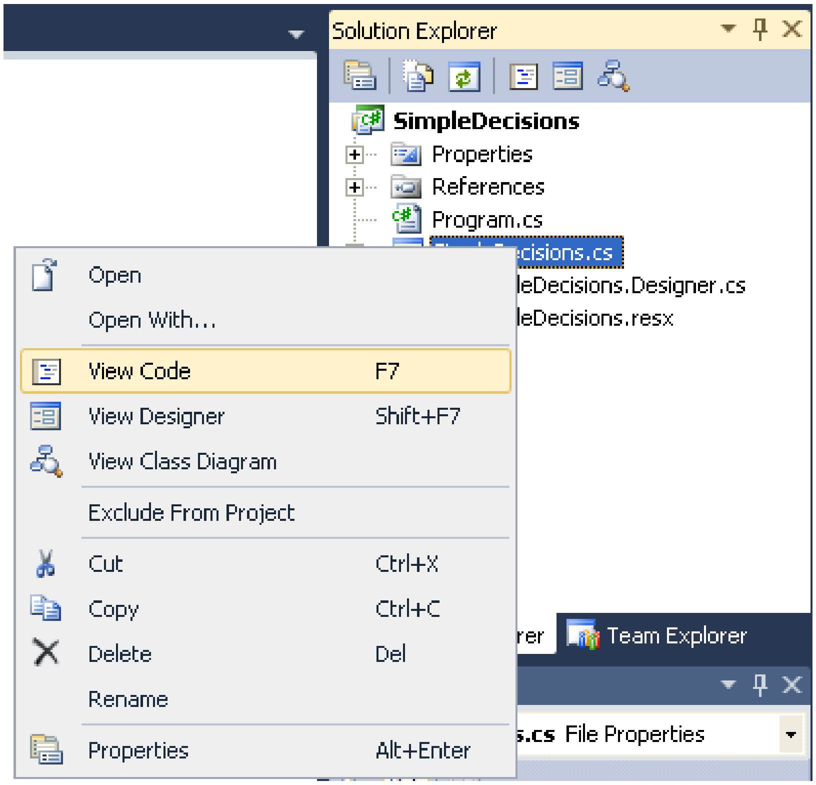
Click the **ListBox** control, and in the **Properties Window**, change the Name property to **decisionListBox**. Save your changes 💾.



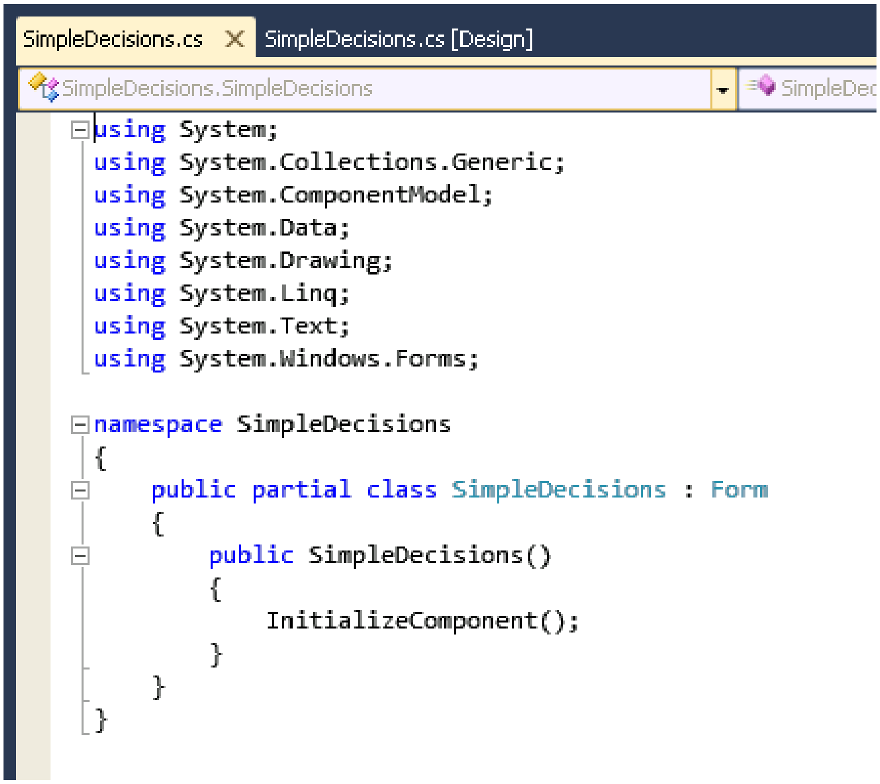
(ApplicationSettings
(DataBindings)
(Name) decisionListBox
AccessibleDescriptic
AccessibleName
AccessibleRole Default

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

Right-click the **SimpleDecisions.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:



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

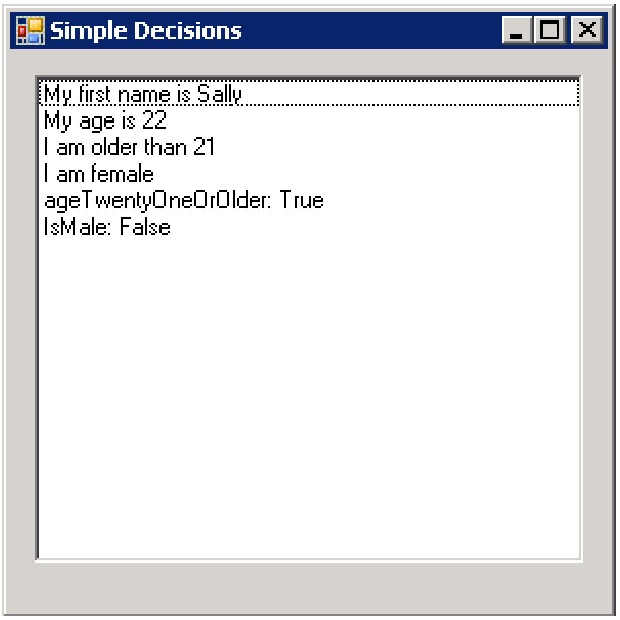
Next, we'll add the C# code to describe Sally, and the if statements. We will display information in our ListBox.

Add the decision-making code to **SimpleDecisions.cs** as shown below. Remember, don't cut and paste! Typing the code yourself will help you to learn the C# language.

CODE TO TYPE: SimpleDecisions.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 SimpleDecisions
{
    public partial class SimpleDecisions : Form
    {
        public SimpleDecisions()
        {
            InitializeComponent();

            // Set data variables.
            string firstName = "Sally";
            string gender = "Female";
            int age = 22;

            // Set result variables.
            bool ageTwentyOneOrOlder = false, isMale = true;

            // Output basic information..
            decisionListBox.Items.Add("My first name is " + firstName);
            decisionListBox.Items.Add("My age is " + age);

            // Determine if age is greater than or equal to 21.
            if (age > 21)  {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
            if (age < 21)
                decisionListBox.Items.Add("I am younger than 21");
            if (age == 21)
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am 21");
            }

            // Determine gender.
            if (gender == "Female")
            {
                isMale = false;
                decisionListBox.Items.Add("I am female");
            }
            if (gender == "Male") decisionListBox.Items.Add("I am male");

            // Output our results variables.
            decisionListBox.Items.Add("ageTwentyOneOrOlder: " + ageTwentyOneOrOl
der);
            decisionListBox.Items.Add("isMale: " + isMale);
        }
    }
}
```

Save your changes 💾, and click ▶ to run it. You should see:



```
My first name is Sally
My age is 22
I am older than 21
I am female
ageTwentyOneOrOlder: True
IsMale: False
```

Now let's discuss how this code works.

OBSERVE: SimpleDecisions.cs

```
.
.
.
            // Set result variables.
            bool ageTwentyOneOrOlder = false, isMale = true;
.
.
.
```

Note that we declared the Boolean variables **ageTwentyOneOrOlder** and **isMale** on the same line. This does save space, but can make your code more difficult to follow, so in the future, we'll declare all variables on separate lines.

You might have noticed that we used different line formatting on different statements, sometimes left out the braces, etc. We will change the code to be consistent, but before we do, let's take a look at this formatting.

C# doesn't really care about spacing. Your code can be on the same line, or different lines, or all smashed together. We format our code purely to make coding—and reading code—easier. We used the various formatting here to illustrate that you can use different formats for if constructs (and other C# constructs that use braces).

OBSERVE: SimpleDecisions.cs

```
.
.
.            // Determine if age is greater than or equal to 21.
            if (age > 21) {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
.
.
.
```

The code above has the opening brace on the same line as the if statement. While it saves you a line of code, and the program still works, it isn't as easy to match up with the closing brace when you try to read the code.

Contrast this formatting with the syntax we've used so far in these lessons, where both the opening and closing braces are on separate lines and in the same column. It is more common to have the opening brace on the same line, but Studio will often change your opening brace placement as you type in a closing brace. So, for these lessons, we'll typically keep the opening and closing braces or their own lines.

But what about the next line, where there are no braces at all?

OBSERVE: SimpleDecisions.cs - No braces

```
.
.
.
            if (age< 21)
                decisionListBox.Items.Add("I am younger than 21");
.
.
.
```

C# allows you to omit the braces if you have only a single line of code to execute after the if statement. Confusing? It can be! How about the line that has the code to be executed on the same line, as shown below?

OBSERVE: No brace, code on same line

```
.
.
.
            if (gender == "Male") decisionListBox.Items.Add("I am male");
.
.
.
```

Again, no braces are needed if there's only one line of code to execute when the if statement is true. Remember, C# doesn't really care about spacing. This second example is even harder to read than the first, but you will see both formats commonly in C# code. Programmers like to save space, but sometimes at the expense of readability. We'd rather have our code be *easily understood* than *brief*, so we suggest you make it a habit to use those braces! Let's redo the code to follow these guidelines. Remove ~~code that looks like this~~ and add code that looks like this:

CODE TO TYPE: SimpleDecisions.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 SimpleDecisions
{
    public partial class SimpleDecisions : Form
    {
        public SimpleDecisions()
        {
            InitializeComponent();

            // Set data variables.
            string firstName = "Sally";
            string gender = "Female";
            int age = 22;

            // Set result variables.
            bool ageTwentyOneOrOlder = false~~, isMale = true~~;
            bool isMale = true;

            // Output basic information.
            decisionListBox.Items.Add("My first name is " + firstName);
            decisionListBox.Items.Add("My age is " + age);

            // Determine if age is greater than or equal to 21.
            if (age > 21) ~~{~~
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
            if (age < 21)
            {
                decisionListBox.Items.Add("I am younger than 21");
            }
            if (age == 21)
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am 21");
            }

            // Determine gender
            if (gender == "Female")
            {
                isMale = false;
                decisionListBox.Items.Add("I am female");
            }
            if (gender == "Male") ~~decisionListBox.Items.Add("I am male");~~
            {
                decisionListBox.Items.Add("I am male");
            }

            // Output our results variables.
            decisionListBox.Items.Add("ageTwentyOneOrOlder: " + ageTwentyOneOrOl
der);
            decisionListBox.Items.Add("isMale: " + isMale);
        }
    }
}
```

Save your changes 💾 and click ▶ to run it again. You should see the same results, but your code is much easier to follow.

Now, finally, let's discuss the actual decision-making in this code.

OBSERVE: Decision-making using if and comparison operators

```
.
.
.
            // Determine if age is greater than or equal to 21.
            if (age > 21) {
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
            if (age < 21)
            {
                decisionListBox.Items.Add("I am younger than 21");
            }
            if (age == 21)
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am 21");
            }

            // Determine gender
            if (gender == "Female")
            {
                isMale = false;
                decisionListBox.Items.Add("I am female");
            }
            if (gender == "Male")
            {
                decisionListBox.Items.Add("I am male");
            }

            // Output our results variables.
            decisionListBox.Items.Add("ageTwentyOneOrOlder: " + ageTwentyOneOrOl
der);
            decisionListBox.Items.Add("isMale: " + isMale);
        }
```

Each **if** statement compares the content of one of Sally's **data variables** to determine the value(s) of one or more of the **result variables**. Based on these results, **additional code adds information to our ListBox**. The ListBox control **decisionListBox** contains a collection object named **Items**, and we call the **Add** method of this collection to add the specified text to the ListBox.

Each of the if statements should be self-explanatory, comparing the value on the left side of the comparison operator to the value on the right side of the operator. If the result is true, we execute the code within the following braces, adding text to the ListBox, and sometimes setting one of the result variables. We don't always need to set the result variables, because each of the result variables has a default value. For example, **isMale** is true by default, so if **gender** == "Female", we set it to false, but if **gender** == "Male", we don't need to set it.

Tip What would happen if we changed the default values of the result variables? We would then have to go back and change our code logic, so you might want to consider setting the result variables in each result block (the code within the braces following the decision statement).

Run it again ▶ and go through the code; make sure you understand why each line added in the ListBox is there!

The if..else Choice

In the previous section, you may have wondered why we had to first ask the question if Sally was Female, and then ask if she was Male, when it would make more sense to ask and deduce: "Is Sally Female? Otherwise, she must be Male." We can do this with the **if..else** syntax. We'll change the gender question in our program to use this new syntax. We'll also modify the age question to use **if..else**, but because there are three possible responses (less than 21, greater than 21, or equal to 21), we'll *nest* an **if..else** within an **if** statement.

Note We'll learn how to apply the **if..else..if** in this nesting scenario later.

Modify the **SimpleDecisions.cs** code as shown below to use the **if..else** syntax (By now I probably don't need to remind you again: *don't copy and paste*!):

CODE TO TYPE: SimpleDecisions.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 SimpleDecisions
{
    public partial class SimpleDecisions : Form
    {
        public SimpleDecisions()
        {
            InitializeComponent();

            // Set data variables.
            string firstName = "Sally";
            string gender = "Female";
            int age = 22;

            // Set result variables.
            bool ageTwentyOneOrOlder = false;
            bool isMale = true;

            // Output basic information.
            decisionListBox.Items.Add("My first name is " + firstName);
            decisionListBox.Items.Add("My age is " + age);

            // Determine if age is greater than or equal to 21.
            if (age > 21)
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
            else
            {
            if (age < 21)
            {
                ageTwentyOneOrOlder = false;
                decisionListBox.Items.Add("I am younger than 21");
            }
            ~~if (age == 21)~~
            else
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am 21");
            }
            }

            // Determine gender
            if (gender == "Female")
            {
                isMale = false;
                decisionListBox.Items.Add("I am female");
            }
            ~~if (gender == "Male")~~
            else
            {
                isMale = true;
                decisionListBox.Items.Add("I am male");
            }

            // Output our results variables.
            decisionListBox.Items.Add("ageTwentyOneOrOlder: " + ageTwentyOneOrOl
der);
               decisionListBox.Items.Add("isMale: " + isMale);
          }
     }
}
```

Note Visual Studio might change the indentation for you as you type the nested ifs and elses. The new indentation will be reflected in subsequent examples here.

Save your changes 💾 and run it again ▶; you should see the same results as before.

Let's discuss our changes.

OBSERVE: Decision making using if..else syntax

```
.
.
.
               // Determine if age is greater than or equal to 21.
               if (age > 21)
               {
                    ageTwentyOneOrOlder = true;
                    decisionListBox.Items.Add("I am older than 21");
               }
               else
               {
                    if (age < 21)
                    {
                         ageTwentyOneOrOlder = false;
                         decisionListBox.Items.Add("I am younger than 21");
                    }
                    else
                    {
                         ageTwentyOneOrOlder = true;
                         decisionListBox.Items.Add("I am 21");
                    }
               }

               // Determine gender
               if (gender == "Female")
               {
                    isMale = false;
                    decisionListBox.Items.Add("I am female");
               }
               else
               {
                    isMale = true;
                    decisionListBox.Items.Add("I am male");
               }
.
.
.
```

The nested age logic might take a bit of thought; the **first if asks if the age is greater than 21**. **If the age is not greater than 21**, then it is either less than 21 or exactly 21. So we next ask **if the age is less than 21**, and **if not**, we indicate the age is 21. You can read this code almost like a (somewhat complex) English sentence:

- **if the age is over 21:**
  - **ageTwentyOneOrOlder is true**
  - **add "I am older than 21" to the decisionListBox**
- **else (age is *not* over 21):**
  - **if age is less than 21:**

        - **ageTwentyOneOrOlder is false**
        - **add "I am younger than 21" to the decisionListBox**
    - **else (age is not less than 21):**
        - **ageTwentyOneOrOlder is true**
        - **add "I am 21" to the decisionListBox**

The gender test is now a simple question: is the gender female? if not, it is male.

We also added the **result variables** to every possible result. Regardless of the default values of these variables, they will be correctly set in all possible result paths.

And and Or Operators

You can use *and* and *or* to chain two or more questions together, much like you use them in English. For example, what if we wanted to find out if a person's age was greater than or equal to 18, but less than 21? We could create two **if** statements, and ask: are you 18 or older? Then, we could ask: are you younger than 21? It is very common to combine multiple questions like this.

Here is the C# syntax for each operator, and a quick example.

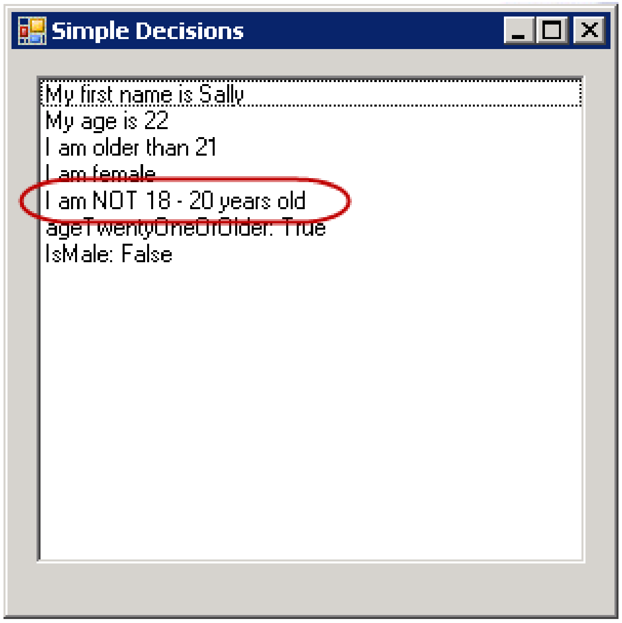
OR - || (two pipe symbols)	if (color == "RED" || color == "BLUE")	The color is RED or BLUE.
AND - && (two ampersand symbols)	if (age >= 18 && age < 21)	age is greater than or equal to 18 and less than 21 (in other words, the age is 18, 19, or 20).

So, let's modify the code one more time with this additional age test:

CODE TO TYPE: SimpleDecisions.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 SimpleDecisions
{
    public partial class SimpleDecisions : Form
    {
        public SimpleDecisions()
        {
            InitializeComponent();

            // Set data variables.
            string firstName = "Sally";
            string gender = "Female";
            int age = 22;

            // Set result variables.
            bool ageTwentyOneOrOlder = false;
            bool isMale = true;

            // Output basic information.
            decisionListBox.Items.Add("My first name is " + firstName);
            decisionListBox.Items.Add("My age is " + age);

            // Determine if age is greater than or equal to 21.
            if (age > 21)
            {
                ageTwentyOneOrOlder = true;
                decisionListBox.Items.Add("I am older than 21");
            }
            else
            {
                if (age < 21)
                {
                    ageTwentyOneOrOlder = false;
                    decisionListBox.Items.Add("I am younger than 21");
                }
                else
                {
                    ageTwentyOneOrOlder = true;
                    decisionListBox.Items.Add("I am 21");
                }
            }

            // Determine gender
            if (gender == "Female")
            {
                isMale = false;
                decisionListBox.Items.Add("I am female");
            }
            else
            {
                isMale = true;
                decisionListBox.Items.Add("I am male");
            }

            // Check if age falls between 18 and 20.
            if (age >= 18 && age < 21)
            {
                decisionListBox.Items.Add("I am 18 - 20 years old");
            }
            else
            {
                decisionListBox.Items.Add("I am NOT 18 - 20 years old");
            }

            // Output our results variables.
            decisionListBox.Items.Add("ageTwentyOneOrOlder: " + ageTwentyOneOrOl
der);
            decisionListBox.Items.Add("isMale: " + isMale);
        }
    }
}
```

Save your changes and run it . You should see something like this, with the new output line displayed in the ListBox:



```
My first name is Sally
My age is 22
I am older than 21
I am female
I am NOT 18 - 20 years old
ageTwentyOneOrOlder: True
IsMale: False
```

Let's discuss it.

OBSERVE: Decision-making with Boolean operators

```
.
.
.
            // Check if age falls between 18 and 20
            if (age >= 18 && age < 21)
            {
                decisionListBox.Items.Add("I am 18 - 20 years old");
            }
            else
            {
                decisionListBox.Items.Add("I am NOT 18 - 20 years old");
            }
.
.
.
```

The if statement checks to see if the **age is 18 or greater**, **AND** **is less than 21**. Sally's age is 22, so this evaluates false, and so we continue to the code block after **else** and display that her age is not between 18 and 20.

Next

That's it for this lesson! *IF* you're ready to move on, && you don't forget to do the projects and quizzes, we'll see you in the next lesson!

Methods

What's in a Method?

Previously, we've seen *methods* as we've coded. A method is a fundamental software programming element used to *simplify the coding process*, *encapsulate related functionality*, and *remove redundancy*.

Let's review the DrawCircle method of the Circle class, and fully explain what exactly we're doing when we create or use (or in computer-speak, *call*) a method. Look at the following code containing the DrawCircle method. The comments have been removed.

Circle.cs DrawCircle method

```
public void DrawCircle(Graphics drawingArea, int x, int y)
{
    if (drawingArea != null) {
        drawingArea.DrawEllipse(new Pen(Color.Red), x, y, _circleDiameter, _circleDiameter);
    }
}
```

The name of the method is **DrawCircle**; we use this name in our code whenever we want to call this method. Preceding the name is a keyword known as the *access modifier*, that indicates the accessibility of the method. In C#, there are four different access modifiers: **public**, **private**, **protected**, and **internal**. In our example, we used **public**, which means that this method may be used by any other code within the same assembly, or another assembly that references it. **Protected** methods can only be accessed by code within the same class, or in a class that is derived from the same class. **Private** methods can only be accessed by code within the same class. **Internal** methods can be accessed by any code within the same assembly, but not from another assembly.

Note

We will discuss more about what exactly an assembly is when we discuss the .Net framework. For now, you can view an assembly as your entire program, or other separate individual programs that you include with the **using** statement.

Tip

Although we did not discuss access modifiers when we discussed variables, you may remember seeing **private** in front of some of the variables used in previous lessons. Class variables use the same access modifiers as methods. Variables used within a method do not use access modifiers. We will discuss class variables more later.

Following the access modifier is the return type of the method. In our example, the return type is **void**, which is a keyword meaning that this method does not return anything. We will discuss what it means to "return" from a method shortly.

Following the name of the method are opening and closing parentheses, with a list of parameters (separated by commas) that contain information to be passed to the method when the method is used. In our example, there are three parameters: (**Graphics drawingArea**, **int x**, **int y**). Each parameter includes its data type: **Graphics**, **int**, and **int**. The parameter datatype list composes the *signature* of a method. The name of each parameter should help anyone who calls the method to know what the parameter should contain, and is also the variable that is used within the method itself. Parameter variable names use *camel case*, and should be descriptive. The **drawingArea** parameter, of datatype **Graphics**, seems sufficiently descriptive, indicating a parameter that contains the drawing surface to be drawn on. The other two parameters, **x** and **y**, are not very descriptive unless you connect them to the x and y axes in math. Both of these parameters are of data type **int**. More descriptive terms might have been **horizontalLocation** (for **x**) and **verticalLocation** (for **y**).

To summarize, methods are composed of:

Access modifier, **Return type** **Method name** (parameter list: **Data type** **Parameter name**)

Adding a Class Method

Let's return to our Circle class, and add a new method to the Circle class that will set the circle color.

Open the MyCircle project, and modify the Circle.cs class as shown below.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object.

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        ~~// Private member property _circleDiameter, with default of 0.~~
        private int _circleDiameter = 0;
        private Color _circleColor = Color.Red;

        // ** Accessors **
        // Public accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
              // Sets the _circleDiameter. The value variable is the
              // value the user specified when calling the accessor.
              set
              {
                    _circleDiameter = value;
              }
              // Returns the current value of _circleDiameter.
              get
              {
                    return _circleDiameter;
              }
        }

        // ** Public methods **
        // Public method (behavior) to draw a circle.
        public void DrawCircle(Graphics drawingArea, int x, int y)
        {
              // Make sure the drawingArea is valid.
              if (drawingArea != null)
              {
                    // Call the DrawEllipse method of the drawing area
                    // creating a dynamic Pen for a color, using the x
                    // and y coordinates to specify starting point at the
                    // top-left corner, and using the m_intDiameter
                    // member property for both the height and width.
                    drawingArea.DrawEllipse(new Pen(~~Color.Red~~ _circleColor), x, y, _cir
cleDiameter, _circleDiameter);
              }
        }

        // Get current circle color.
        public Color GetColor()
        {
              return _circleColor;
        }

        // Set circle color.
        public void SetColor(Color circleColor)
        {
              _circleColor = circleColor;
        }
    }
}
```

Click Save All , and then click Debug to execute the program. When the form appears, enter a number (such as 50), and click **Create Circle**. It should work just as it did before, but we've made some important changes—let's discuss them.

OBSERVE: Circle.cs

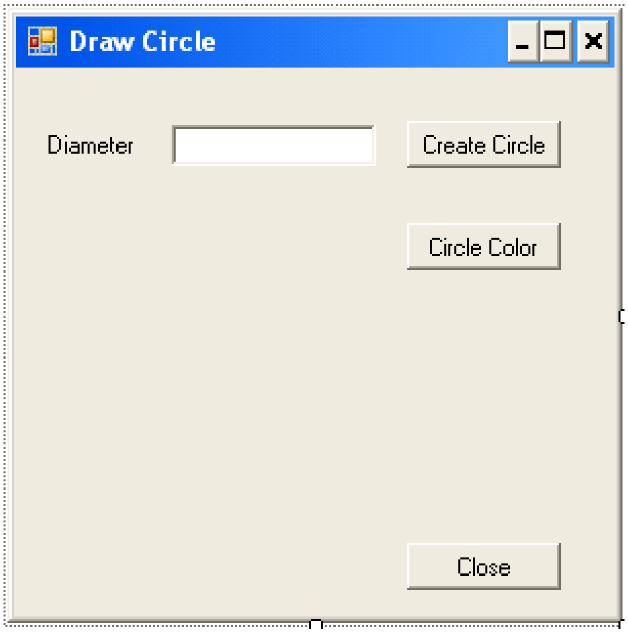
```
.
.
.
        class Circle
        {
            // ** Properties **
            private int _circleDiameter = 0;
            private Color _circleColor = Color.Red;

            .
            .
            .

            // Get current circle color.
            public Color GetColor()
            {
                  return _circleColor;
            }

            // Set circle color.
            public void SetColor(Color circleColor)
            {
                  _circleColor = circleColor;
            }
        }
.
.
.
```

First, we removed the comment before the _circleDiameter class property, because the descriptive property name should make its purpose obvious. Second, we added another class property (remember, a property is a variable) **_circleColor** of data type **Color**, with a default of **Color.Red**, to preserve the circle color. Then, we added the **GetColor()** method to get the current value of the **_circleColor** property, and **SetColor(Color circleColor)** to set the **_circleColor** property.

Note If you're wondering if we could have used accessors for this property, congratulations! The answer is "Yes!" In fact, we will modify the code shortly to change these methods into class property accessors.

Now that our Circle class supports a color property, we need to change the MyCircle Form to support changing the color as well. We will add a button that will bring up a dialog box where the user can select a color. We'll also add a Close button to make it easier to close the MyCircle program.

Click on the **MyCircle.cs [Design]** tab to open the Form Designer. If that tab is not visible, right-click on the **MyCircle.cs** entry in the Solution Explorer, and select **View Designer**. Add two new buttons to the form (either by double-clicking the **Button** control in the Toolbox, or by dragging the **Button** onto the form from the Toolbox). Change the Name property of one button to **circleColorButton**, and the Text property to **Circle Color**. Change the Name property of the other button to **closeButton**, and the Text property to **Close**. When you finish, click .



Calling a Class Method

Now let's add the code to add color selection to the MyCircle Form, and enable the Close button to close the program.

Modify the MyCircle Form code as shown below.

Tip Remember that Studio will generate the event handler method for a button automatically if you double-click on the button. You would need to click on each of the new Form buttons.

CODE TO TYPE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        ~~// Circle object~~
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            // Variable for TryParse result.
            int diameter = 0;

            // Try to convert the entered text to an integer, and assign result to
            // Circle diameter property using class set accessor
            // If the TryParse method fails, diameter will be 0.
            if (Int32.TryParse(circleDiameterTextbox.Text, out diameter) == true)
            {
                _myCircle.Diameter = diameter;
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("New diameter is: {0}", diameter);
                // Get a graphics canvas to draw on, and call the DrawCircle public met
hod
                // of Circle class to draw the actual red circle.
                _myCircle.DrawCircle(CreateGraphics(), 10, 10);
            } else {
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("Non-numeric entry: {0}", circleDiameterTextbox.Text)
;
            }
        }

        private void circleColorButton_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            colorDialog.AllowFullOpen = false;
            colorDialog.AnyColor = true;
            colorDialog.SolidColorOnly = true;
            colorDialog.Color = _myCircle.GetColor();

            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                _myCircle.SetColor(colorDialog.Color);
            }
        }

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

Click Save All , and then click Debug to execute the program. When the form appears, enter a number (such as 50), and click **Circle Color**. A dialog appears where you can choose from a number of colors; click one and then click **OK**. Then click **Create Circle**. It should draw a circle with the diameter you entered, in the color you selected! If you left the circle diameter the same as the last time you ran the program, you might have seen the circle first drawn in red, then drawn again in the new color you selected.

Click the **Close** button to close the MyCircle program and return to the Studio IDE.

Let's discuss the modified code.

OBSERVE: MyCircle.cs

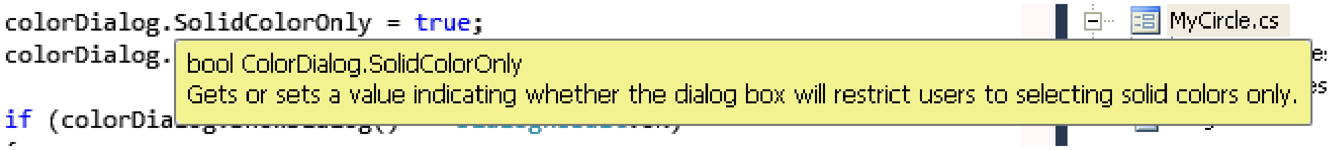
```
.
.
.
        private void circleColorButton_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            colorDialog.AllowFullOpen = false;
            colorDialog.AnyColor = true;
            colorDialog.SolidColorOnly = true;
            colorDialog.Color = _myCircle.GetColor();

            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                _myCircle.SetColor(colorDialog.Color);
            }
        }

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

These class methods are button event handlers for the mouse click event, which means that if a user clicks on the button, the code for that button is called. The **closeButton_Click** simply calls the **Close()** method (the parentheses indicate a method), a method of the MyCircle Form.

The other method, **circleColorButton_Click**, displays a color dialog box where the user can select a color. First, we create a method, or local, variable (a variable that is only available within the method) named **colorDialog**, and assign to that an instance of the ColorDialog object created using the **new** keyword. **ColorDialog** is an **object data type**.

We also set a number of **ColorDialog** properties of the **colorDialog** instance. The **AllowFullOpen** property is set to **false**, which prevents the user from adding custom colors. The **AnyColor** property, set to **true**, allows the display of all available colors in the set of basic colors. The **SolidColorOnly** property, set to **true**, means that only solid colors will be displayed. To see a description for any of these methods, hover with your mouse pointer over the method; Intellisense pops up a description.

Hover your mouse pointer over the **SolidColorOnly** property, and see the Intellisense popup:

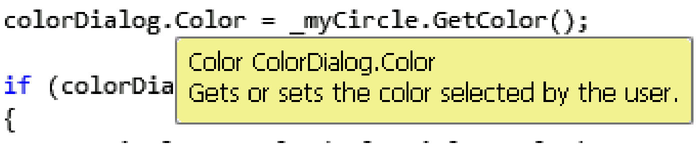


```
colorDialog.SolidColorOnly = true;
colorDialog.

if (colorDia
```

bool ColorDialog.SolidColorOnly
Gets or sets a value indicating whether the dialog box will restrict users to selecting solid colors only.

The last property, **Color**, sets the default color of the **colorDialog** instance. We want the default color to be the current color of the Circle class, so we use the **_myCircle** class instance variable to call the **GetColor** method of the Circle class. This method returns a **Color** object, which is the data type of the **colorDialog Color** property. How do we know the data type? Intellisense!

Hover your mouse pointer over the **Color** property; the Intellisense popup appears:



```
colorDialog.Color = _myCircle.GetColor();

if (colorDia
{
```

Color ColorDialog.Color
Gets or sets the color selected by the user.

After setting the last property, we have an if statement. This statement first triggers display of the color dialog box using **colorDialog.ShowDialog**(), and then after the color dialog box is closed, tests whether or not the value returned is equal to **DialogResult.OK** using the double equals (==). The return value is only set to **OK** if the user clicked the **OK** button in the color dialog box, which would mean that they want to set the color. With the if statement being true, we can then call the **SetColor** method of the **_myCircle** instance. We can see how a parameter is used when you call a method: by supplying the parameter information within the parenthesis after the method name. In our code, the **colorDialog.Color** is the parameter *passed* to the **SetColor()** method.

Note You might wonder how **colorDialog.ShowDialog**() effectively paused until the user dismissed the color dialog box. Dialog boxes come in two flavors: *modal* and *modeless*. A *modal* dialog waits until the user dismisses it, making it impossible to return the focus to the program that called it until that happens. In our case, the MyCircle program calls the color dialog, and you cannot return to the MyCircle program until the color dialog is dismissed. A *modeless* dialog does not have this restriction, so the color dialog must implement a *modal* dialog.

Class Method or Class Accessor?

These new methods could have been implemented as accessors. When do you use an accessor to set a class property, and when do you use a method? The answer is simple: use an accessor whenever you can! Class accessors have the advantage of not needing to specify the parameters when you define them. A reason not to use an accessor is if you want more control over the parameters you might want to pass. Other reasons for accessors or methods will be included as the lessons continue.

So, let's transform those methods into accessors.

Modify Circle.cs as shown below:

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        private int _circleDiameter = 0;
        private Color _circleColor = Color.Red;

        // ** Accessors **
        // Public accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
            ~~// Sets the _circleDiameter. The value variable is the~~
            ~~// value the user specified when calling the accessor.~~
            set
            {
                _circleDiameter = value;
            }
            ~~// Returns the current value of _circleDiameter.~~
            get
            {
                return _circleDiameter;
            }
        }

        // Public accessors for getting/setting member property _circleColor.
        public Color Color
        {
            set
            {
                _circleColor = value;
            }
            get
            {
                return _circleColor;
            }
        }

        // ** Public methods **
        // Public method (behavior) to draw a circle.
        public void DrawCircle(Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid.
            if (drawingArea != null)
            {
                // Call the DrawEllipse method of the drawing area
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the m_intDiameter
                // member property for both the height and width.
                drawingArea.DrawEllipse(new Pen(_circleColor), x, y, _circleDiameter, _
circleDiameter);
            }
        }

        ~~// Get current circle color.~~
        ~~public Color GetColor()~~
        ~~{~~
            ~~return _circleColor;~~
        ~~}~~
        ~~// Set circle color~~
        ~~public void SetColor(Color circleColor)~~
        ~~{~~
            ~~_circleColor = circleColor;~~
        ~~}~~
    }
}
```

Save it 💾, but don't run it yet! Let's discuss the modified code first.

OBSERVE: Circle.cs

```
.
.
.
        // ** Properties **
        private int _circleDiameter = 0;
        private Color _circleColor = Color.Red;

        // ** Accessors **
        // Accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
            set
            {
                _circleDiameter = value;
            }
            get
            {
                return _circleDiameter;
            }
        }

        // Public accessors for getting/setting member property _circleColor.
        public Color Color
        {
            set
            {
                _circleColor = value;
            }
            get
            {
                return _circleColor;
            }
        }
.
.
.
```

Yes, we did remove more comments near the accessors, again because the descriptive accessor names should be self-explanatory. Certainly one potentially confusing section is **Color** **Color**! Two Colors right next to each other? Is that a typo? No, it isn't. The first **Color** is the data type for the accessor: both the return type of the get accessor, and the parameter data type for the set accessor. The second **Color** is the accessor name. A bit confusing, but **Color** is the best name for this accessor, and mimics the other methods we've seen that use Color.

Once you changed the Circle class, Studio alerted you to a number of errors. Why? Well, two reasons. First, the methods were called SetColor and GetColor before, and we've changed both of them to a single Color name. Also, the previous implementation used parameters, which means you called it using the method name AND any parameters within parentheses (for example, SetColor(**colorDialog.Color**)). Accessors are used like variables, where you can assign values, so we need to change the syntax we use with SetColor and GetColor to the assignment format.

Modify the MyCircle.cs code as shown below.

CODE TO TYPE: MyCircle.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 MyCircle
{
    public partial class MyCircle : Form
    {
        private Circle _myCircle;

        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle();
        }

        private void createCircleButton_Click(object sender, EventArgs e)
        {
            // Variable for TryParse result.
            int diameter = 0;

            // Try to convert the entered text to an integer, and assign result to
            // Circle diameter property using class set accessor.
            // If the TryParse method fails, diameter will be 0.
            if (Int32.TryParse(circleDiameterTextbox.Text, out diameter) == true)
            {
                _myCircle.Diameter = diameter;
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("New diameter is: {0}", diameter);
                // Get a graphics canvas to draw on, and call the DrawCircle public met
hod
                // of Circle class to draw the actual red circle
                _myCircle.DrawCircle(CreateGraphics(), 10, 10);
            } else {
                // Output text to Studio console so we can see what happened.
                Console.WriteLine("Non-numeric entry: {0}", circleDiameterTextbox.Text)
;
            }
        }

        private void circleColorButton_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            colorDialog.AllowFullOpen = false;
            colorDialog.AnyColor = true;
            colorDialog.SolidColorOnly = true;
            colorDialog.Color = _myCircle.~~Get~~Color~~()~~;

            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                _myCircle.~~SetColor(colorDialog.Color)~~;
                _myCircle.Color = colorDialog.Color;
            }
        }

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

Click . Let's discuss the modified code.

OBSERVE: MyCircle.cs

```
.
.
.
        private void circleColorButton_Click(object sender, EventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            colorDialog.AllowFullOpen = false;
            colorDialog.AnyColor = true;
            colorDialog.SolidColorOnly = true;
            colorDialog.Color = _myCircle.Color;

            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                _myCircle.Color = colorDialog.Color;
            }
        }
.
.
.
```

Only two lines of code had to change, both in the **circleColorButton_Click** method. The first change was to setting the default dialog color **colorDialog.Color**, now calling the **Color get** property of the **_myCircle** Circle class instance. The second change was to call the **Color set** property of the **_myCircle** Circle class instance to set the circle color. In both cases, the method calls, using parentheses, were changed to property assignments.

Let's test our changes and make sure we get the same results as before!

Click ▶ to execute the program. When the form appears, enter a number (such as 50) in the textbox and click **Create Circle**. Then, click **Circle Color**, select a different color in the color dialog and click **OK**, and click **Create Circle** again. When you finish observing the results, click **Close**.

Constructors

We have another type of class method to introduce to you: constructors. A constructor class method is a method that is used when you first create an instance of an object. For example, in MyCircle.cs, we created an instance of (*instantiated*) the **Circle** class, and assigned it to a **class variable**, using this code:

OBSERVE: Object Instantiation and Assignment

```
    _myCircle = new Circle();
```

The **_myCircle** is the MyCircle Form class variable. The **new** keyword creates an instance of an object; in this case, an instance of the **Circle** class. You'll notice that **Circle()** includes parentheses. Does that mean that we're calling a method of the Circle class? Yes, we are! We're calling a constructor! But wait, we didn't write any code that looked like a constructor. So what is going on?

Every C# class must have a constructor, and if one is not explicitly written in the code, a default constructor is provided when the project is built. A default constructor contains no parameters, which matches the format of the **Circle()**: no parameters are included within the parentheses. So, we're going to provide our own default constructor, and a second constructor that takes a single parameter, a default value for the diameter of the circle.

Modify the Circle.cs code as shown below.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;   // Added to gain access to the Pen object

namespace MyCircle
{
    class Circle
    {
        // ** Properties **
        private int _circleDiameter = 0;
        private Color _circleColor = Color.Red;

        // ** Constructors **
        // Default constructor.
        public Circle()
        {
            // Do nothing
        }

        // Set default circle diameter constructor.
        public Circle(int defaultCircleDiameter)
        {
            if (defaultCircleDiameter >= 0)
            {
                _circleDiameter = defaultCircleDiameter;
            } else {
                _circleDiameter = 0;
            }
        }

        // ** Accessors **
        // Public accessors for getting and setting member property _circleDiameter.
        public int Diameter
        {
            set
            {
                _circleDiameter = value;
            }
            get
            {
                return _circleDiameter;
            }
        }

        // Public accessors for getting/setting member property _circleColor.
        public Color Color
        {
            set
            {
                _circleColor = value;
            }
            get
            {
                return _circleColor;
            }
        }

        // ** Public methods **
        // Public method (behavior) to draw a circle.
        public void DrawCircle(Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid.
            if (drawingArea != null)
            {
                // Call the DrawEllipse method of the drawing area
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the m_intDiameter
                // member property for both the height and width.
                drawingArea.DrawEllipse(new Pen(_circleColor), x, y, _circleDiameter, _
circleDiameter);
            }
        }
    }
}
```

Click , but don't run it yet. Let's discuss the modified code.

OBSERVE: Circle.cs

```
.
.
.
        // ** Constructors **
        // Default constructor
        public Circle()
        {
            // Do nothing
        }

        // Set default circle diameter constructor
        public Circle(int defaultCircleDiameter)
        {
            if (defaultCircleDiameter >= 0)
            {
                _circleDiameter = defaultCircleDiameter;
            } else {
                _circleDiameter = 0;
            }
        }
.
.
.
```

Constructors have the same name as the class they're in. In our example, both of the constructors are named **Circle**. How does C# know which constructor to use? As we've discussed before, each method has a unique *signature*, which includes the name of the method and the list of parameters by data type. The default constructor has no parameters, so its signature is **Circle()**. The second constructor **requires an integer**, so its signature is **Circle(int)**.

Note Using the same name for methods, including constructors, with different signatures, is known as *overloading* in object-oriented terminology.

Since our second constructor has an integer parameter, we add code to the constructor that makes sense based on our Circle class: we do not allow the circle diameter to be less than zero.

Finally, notice that something is missing from a constructor method that other methods include: a return data type. In both of the constructors, the constructor name, **Circle**, follows the **public** access modifier. Adding a return data type such as void is a common mistake.

Note Can a constructor be private? It turns out it can! However, if you change your *default* constructor to private, you won't be able to instantiate an instance of the class, so if you see a warning that indicates you can't create the class, check your access modifier!

So, shall we try to use the new constructor with the **Circle**(**int**) signature? For now, the convenience of being able to set a default circle diameter is really not that important to our code, so we'll simply change the code that instantiates the Circle object to pass in a zero, forcing the code to call the non-default constructor, rather than the default constructor.

Modify the MyCircle.cs code as shown below (note that we only show the method you need to modify).

CODE TO TYPE:

```
.
.
.
        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            ~~_myCircle = new Circle();~~
            _myCircle = new Circle(0);
        }
.
.
.
```

Click 💾. Click ▶ to execute the program. It should work exactly as it did before. Let's discuss the modified code.

OBSERVE: MyCircle.cs

```
.
.
.
        public MyCircle()
        {
            InitializeComponent();
            // Create new Circle object, and assign to class member property.
            _myCircle = new Circle(0);
        }
.
.
.
```

As you can see, the only change was to pass **0** as a parameter to the constructor, forcing the code to not use the default (parameterless) constructor.

Note You might have noticed that the public **MyCircle()** has the same format as a constructor. Why? MyCircle.cs contains the MyCircle Form, also a class, so the public **MyCircle()** is indeed a constructor!

Next

That's it for this lesson! Remember to complete any associated projects, then quizzes.

The .NET Framework

What is .NET?

So, what exactly is .NET? Microsoft .NET is what is known as a *software framework*. You can search the Internet for the history of .NET and frameworks, including the Wikipedia .NET Framework article, but for this lesson, the .NET Framework is a software development platform that includes a *runtime environment*, a *comprehensive base class library*, *comprehensive data type definitions and programming constructs*, and support for a myriad of programming languages that adhere to the .NET specifications, essentially delivering a *language-independent model*. You'll notice that we didn't indicate that .NET is a Windows-only development platform. Certainly earlier versions of .NET were primarily targeted at the Windows operating system, but the .NET specification may be implemented on any operating system, and in fact implementations exist for the Mac, iPhone, and different versions of Linux and Unix. And even though this course is about C#, a language especially designed for .NET by Microsoft, numerous other .NET languages exist, as evidenced by the list on Wikipedia.

CLR, CTS, and CLS

At the heart of a computer is the central processing unit (CPU, or simply "the processor"). No matter what computer language is used, eventually, the code must be converted into a form that the processor can understand and execute. Traditional software development involved writing code in a human-understandable computer language, and *compiling* that code into machine-understandable binary code native to the operating system and processor. To gain operating system and platform independence, .NET includes a runtime environment known as the *Common Language Runtime (CLR)*. The CLR embodies the *Common Type System (CTS)* that fully defines all of the possible programming constructs and data types the CLR supports.

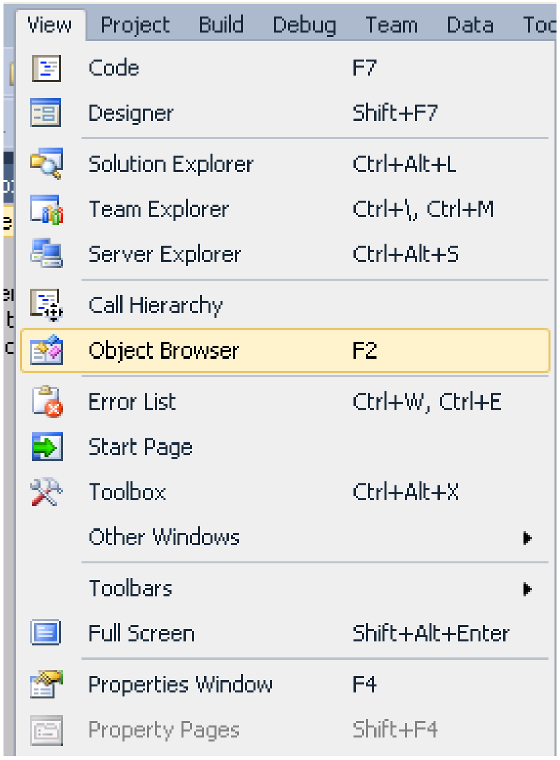
Each .NET language is not required to fully implement the entire CTS, but must define a subset of the specification known as the *Common Language Specification (CLS)*. To ensure any .NET code you write is platform- and language-independent, that language must fully expose all CLS features.

Built on top of the CLR is a comprehensive set of base class libraries available to all .NET programming languages. These class libraries, known as *assemblies*, encapsulate a myriad of functionality for writing software applications, from language primitives like variables and file access, to database access, security, and Windows Forms.

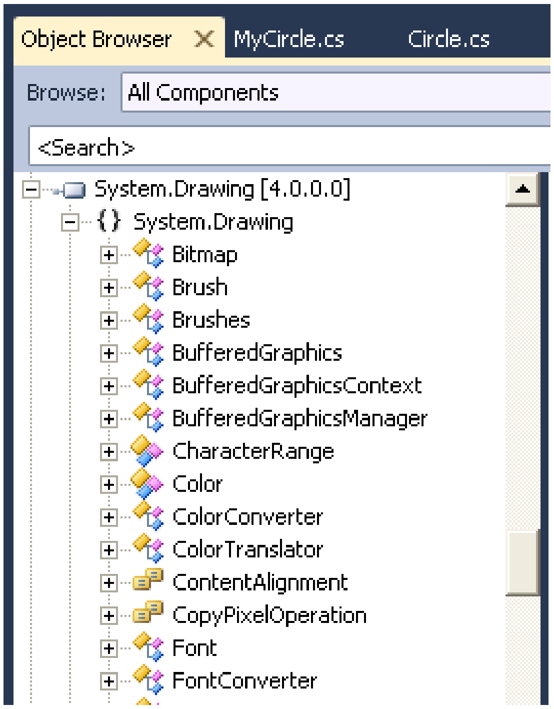
Using the Object Browser

Let's use Visual Studio to explore some of the base class libraries.

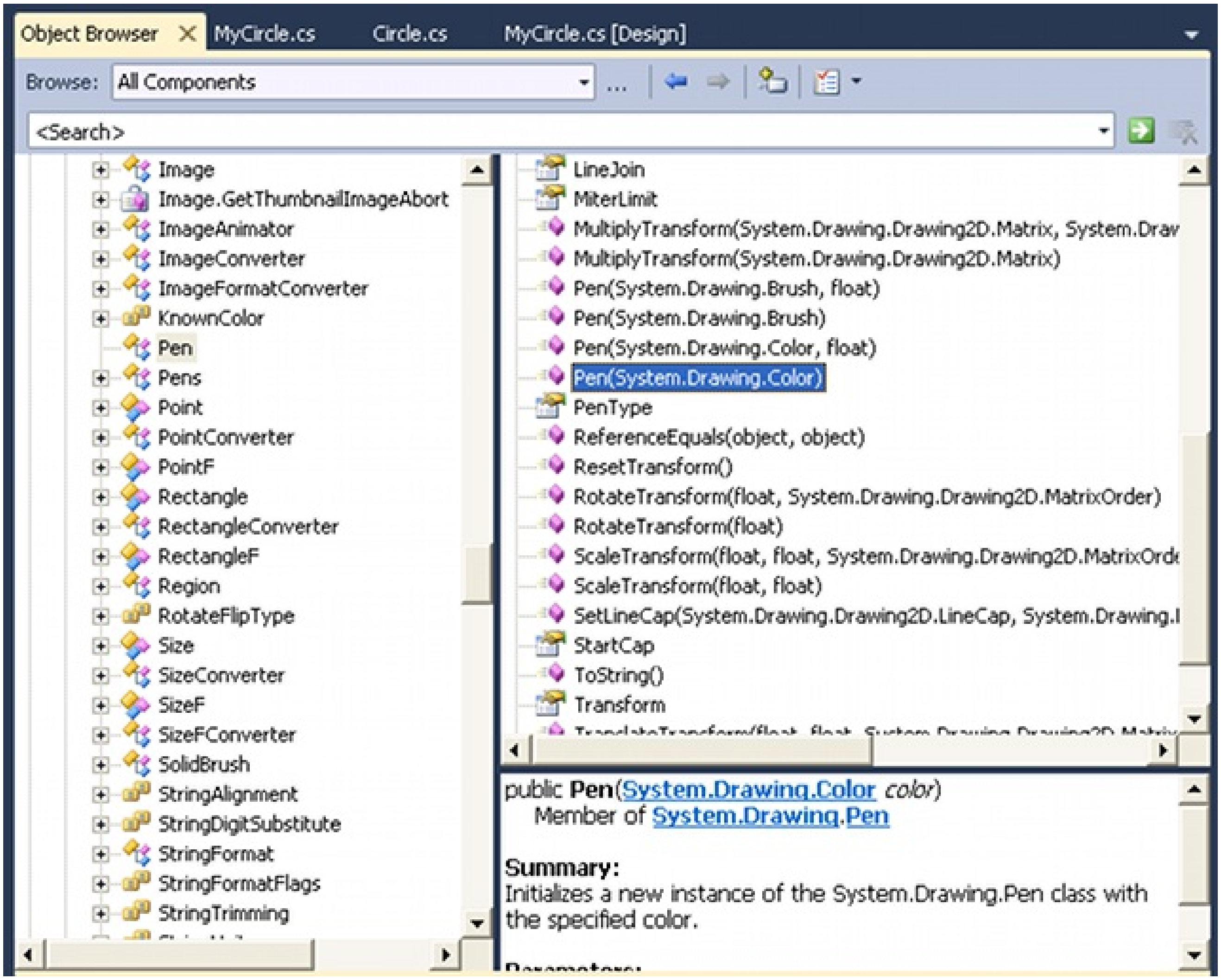
Open the MyCircle solution, and select **View | Object Browser** (or press the **F2** shortcut key).



Navigate down the Object Browser tree to System.Drawing. Click the **+** icon to expand the tree to expose the elements (methods, properties, etc.) of this assembly. Click the **+** next to the System.Drawing namespace (delineated by the braces {}):



Scroll further down the Object Browser tree, staying within the System.Drawing expanded list, to find and select the **Pen** item. In the right pane of the Object Browser, scroll down until you find the **Pen(System.Drawing.Color)** entry.


Object Browser
MyCircle.cs
Circle.cs
MyCircle.cs [Design]
Browse:
All Components
<Search>
Image
Image.GetThumbnailImageAbort
ImageAnimator
ImageConverter
ImageFormatConverter
KnownColor
Pen
Pens
Point
PointConverter
PointF
Rectangle
RectangleConverter
RectangleF
Region
RotateFlipType
Size
SizeConverter
SizeF
SizeFConverter
SolidBrush
StringAlignment
StringDigitSubstitute
StringFormat
StringFormatFlags
StringTrimming
LineJoin
MiterLimit
MultiplyTransform(System.Drawing.Drawing2D.Matrix, System.Draw
MultiplyTransform(System.Drawing.Drawing2D.Matrix)
Pen(System.Drawing.Brush, float)
Pen(System.Drawing.Brush)
Pen(System.Drawing.Color, float)
Pen(System.Drawing.Color)
PenType
ReferenceEquals(object, object)
ResetTransform()
RotateTransform(float, System.Drawing.Drawing2D.MatrixOrder)
RotateTransform(float)
ScaleTransform(float, float, System.Drawing.Drawing2D.MatrixOrde
ScaleTransform(float, float)
SetLineCap(System.Drawing.Drawing2D.LineCap, System.Drawing.I
StartCap
ToString()
Transform

public **Pen**(System.Drawing.Color *color*)
Member of System.Drawing.Pen

Summary:
Initializes a new instance of the System.Drawing.Pen class with the specified color.

You might remember using the Pen object in the MyCircle project when drawing the circle, using the following code:

Circle.cs Drawing Using Pen Object

```
drawingArea.DrawEllipse(new Pen(_circleColor), x, y, _circleDiameter, _circleDiameter);
```

From the Object Browser, we can see that we've identified one of the class constructors for the Pen object that uses a System.Drawing.Color as a parameter. In the code snippet above, the **_circleColor** is the color passed to the **Pen** constructor. Feel free to explore other elements of the System.Drawing assembly.

Namespaces

The base class libraries are essential tools for developing software with .NET. Using the Object Browser, we opened the System.Drawing assembly, then expanded the System.Drawing namespace. In addition to the System.Drawing namespace, the System.Drawing assembly also contained a number of other namespaces (System.Drawing.Design, System.Drawing.Text). A namespace is a mechanism within an assembly to group related items. In addition to grouping, the namespace is a complete reference to an element you might want to use in your code. Let's confirm that statement by commenting out the reference to System.Drawing, trying to compile, then correcting any errors by prefixing items in the System.Drawing namespace with the fully qualified namespace name.

Open the MyCircle project and comment out the **using System.Drawing** statement with //, then compile it (select **Build | Build Solution**). Then, modify it as shown below and compile again (if the focus is in the project, you can use the **F6** shortcut key) to verify that you've corrected all of the errors that needed the full namespace reference.

CODE TO TYPE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// using System.Drawing;   // Added to gain access to the Pen object

namespace MyCircle
{
    class Circle
    {
        // ***** Properties *****
        private int _circleDiameter = 0;
        ~~private Color _circleColor = Color.Red;~~
        private System.Drawing.Color _circleColor = System.Drawing.Color.Red;

        // ***** Constructors *****
        // Default constructor
        public Circle()
        {
            // Do nothing
        }

        // Set default circle diameter constructor
        public Circle(int defaultCircleDiameter)
        {
            if (defaultCircleDiameter >= 0)
            {
                _circleDiameter = defaultCircleDiameter;
            } else {
                _circleDiameter = 0;
            }
        }

        // ***** Accessors *****
        // Public accessors for getting and setting member property _circleDiameter
        public int Diameter
        {
            set
            {
                _circleDiameter = value;
            }
            get
            {
                return _circleDiameter;
            }
        }

        // Public accessors for getting/setting member property _circleColor
        ~~public Color Color~~
        public System.Drawing.Color Color
        {
            set
            {
                _circleColor = value;
            }
            get
            {
                return _circleColor;
            }
        }

        // ***** Public methods *****
        // Public method (behavior) to draw a circle
        ~~public void DrawCircle(Graphics drawingArea, int x, int y)~~
        public void DrawCircle(System.Drawing.Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid
            if (drawingArea != null)
            {
                // Call the DrawEllipse method of the drawing area
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the m_intDiameter
                // member property for both the height and width
                ~~drawingArea.DrawEllipse(new Pen(_circleColor), x, y, _circleDiameter, _
circleDiameter);~~
                drawingArea.DrawEllipse(new System.Drawing.Pen(_circleColor), x, y, _ci
rcleDiameter, _circleDiameter);
            }
        }
    }
}
```

Click ▶ to see that it works as before.

Let's discuss the changes.

OBSERVE: Circle.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// using System.Drawing;   // Added to gain access to the Pen object

namespace MyCircle
{
    class Circle
    {
        // ***** Properties *****
        private int _circleDiameter = 0;
        private System.Drawing.Color _circleColor = System.Drawing.Color.Red;
.
.
.
        // Public accessors for getting/setting member property _circleColor
        public System.Drawing.Color Color
        {
            set
            {
                _circleColor = value;
            }
            get
            {
                return _circleColor;
            }
        }

        // ***** Public methods *****
        // Public method (behavior) to draw a circle
        public void DrawCircle(System.Drawing.Graphics drawingArea, int x, int y)
        {
            // Make sure the drawingArea is valid
            if (drawingArea != null) {
                // Call the DrawEllipse method of the drawing area
                // creating a dynamic Pen for a color, using the x
                // and y coordinates to specify starting point at the
                // top-left corner, and using the _circleDiameter
                // member property for both the height and width
                drawingArea.DrawEllipse(new System.Drawing.Pen(_circleColor), x, y, _ci
rcleDiameter, _circleDiameter);
            }
        }
    }
}
```

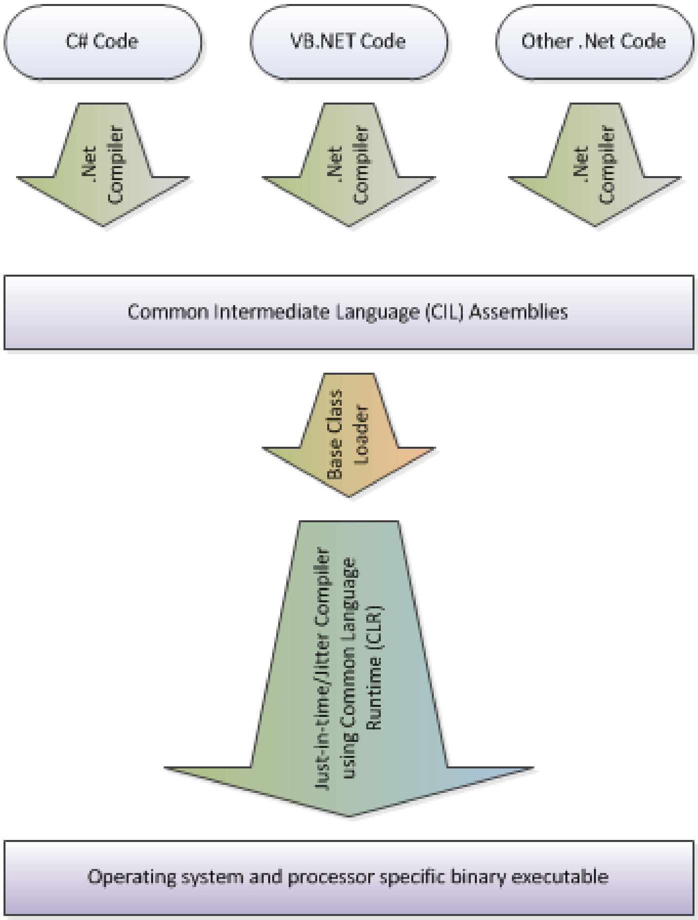
References to Color, Graphics, and Pen all had to be prefixed with **System.Drawing** to enable the compiler to find the referenced objects. Although you can use the fully qualified names, using namespaces within your code allows your code to be much easier to read, and spares you a lot of typing!

You may also note the line **namespace MyCircle** that uses the namespace reserved word. That's right, you are creating your own namespaces when you code! We've been taking the default namespace supplied by Studio, but in future projects we modify the namespaces of our own code.

Compiling, CIL, Assemblies, and Jitter

So far we've seen that we can write code in C# using Visual Studio. We've also built, or compiled, our C# code. When we compile our code (as we did above with **Build | Build Solution**, and as the Debugger does for us when we click ▶), what we're really doing is creating assemblies, just like the assemblies included with the .NET framework known as the base class libraries. The assemblies contain platform-neutral code known as *Common Intermediate Language (CIL)*.

We also learned about CLR. Compiled code in the CIL format still will not execute on the computer, so when the .NET application is executed, the CLR is used to compile the CIL into a format that can be executed on the current operating system. This final CLR-to-CIL compilation is known as *just-in-time* compilation, or *Jitter*, as the compiling can be delayed until the code is needed. This Jitter process also will load any base class libraries used by your code. Do we need an image? Sure!


C# Code
VB.NET Code
Other .Net Code
.Net Compiler
.Net Compiler
.Net Compiler
Common Intermediate Language (CIL) Assemblies
Base Class Loader
Just-in-time/Jitter Compiler using Common Language Runtime (CLR)
Operating system and processor specific binary executable

Advantages of .NET

To complete this lesson, let's summarize some of the advantages of .NET we've discussed:

- Language independence: When using a .NET CLS-compliant language, whether you write in C#, or another language, doesn't matter.
- Cross-platform: .NET CLS-compliant CIL assemblies will run on any platform that has a .NET CLS-compliant CLR and Jitter compiler.

We'll cover more advantages of .NET as we proceed through the lessons, but feel free to find other advantages online!

Next

That's it for this lesson! Now that you have completed this lesson, complete any associated projects, then quizzes.

Object-Oriented Concepts

Introduction

We've already worked at creating a C# class with properties and methods, creating an instance of a class, and touched briefly on a few object-oriented terms. In this lesson, we'll go much further as we explore objects. We'll cover object terminology, begin working with the Visual Class Designer included in Studio, and introduce the Unified Modeling Language (UML) as it relates to the Visual Class Designer. Let's get started!

Note *Object-Oriented Programming* is often abbreviated as *OOP*; you'll see it used throughout the lessons. In fact, you'll see the base *OO* acronym in many cases.

Objects In Our World

Object State and Behavior

An observation of objects in the real world reveals that they have two primary characteristics: *state* and *behavior*. The *state* describes a *characteristic of the object*. For example, hair is an object, and possible states of hair may include color, thickness, health (does it have split ends?), length, etc. *Behavior* is something the object can *do*, or what it might do in reaction to some internal or external force or event. Again, for hair, behaviors may include moving (maybe the wind is blowing), drying, falling out, turning grey, etc.

In our discussion of objects, let's find a real-world object that many of us have used, or at least seen: that rather bright red box that dispenses movies for $1 near popular fast-food chains. To avoid any copyright, privacy, and health reasons, we will not mention any fast-food chains by name, and we'll refer to these bright red machines as **Red Movie Machines**. In fact, we'll even supply our own image of one below.

Think about the possible states of our Red Movie Machine. Ask yourself, "What are some characteristics the Red Movie Machine can have?"

So what *are* some of the states of the Red Movie Machine? Consider the purpose of the object itself: to hold individual movies, allow users to search available movies, and rent one of the movies if the machine has that movie in stock. Knowing the purpose of an object helps to determine its possible states. For the Red Movie Machine, states may be status (idle, in use, broken, no electricity), or location (where is the machine physically located). What about empty or full? Yes, those could be properties, and if the entire machine was empty, then it would make sense, but its emptiness or fullness is otherwise only relevant to specific movies. So, our Red Movie Machine contains other objects: movies! The empty/full state would belong to each movie title. We will discuss objects within objects later.

Think about the possible behaviors of our Red Movie Machine. Ask yourself, "What can the Red Movie Machine do?"

What are a few behaviors of the Red Movie Machine? Considering the purpose again, the machine would have the ability to search for a movie, take money, return change, bill a credit card, dispense a movie, and probably a lot more!

Software OOP Terminology

When writing object-oriented software, these same real world concepts apply, although we use different names to refer to them. The state of an object is known as a *property*, *attribute*, or *field*. Although these three terms are generally interchangeable for OOP purposes, C# distinguishes between a field and a property, so we will use both terms as well. A *field* is *a public or protected class variable*, whereas a *property* is *a class method used to access a field*.

The behavior of an object is handled through *methods* or *functions*. These terms are not quite interchangeable. We will use the term *method*. Methods that allow us to access or change an object's properties are known as *accessors*. Methods that allow an object to initiate an action are just normal object *methods*. Methods that allow an object to react to an internal or external event are known as *event handlers*.

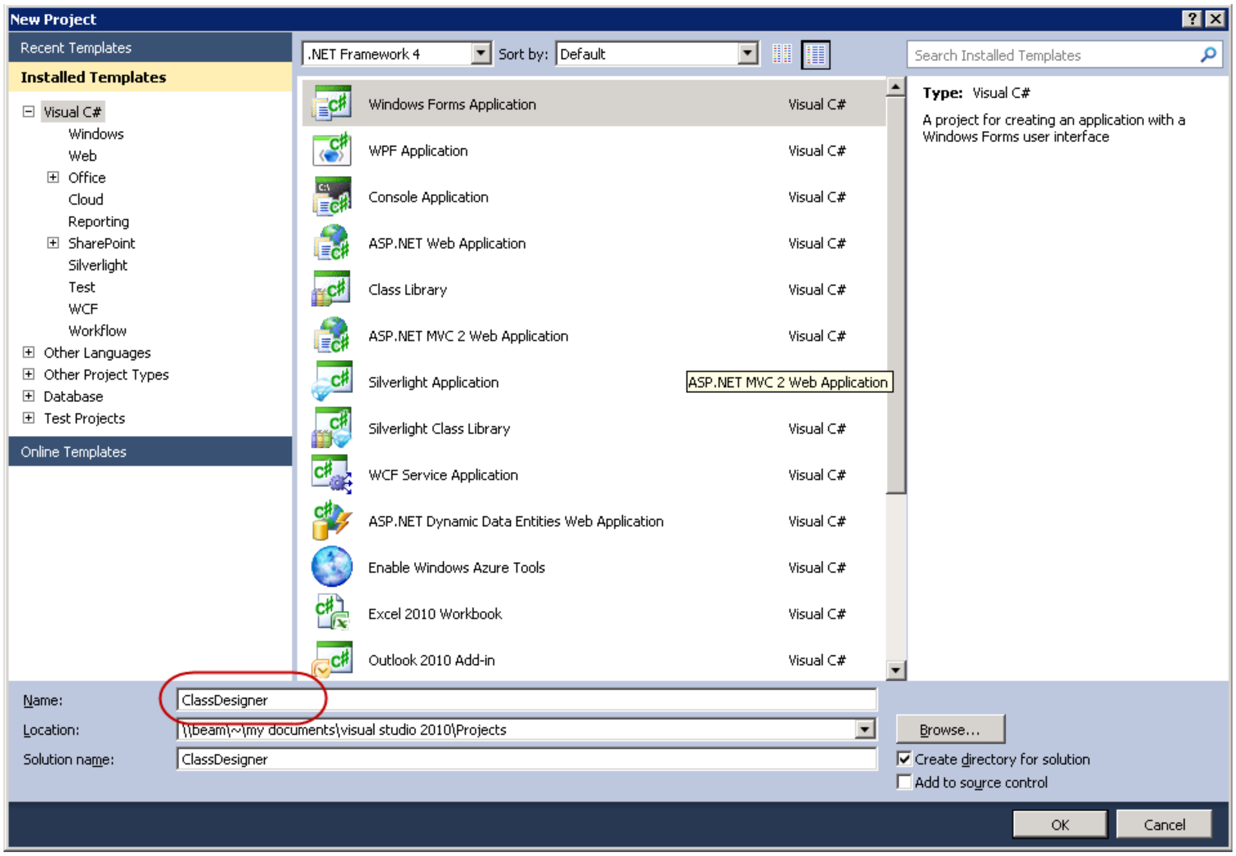
So, how do we separate behaviors into fields, properties, methods, or events? In our earlier lesson about C# classes, and the lesson about methods, you've seen examples of creating private class variables, and get/set accessors for accessing them. We'll discuss these in more detail soon.

Studio Visual Class Designer

The Studio Visual Class Designer is a useful tool for visualizing components of your software. Using the Designer, you can create your classes, establish class properties, create method names and signatures, and visualize the objects and object-relationships within your software. The Designer is not just a visualization tool. When we say "create," we mean create: as you design with the Designer, the underlying C# code for the classes is also created. The relationship between your C# code and Designer is bidirectional: if you change your C# code, those changes are also reflected in the visual class representation within the Designer.

Let's create a new Studio C# project and begin working with the Visual Class Designer.

Create a new Visual Studio Windows Forms Application by selecting **File | New | Project**. In the New Project dialog box, make sure **Windows** is selected under the C# tree item, **Windows Forms Application** is selected in the center section, and all other dropdowns and checkboxes have the default settings. Change the Name to **ClassDesigner**, and click **OK**:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Class Library Visual C#
ASP.NET MVC 2 Web Application Visual C#
Silverlight Application
ASP.NET MVC 2 Web Application
Silverlight Class Library Visual C#
WCF Service Application Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
Enable Windows Azure Tools Visual C#
Excel 2010 Workbook Visual C#
Outlook 2010 Add-in Visual C#
Search Installed Templates
Browse...
OK
Cancel

.NET Framework 4 Sort by: Default

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: ClassDesigner

Location: \\beam\~\my documents\visual studio 2010\Projects

Solution name: ClassDesigner

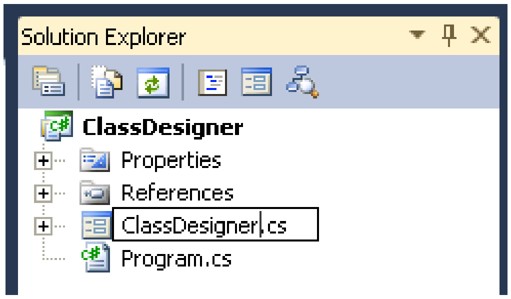
☑ Create directory for solution

☐ Add to source control

Your new Windows Form Application appears.

Note When you create the new project, any previously open project is automatically closed, so you don't need to explicitly close the MyCircle project.

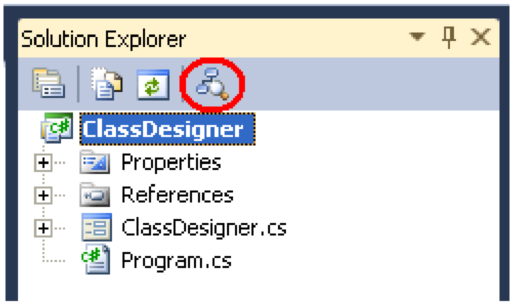
Rename the form: locate the entry for Form1.cs and click on it in the Solution Explorer. Click on the Form1.cs entry a second time, and replace the highlighted Form1.cs text with **ClassDesigner.cs**:



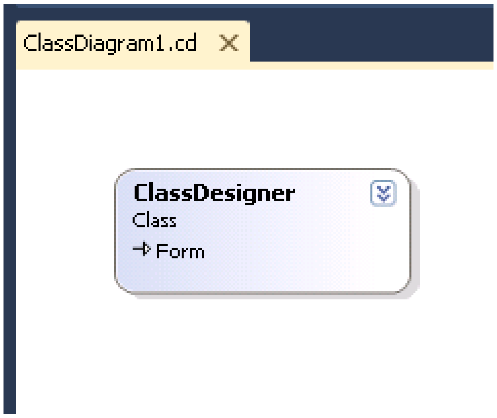
Click 💾 to save the project.

To work with the Visual Class Designer, we need to add a class diagram to our project.

Click the **View Class Diagram** icon on the top right side of the Solution Explorer (when the new tab appears, remember you might need to right-click it and select **Move to Next Tab Group** so you can see this lesson text):



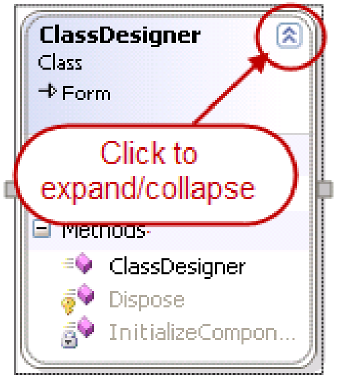
Click 💾 to save the project and new class diagram file.



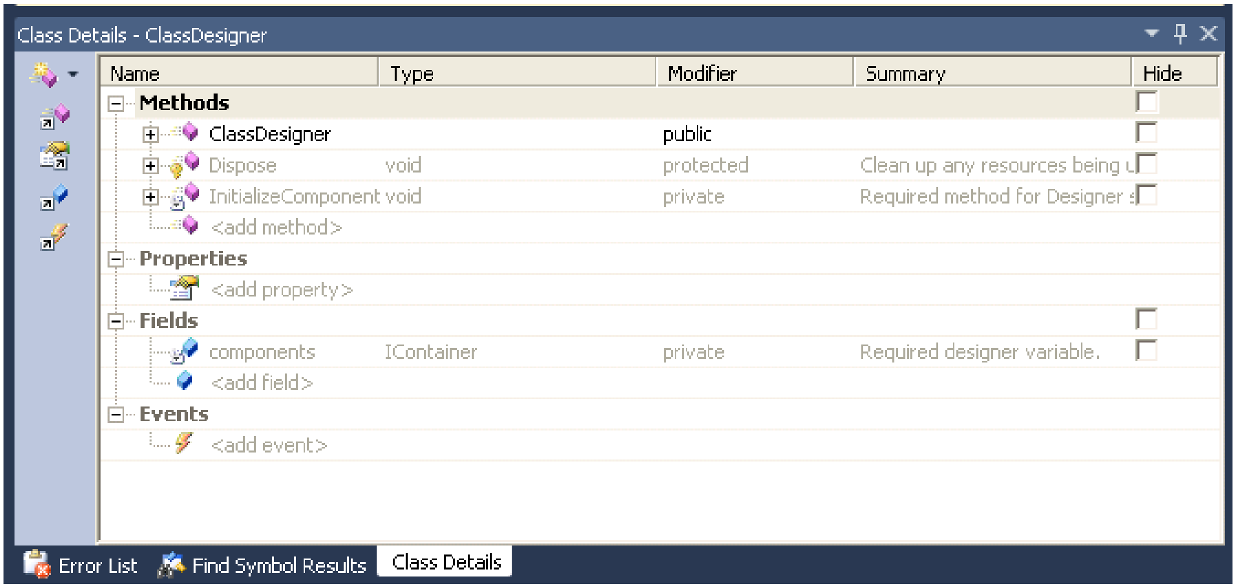
You'll notice that the class diagram for the project already contains an item, known as a class icon. The class icon containing **ClassDesigner** is the main class we work with when we create a Windows Form application; it is the class that was named Form1.cs by default, which we changed to ClassDesigner.cs.

You'll notice that the class icon includes an arrow icon at the top right. Clicking this icon will display (or hide) the elements that make up this class (properties, methods, etc.).

Click on the arrow "expand" icon of the ClassDesigner to display the elements of this class:



The expanded view of the class in the Designer contains information about the class: its parent class, fields (properties), methods, and more. As you click on a class in the Visual Class Designer, notice that the Class Details window, below the Designer window, also displays the details of the selected class, allowing you to modify any of those details, including adding, and deleting any of the class elements.



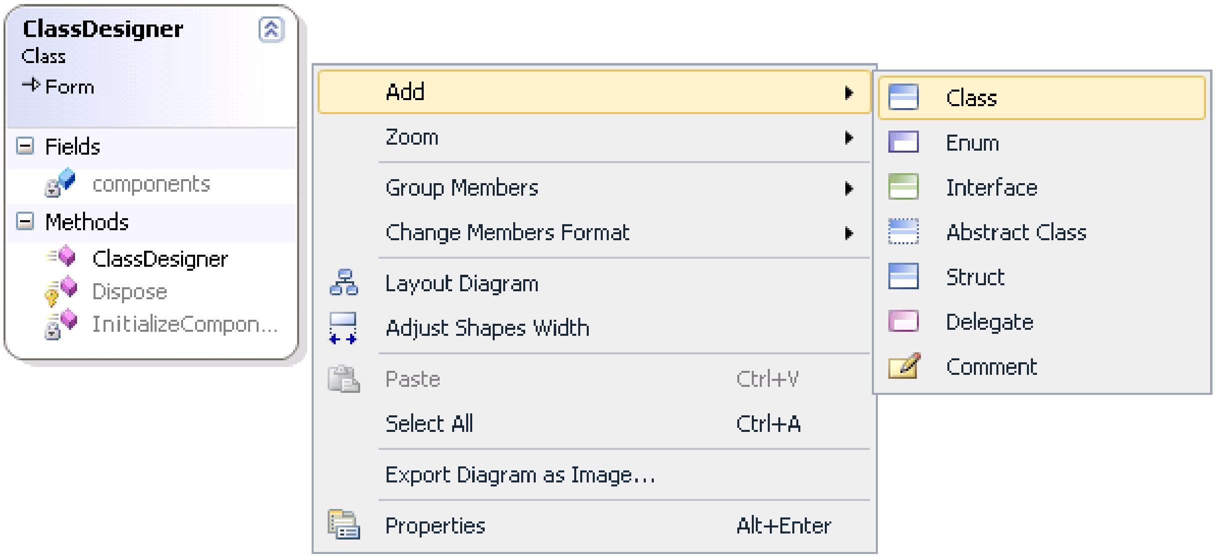
Class Details - ClassDesigner

Name	Type	Modifier	Summary	Hide
Methods				□
ClassDesigner		public		□
Dispose	void	protected	Clean up any resources being u	□
InitializeComponent	void	private	Required method for Designer s	□
<add method>				
Properties				
<add property>				
Fields				□
components	IContainer	private	Required designer variable.	□
<add field>				
Events				
<add event>				

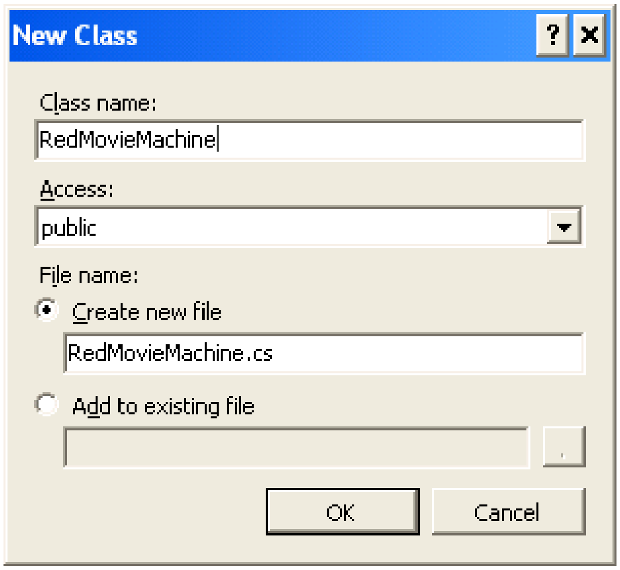
Note If the Class Details window is not visible, you can make it visible by selecting **View | Other Windows | Class Details**.

Let's add a new class that represents the Red Movie Machine we discussed earlier, adding in a few of the properties and methods.

Click the double arrow icon in the ClassDesigner class to collapse the display of its elements so we can focus on our new class. Right-click in an empty area of the Visual Class Designer and select **Add | Class**:



Change the Class Name to **RedMovieMachine**. All other information should remain the same, although the File Name will automatically change to match the class name.



Class name: RedMovieMachine

Access: public

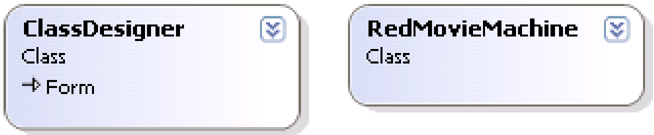
File name:

● Create new file

RedMovieMachine.cs

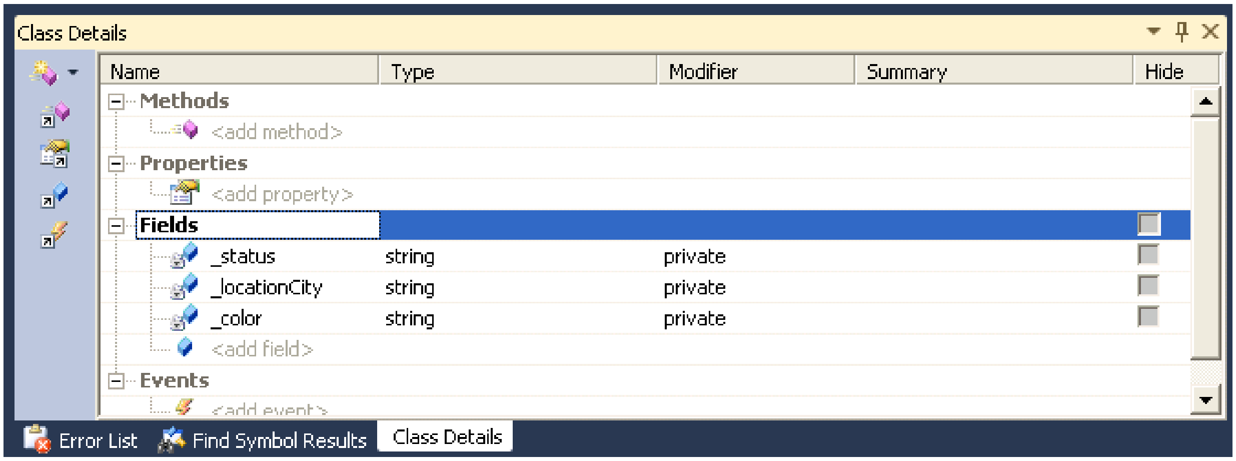
○ Add to existing file

If necessary, drag the new RedMovieMachine class icon in the Designer so it doesn't overlap any other class icons. Click 💾.



Next, let's add three properties to the RedMovieMachine class. If you look at the Class Details, you'll see that there is a section for Fields, and another for Properties. *Fields* are class variables, whereas *Properties* are the accessors, using the get and set C# syntax, for accessing class variables. Let's add three fields to our class: status, city location, and color.

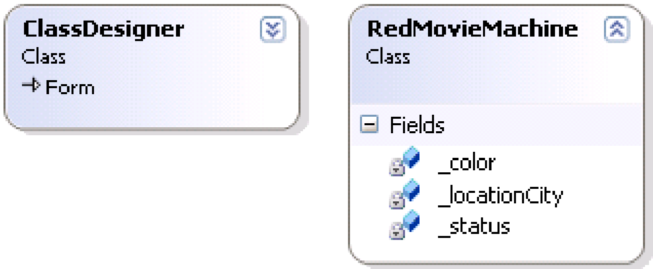
Click on the RedMovieMachine class icon in the Class Designer, then click on the row **<add field>** row in the **Fields** section of the Class Details window. Edit this row, changing the value in the Name column to **_status**, and the Type column to **string**. Repeat this process for **_locationCity** and **_color**, making both Types **string**. It should look like this:



Class Details

Name	Type	Modifier	Summary	Hide
Methods				
<add method>				
Properties				
<add property>				
Fields				□
_status	string	private		□
_locationCity	string	private		□
_color	string	private		□
<add field>				
Events				
<add event>				

In the Class Designer, the RedMovieMachine class icon now includes a Fields section, and all of the new fields are included. Our C# code for the RedMovieMachine.cs class has also been updated:



Tip

Remember, the Save button only saves the current file you're working on that has the focus in Studio. If you've changed the file, you'll see an asterisk (*) in the tab title bar after the filename. The Save All button saves all files of your project. Because the Class Designer is contained within a class diagram file (you can see the default name of the class diagram for this project, ClassDiagram1.cd, in the Solution Explorer) that changes as you modify the design, and the underlying C# class file (RedMovieMachine.cs) is also changed, use the Save All button to make sure that your changes are saved in both files.

Double-click the **RedMovieMachine.cs** entry in the Solution Explorer to bring up the Code Editor for the class.

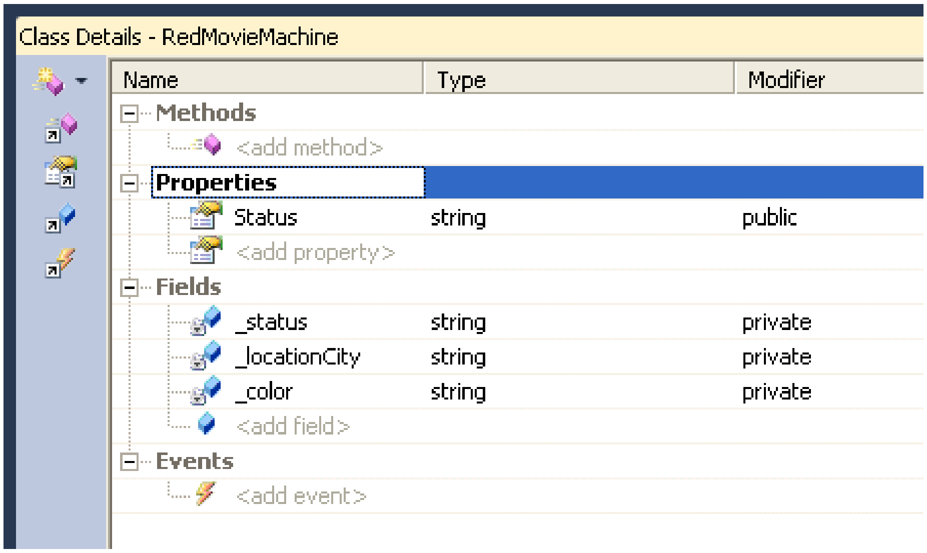
OBSERVE: RedMovieMachine.cs

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

namespace ClassDesigner
{
    public class RedMovieMachine
    {
        private string _status;
        private string _locationCity;
        private string _color;
    }
}
```

You can see the **private member variables**, or Fields, that we added, and see that they look identical to how we've coded private class member variables before. Now, we'll add accessors for these fields, making them Properties.

Click the **RedMovieMachine** class icon in the Class Designer, then click the **<add property>** row in the **Properties** section of the Class Details window. Change the value in the Name column to **Status**, and the

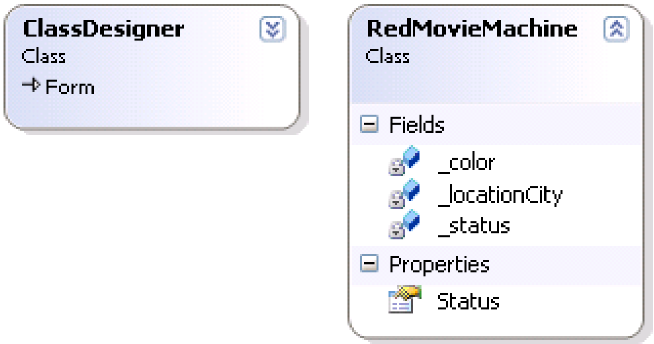
Type to **string**:



Class Details - RedMovieMachine

Name	Type	Modifier
Methods		
<add method>		
Properties		
Status	string	public
<add property>		
Fields		
_status	string	private
_locationCity	string	private
_color	string	private
<add field>		
Events		
<add event>		

In the Class Designer, the RedMovieMachine class icon now includes a Properties section, and the new property is included. Our C# code for the class, RedMovieMachine.cs, has also been updated again:



ClassDesigner
Class
→ Form

RedMovieMachine
Class

Fields
- _color
- _locationCity
- _status

Properties
- Status

Double-click the **RedMovieMachine.cs** entry in the Solution Explorer to bring up the Code Editor for the class:

OBSERVE: RedMovieMachine.cs

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

namespace ClassDesigner
{
    public class RedMovieMachine
    {
        private string _status;
        private string _locationCity;
        private string _color;

        public string Status
        {
            get
            {
                throw new System.NotImplementedException();
            }
            set
            {
            }
        }
    }
}
```

The new **accessors** definitely look like the accessors we've coded before, with two exceptions: neither the **get** nor the **set** is related to the private class variable **_status**, and the **get** includes a **throw** statement. The **get** accessor is a method that must return a value, so the Visual Class Designer added a statement that will raise an exception that this accessor has not yet been implemented. We can manually add the code that links the *property* **Status** to the *field* (or private class variable) **_status**.

Modify the **RedMovieMachine.cs** code as shown below to link the _status field to the Status property:

CODE TO TYPE: RedMovieMachine.cs

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

namespace ClassDesigner
{
public class RedMovieMachine
{
    private string _status;
    private string _locationCity;
    private string _color;

    public string Status
    {
        get
        {
            ~~throw new System.NotImplementedException();~~
            return _status;
        }
        set
        {
            _status = value;
        }
    }
}
```

Save your changes. Let's look at the code we added.

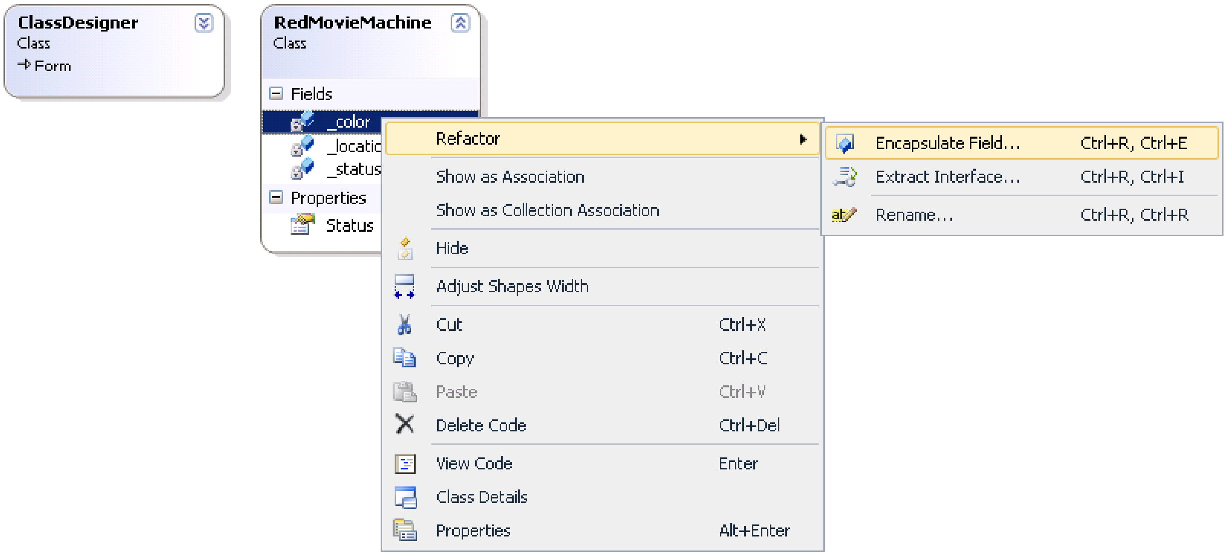
OBSERVE: RedMovieMachine.cs

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

namespace ClassDesigner
{
    public class RedMovieMachine
    {
    private string _status;
    private string _locationCity;
    private string _color;

    public string Status
    {
        get
        {
            return _status;
        }
        set
        {
            _status = value;
        }
    }
}
```

From our previous lessons, the code we've added should be clear: we added a **return** statement within the **get** block of the **Status** property (accessor), and added an assignment statement to the **set** block of the **Status** property (accessor). The **return** statement returns the value of the **_status** field. The assignment assigns the implicit accessor parameter **value** to the **_status** field (private class variable).

But, wait! Shouldn't Visual Class Designer have an easier way to link a Property to a Field? It does! Let's use the Designer to link the remaining two fields to properties.

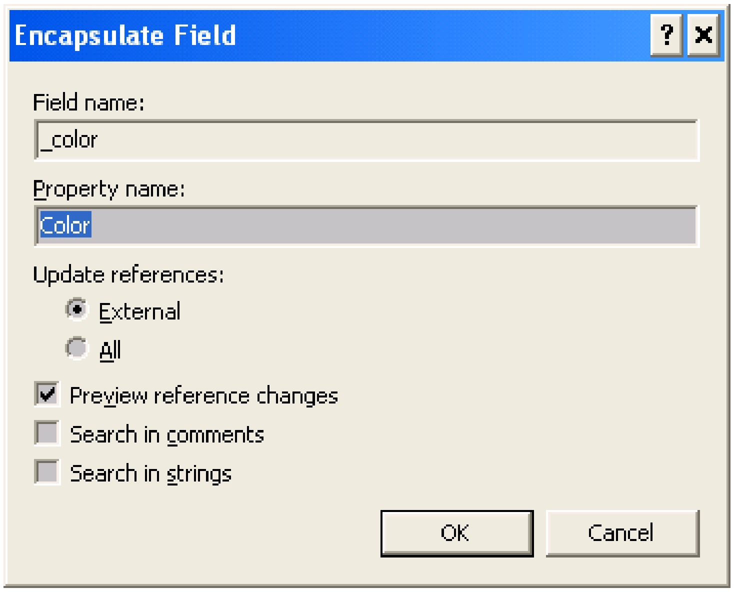
Click the **RedMovieMachine** class icon in the Class Designer, then right-click the **_color** Field property. Select **Refactor | Encapsulate Field**:



Note *Refactoring* is the process of modifying aspects of a program that in some way improves the source code, without modifying the functionality of the program.

In the Encapsulate Field dialog box, make the link between the Field and the Property. Because our variable names use the underscore, the dialog box provides an acceptable default name for the Property.

Ensure that the **Property Name** of the **Encapsulate Field** dialog box is **Color**. Leave all other settings at their default values, and click **OK**.



Field name: _color

Property name: Color

Update references:

● External

○ All

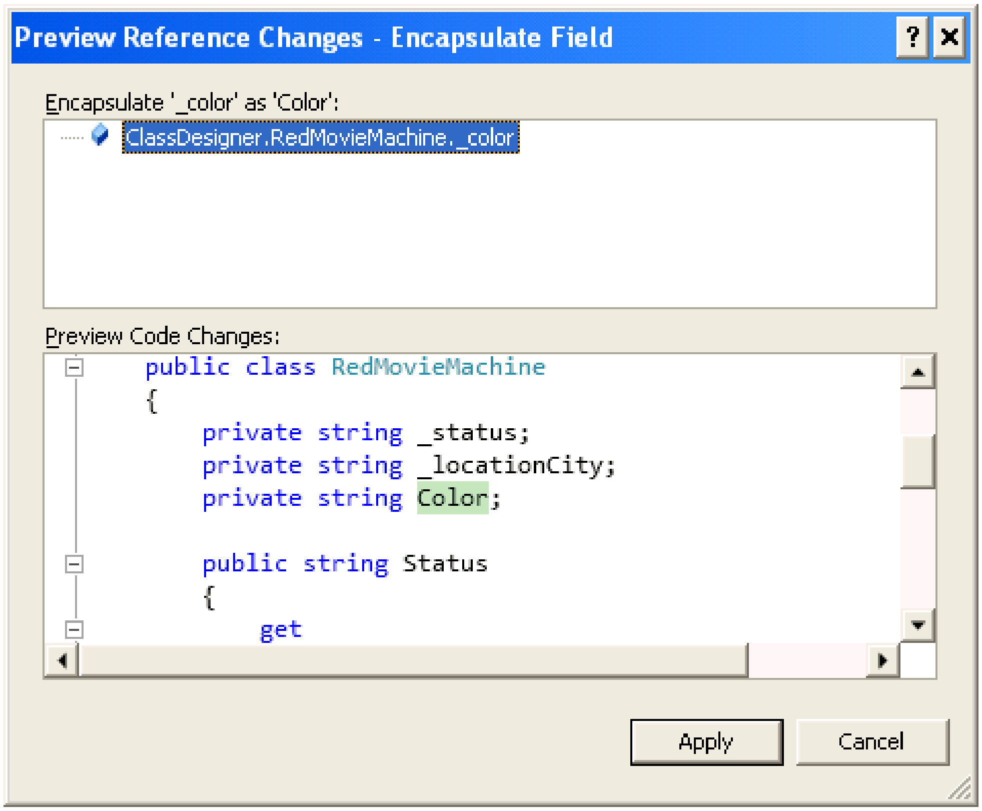
☑ Preview reference changes

□ Search in comments

□ Search in strings

Next, we see a dialog box showing the proposed changes. The code presented doesn't seem correct, as it seems to be changing our _color field to Color, but we'll accept the change for now.

Click **Apply** to accept the proposed changes.



Encapsulate '_color' as 'Color':

ClassDesigner.RedMovieMachine._color

Preview Code Changes:

```
public class RedMovieMachine
{
    private string _status;
    private string _locationCity;
    private string Color;

    public string Status
    {
        get
```

The Class Diagram now includes the Color Property. What about the code?

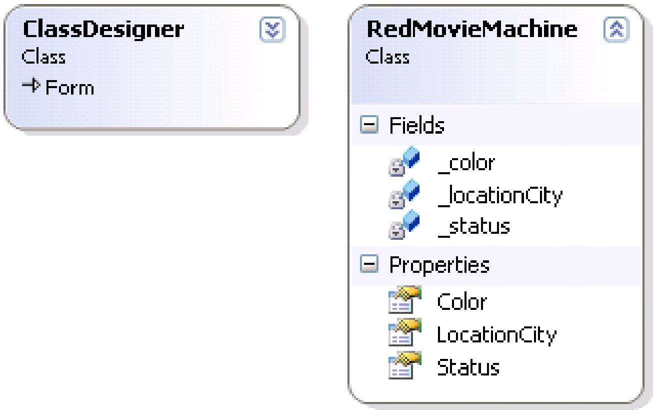
Click the **RedMovieMachine.cs** tab to see the Code Editor for the class. Examine the class contents to see what changed:

OBSERVE: RedMovieMachine.cs

```
.
.
.
        public string Color
        {
            get { return _color; }
            set { _color = value; }
        }
.
.
.
```

It worked! The code generated by the Visual Class Designer is in a more compact form, yet still completely readable! Let's complete the task by linking the **_locationCity** field to a Property.

Click the **RedMovieMachine** class icon in the Class Designer, then right-click the **_locationCity** Field property. Select **Refactor | Encapsulate Field**. In the Encapsulate Field dialog box, ensure that the Property Name is **LocationCity**. Leave all other settings at their default values and click **OK**. Then, click **Apply** to accept the proposed changes. The final class diagram should look like this:



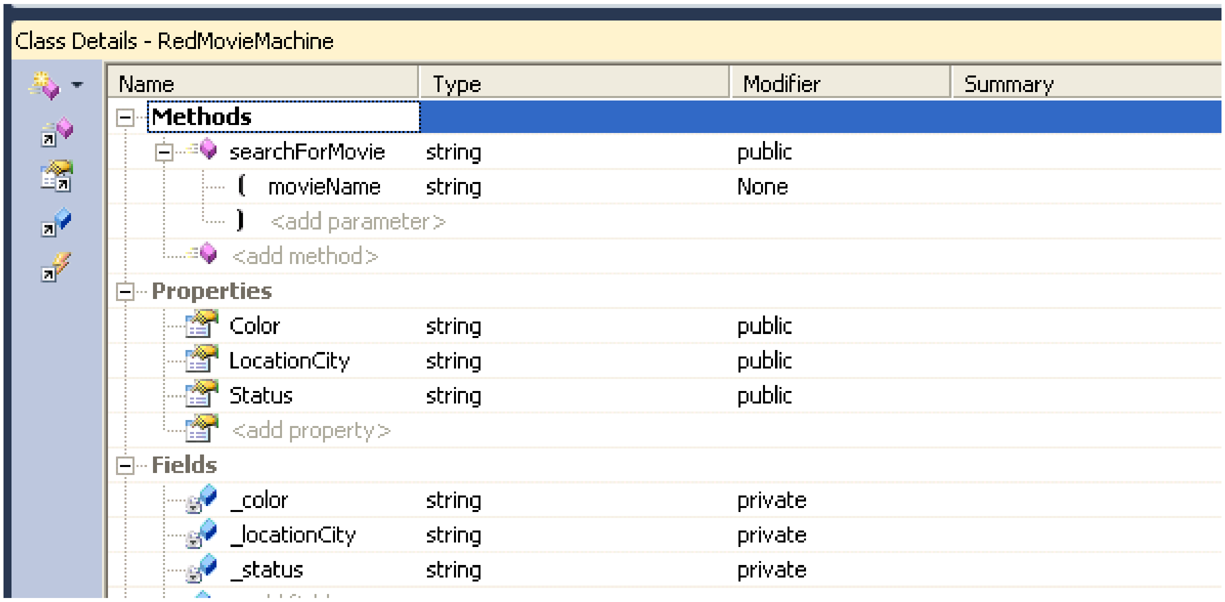
The final source code should look like the code shown below. You are welcome to reformat the **Status** accessor to match the more compact format created by the Visual Class Designer, as we did.

OBSERVE: RedMovieMachine.cs

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

namespace ClassDesigner
{
    public class RedMovieMachine
    {
        private string _status;
        private string _locationCity;
        private string _color;

        public string LocationCity
        {
            get { return _locationCity; }
            set { _locationCity = value; }
        }

        public string Color
        {
            get { return _color; }
            set { _color = value; }
        }

        public string Status
        {
            get { return _status; }
            set { _status = value; }
        }
    }
}
```

To finish up with Visual Class Designer, we need to add a method to our class. For now, we'll add a method that allows us to search for a movie. We'll also add comments about our method.

Click the **RedMovieMachine** class icon in the Class Designer, then click on the **<add method>** row in the **Methods** section of the Class Details window, changing the Name value to **searchForMovie**, and the Type to **string**. Next, click the plus symbol next to the searchForMovie entry, and click the **<add parameter>** row in the searchForMovie method, changing the in the Name value to **movieName**, and the Type to **string**:

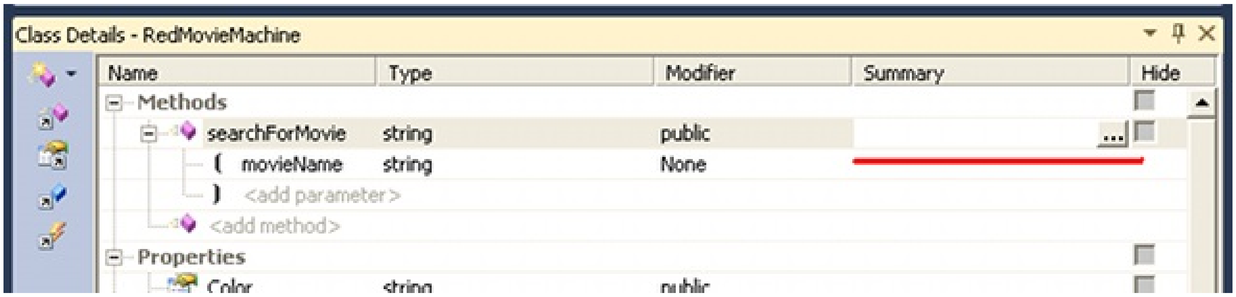


Class Details - RedMovieMachine

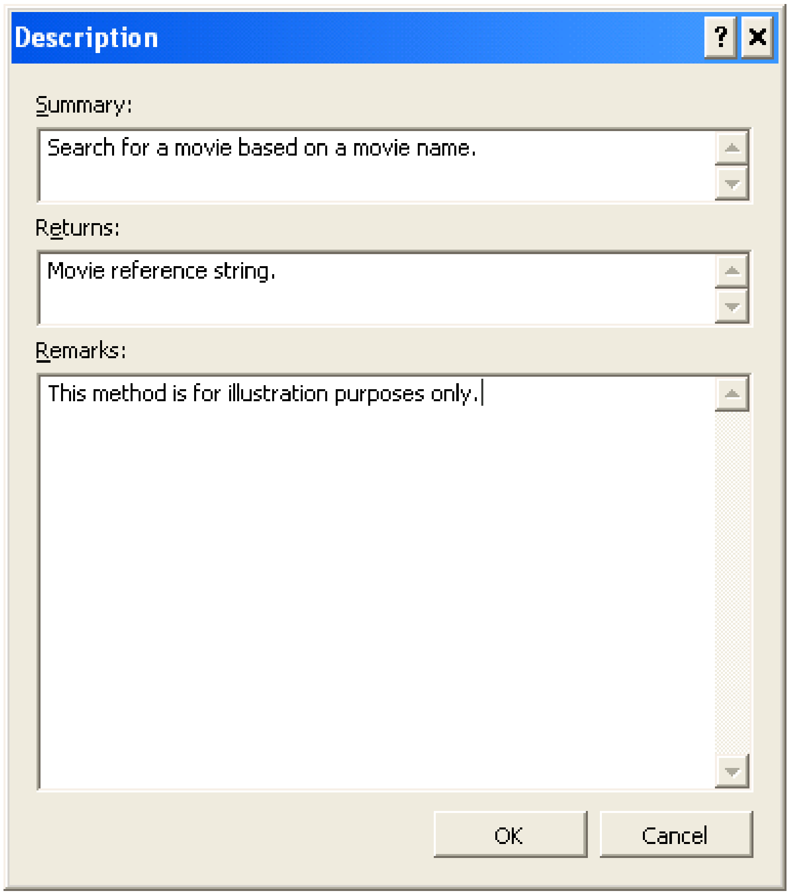
Name	Type	Modifier	Summary
Methods			
searchForMovie	string	public	
(movieName	string	None	
) <add parameter>			
<add method>			
Properties			
Color	string	public	
LocationCity	string	public	
Status	string	public	
<add property>			
Fields			
_color	string	private	
_locationCity	string	private	
_status	string	private	

Before we look at the code, let's also add some comments to our searchForMovie method. We can simply enter a short summary, or, like we're going to do, enter full comments for the method.

In the Class Details window, click in the Summary column of the searchForMovie method. Click the ellipsis (...) that appears in the Summary row:



In the Description dialog, enter the text as shown below, and click **OK**.



Summary:

Search for a movie based on a movie name.

Returns:

Movie reference string.

Remarks:

This method is for illustration purposes only.

Let's look at the code and the comments we added.

Click the **RedMovieMachine.cs tab** to change to the Code Editor for the class. Examine the class contents to see what changed.

OBSERVE: RedMovieMachine.cs

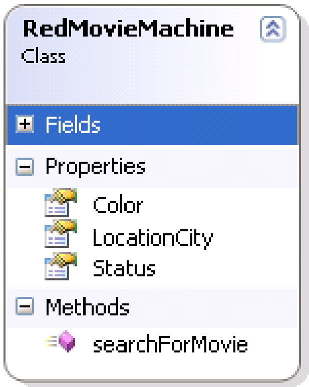
```
.
.
        /// <summary>
        /// Search for a movie based on a movie name.
        /// </summary>
        /// <remarks>This method is for illustration purposes only.</remarks>
        /// <returns>Movie reference string.</returns>
        public string searchForMovie(string movieName)
        {
            throw new System.NotImplementedException();
        }
.
.
.
```

Once again, the Designer worked! We now have a **new method outline**, and we have **comments** included from what we typed in, although these comments have three slashes rather than two, as well as embedded XML tags. Studio supports XML comments that makes it easy to extract documentation from your code. Is that important? Just imagine if you had thousands of lines of code, dozens of classes, each with dozens of properties and methods. XML comments make it easy to document you code and to keep your documentation current!

Unified Modeling Language and Class Diagrams

In our discussions of an object's state and behavior, in conjunction with the Visual Class Designer, we have already been using a standardized system of notation known as *Unified Modeling Language (UML)*. Although UML defines nine types of diagrams, for now we'll just learn the basics of the *Class Diagram* type. UML diagrams are further divided into two basic categories: *structure diagrams* and *behavior diagrams*. We will focus on *structure diagrams* that show the architecture of the system we're modeling. Structure diagrams also include the basic relationships between the objects.

UML Class Diagrams represent an abstraction of entities with common characteristics, and are illustrated using rectangles divided into three components. The top compartment of the Class entity contains the name of the class, in bold, using Pascal Case. The middle compartment contains the states, or class attributes. The bottom compartment contains the class behaviors. From our use of the Visual Class Designer, the class icon in the Designer is the Class entity. Rather than a single compartment for attributes, we have two: one for Fields, and another for Methods. Because the Fields should be kept *private* to limit access to these variables except through the accessor methods, we can collapse the Fields entry. Below is a Class entity of a UML Class Diagram for our Red Movie Machine class.



For now, we'll end our discussion of UML, although we will add more UML concepts as we work through more lessons, and we'll continue to use the Visual Class Designer to model and create our systems. We'll finish this lesson with a discussion of fundamental OOP topics. We'll revisit these topics throughout other lessons as we have an opportunity to present the terms in the context of creating C#-based applications using Visual Studio.

Self-Containment and Encapsulation

The Red Movie Machine is a financial business, which means that the operators of the machines are not fond of vandalism. In fact, you could easily state that the operators mandate that any interaction with their Red Movie Machines be through approved pathways! What does that mean? If you want to rent a movie, you must follow the directions and use the machine only in ways that the creators of the machine allow!

This concept of only allowing access to an object through approved mechanisms is strictly followed in the OO software world. Enforcement includes access to an object's properties. If you want to rent a movie from the Red Movie Machine, you are only allowed to do so using mechanisms that it facilitates and responds to. The machine itself will then update its properties, such as how many movies are left. You can state, then, that the Red Movie Machine as an object is self-contained, and only allows access to itself via *public interfaces*. Another way of describing this self-contained concept is that the state and behavior of the object is *encapsulated* within the object itself. *Encapsulation* is a fundamental OOP concept we will revisit in the next lesson as we learn to create C# classes.

Note Just as in the real world, if you circumvent legitimate and approved means of accessing an object—for example, using a bat to break into the Red Movie Machine—the object may give unexpected results, or stop functioning completely!

Versatility and Polymorphism

As you consider the Red Movie Machine, and other vending machines, most of them have a single slot to insert bill, regardless of whether the denomination is $1, $5, $10, or $20. Very convenient! Imagine if you had to have a different slot for each denomination! Imagine an even worse situation, where the machine handles change, and requires you to insert each coin in a slot designed solely for that coin! But, the slots can handle all of the change or paper currency, making the machine very versatile. In OOP software terminology, the ability to use the same public interface for different types, such as the money slots, is known as *Polymorphism*.

Reuse and Inheritance

In the Red Movie Machine, a fundamental concept is the ability to reuse movies. A user pays to use the movie for a specific amount of time, then returns the movie for the next patron to use. Reusing objects is another fundamental OOP concept. In fact, software developers strive to eliminate duplication of code that performs the same task! The movie rental and return is a great example of *Reuse*. As you work through the lessons, we will point out opportunities to consolidate and reuse code.

Another topic of interest is *Inheritance*, or the ability to create one object from another base object. Perhaps the most fundamental example of inheritance is the nature of human life, where two parents contribute basic cells to create a new life. This principle embodies the idea of using fundamental objects to build other objects, and is frequently used in OOP. For the Red Movie Machine, the handling of the different currencies within the machine might employ a base currency handler, then specific instances for each specific currency. Don't worry, we'll cover this topic again.

Type Versus Containment

The Red Movie Machine takes money and dispenses a product (the movie). In other words, the Red Movie Machine *is* a vending machine, or, to use a software term, a Red Movie Machine is of *type* vending machine, a different object. The Red Movie Machine is NOT a DVD movie; however, it does *use*, or *contain*, DVD movies. Differentiating between what an object *is* versus what an object uses or contains, helps when creating software representations of objects.

Next

Another lesson completed! Now that you have completed this lesson, complete any associated projects, then quizzes.

Simple Design Concepts

Introduction

Creating software is more than writing code. Consider the terms used for coders: programmer, software engineer, developer, software architect, and the list goes on. *Developing software is an act of engineering*, and as such requires *planning*. As we continue with the lessons, we will develop the skills needed for software engineering.

So far, we've touched upon coding standards (Pascal case versus Camel case, variable naming conventions, brace placement, etc.), and four basic principles that should be considered when developing a user interface (spacing and positioning, size, grouping, and intuitiveness). In this lesson we'll going to introduce more graphical user interface (GUI) guidelines, reiterate consistency, and introduce the concept of functional requirements.

User Experience

You may recall from when we introduced creating a user interface, we provided a link to an excellent Microsoft web site discussing how to create the best user experience with your application. Let's return to that article, and jump to the section that describes 20 Tips for a Better, Functional User Experience. We won't be covering all twenty of the tips, but you should definitely take the time to read through them. As we go through a few of the tips, we will create an application that emphasizes the content.

Standards

The first tip is the best place to start: *Stick to Standards*. Although this article was written in 2006 and points you to using the Windows XP Design Guidelines, the concept is still true today, even though Windows Vista has come (and gone?), and Windows 7 is the latest operating system. As the article stresses, as you create your software, you should do your best to adhere to the standards for your *software environment*. The use of the term *software environment* is intentional. As a user works with a software application, their productivity increases, and changing that environment may adversely impact the productivity of your user. If you introduce a new application, and the application includes features that follow the standards that the user has already seen, the user will be more productive in a shorter time. If you introduce features that are new, then the user will have to learn your features if they want to use your software. Reducing a user's learning curve is always preferable.

What exactly is meant by standards? The article lists numerous features: which controls you use, the layout of the controls, the shape and style of the user interface elements, how the user interacts with your interface, and even how your application flows, or moves from one activity to another.

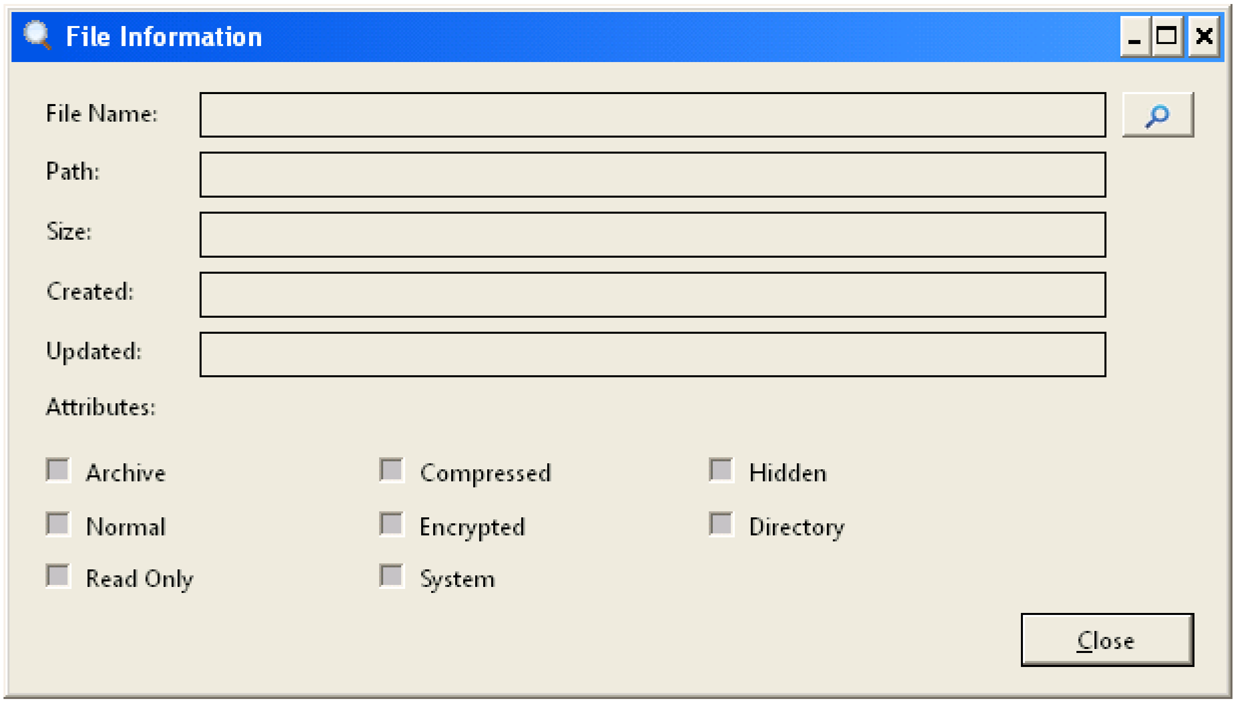
The current standard (when this lesson was created) is embodied in a rather long document (almost 900 pages) from Microsoft known as the User Experience Interaction Guidelines, available in PDF format. This document includes the specific Windows guidelines in the areas of design principles, controls, commands, text, messages, interaction, windows, visual experiences, and the Windows environment. Whew! So, should you find yourself wondering how to design an element, grab this document, jump to the relevant section, and read! In fact, the 20 tips from the 2006 article have been reduced to 19 in the current guide, although the new 19 are definitely more generic guidelines as opposed to the specifics of the 20 tips from 2006. If you read through the 20 from 2006, then the 19 from the current guideline, you'll find the new 19 much easier, and you might even experience that "ah ha!" feeling—the 19 make a lot of sense.

Applying Basic Standards

Let's pick a few of the tips from the 20 from 2006, and design an application using those tips. We'll use tip #1 (**Stick to Standards**), tip #3 (**Simplify Recognition with Icons**), and tip #19 (**Provide Tooltips!**). For tip #1, we will be including the standard Microsoft Windows font style and point size: **Segoe (pronounced "SEE-go") UI 9-point**. For our application, we will be creating a program that allows us to select a file and then display information about that file (such as the size, attributes, etc.).

Create a new Visual Studio Windows Forms Application by selecting File from the menu, then New Project. In the New Project dialog box, change the Name of the project to **FileInfo**; as usual, you can keep the default values for all other settings. Click **OK**. Click on the **Form1.cs** entry in the Solution Explorer, pause and click it again, and replace it with **FileInfo.cs**. Click 💾 to save the project.

For this application, rather than walk you through every control that you will need to place on the Form, we will show you a completed Form and let you create the controls. The form contains Label controls, TextBox controls, Button controls, and CheckBox controls. First, we'll give you a couple of tips that will make adding the controls much easier! This is what the completed application should look like:



Select the form and drag its right edge to make it as wide as the example.

Can you identify all of the controls? The Label controls are the text items down the left side of the form (six total). The CheckBox controls are at the bottom of the form (eight total). The TextBox controls are in the middle of the form (five total). The buttons are at the top right and bottom right (two total).

Tip We mentioned above that we would be using a 9 point Segoe UI font for everything on the form, to adhere to the Windows Vista and Windows 7 standard font name and font size. If you add up all of the controls on the form, there are 21 controls, so setting the font for each of these controls individually might take some time! Rather than dropping all of the controls on your Form, then setting the properties, start with one of each control, set the properties for that control, and then copy and paste that control onto the form for however many additional copies you need of that control, thus duplicating all of the properties (but not the name) in the new pasted control(s). Much easier!

Tip You may want to review the **Format** menu options for aligning and resizing controls. Remember that the first control selected is the model that subsequent selected controls use when resizing or aligning!

For the Label controls, drag a first Label control onto the Form, select the Font property in the Properties Window, and change the Font and Size to **Segoe UI** and **9**. Right-click on the Label control, select **Copy**, then right-click on an empty area of the form and select **Paste**. A copy of the Label control appears on the form. Move the Label control, and repeat the process three more times until you have all five Labels on the form. Select each Label, change the **Text** property to match the target text from the image above. In this case, you can keep their default Name properties. We could name them all, but we will not be doing anything with them, so we can ignore changing their names.

Note You can also use keyboard shortcuts to copy and paste the controls more quickly. With the control selected, press **[Ctrl+C]** to copy, and **[Ctrl+V]** to paste.

Next, let's add the TextBox controls. We will make each TextBox control be read-only so that a user can copy the information, but cannot change it, and we'll change the border style to get the boxed-in look we want.

Drag a TextBox control onto the form. Select the Font property in the Properties Window, and change the Font and Size to **Segoe UI** and **9**. Select the ReadOnly property and change it to **True**. Select the BorderStyle property, and change it to **FixedSingle**. Paste multiple copies of the TextBox control until you have all five, moving each new TextBox to the correct location. For each TextBox, change the Name property to an appropriate control name based on our standards, such as **fileNameTextBox**, **pathTextBox**, etc.

Remember to save your work 💾!

Next, we'll add the CheckBox controls.

Drag a CheckBox control onto the form. Select the Font property in the Properties Window, and change the Font and Size to **Segoe UI** and **9**. Paste copies of the CheckBox control until you have eight of them, moving each new CheckBox to the correct location. For each CheckBox, change the Text property as appropriate (**Archive**, **Normal**, **Read Only**, **Compressed**, **Encrypted**, **System**, **Hidden**, and **Directory**) and change the Name property to an appropriate control name based on our standards, such as **archiveCheckBox**, **normalCheckBox**, etc.

Getting easier? Let's finish with the Button controls.

Drag a Button control onto the form. Select the Font property in the Properties Window, and change the Font and Size to **Segoe UI** and **9**. Copy and paste it to make a second copy. Move each Button control to the correct location on the form. Select the bottom button control and change the Text property to **Close**, and the Name property to **closeButton**.

For the other Button control, we'll add an image to it to make it an icon and clarify its purpose. The button with an image will also stand out, drawing more attention to it, which is what we want for this application.

You'll need to use the image at search_glyph.png. To get it, right-click the link and select "**Save Target As...**". Save the file in your **My Documents** folder (which should be the default).

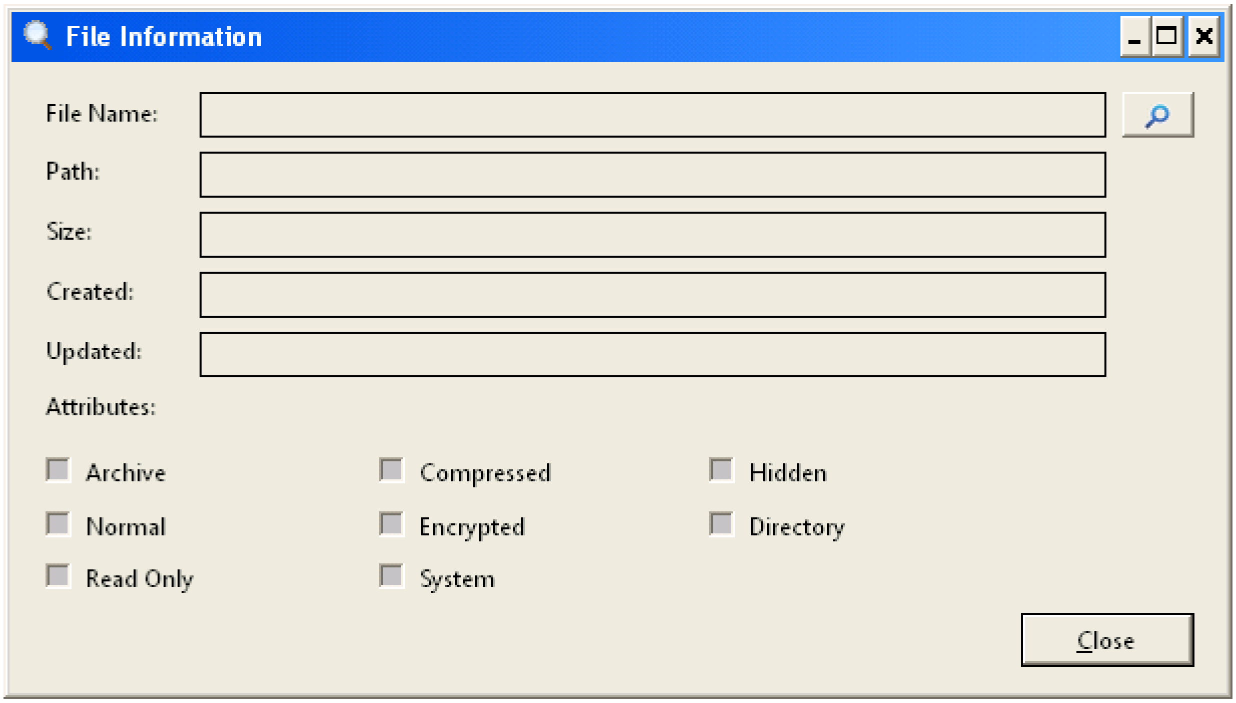
Select the top button control, and change the Name property to **findFileButton**. Select the Text property and delete the text. Select the Image property and click on the ellipsis to see the Select Resource dialog box. Select the **Import...** button, then navigate to **My Documents** (which again should be the default), where you should see the **search_glyph(.png)** file you just saved. Click **Open** and then click **OK**. Resize and move the button to match the example of the completed form above.

Note search_glyph.png file is one of many stock images provided with Visual Studio.

Finally, we need to modify the form itself.

Clicking in any blank space on the form to select it. Change the Text property to **File Information**, and the Font and Size to **Segoe UI** and **9**. Remember to 💾.

Your application form should resemble the version we showed at the beginning:



So, where have we applied standards so far? We've used the recommended Segoe UI 9-point font. We enhanced one of our button controls with an icon to increase awareness of the functionality, encouraging a "use-me-first" attitude since only one button has an image. We also grouped our controls, used equal spacing and alignment, and we also placed the Close button to be easily found at the "end" (the lower right of the form). The choice of making the TextBox controls with a single-line border, and not three-dimensional, also will make the file information more obvious. Could we have done more? Certainly! We might have changed the Form background color, added a help button, and maybe even made the Label control text bold. With such a simple, single-window application, with very limited functionality, not all of the tips applied, but hopefully you are recognizing the point: planning and creating an application is more than just the code, and designing the user interface is a critical component of delivering a great product!

Adding Tooltips and the Form Load

We still need to add Tooltips, and the code to make the program work. "Wait," you say, "What is a Tooltip?!"

You see them in Visual Studio when you hover the mouse pointer over an icon like 💾—a popup appears that gives you information about what you're hovering over. Let's add that feature to our buttons, and of course add the code to make the program work! But, before we add Tooltips, we need to discuss the Form Load event.

By now, you've seen many examples how we have code that responds to certain events, such as when we click on a button control. There are many more events, and we'll cover events more later, but one event we need to add to our application is the Form Load event. This event executes whenever the Form is loaded, which means for our application, when the program is first run, and then will never execute again until we stop and restart the program. So, if we need certain actions to take place only once, and only when the Form loads, the Form Load event is the right place to put them!

Previously, we've put "initialization" code in the public Form constructor method (the method that has the InitializeComponent() call). We've mainly been using this method because we had yet to introduce the Form Load event. Let's add this method to our code, and add and test our Tooltips! We're also going to see how to link the event code you type to the Form Load event using the Event properties page.

Open the Code Editor by selecting **FileInfo.cs** in the Solution Explorer and pressing **F7**, or by right-clicking **FileInfo.cs** in the Solution Explorer and selecting **View Code**. Modify **FileInfo.cs** as shown.

CODE TO TYPE: FileInfo.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 FileInfo
{
    public partial class FileInfo : Form
    {
        public FileInfo()
        {
            InitializeComponent();
        }

        private void FileInfo_Load(object sender, EventArgs e)
        {
            ToolTip toolTip = new ToolTip();
            toolTip.SetToolTip(findFileButton, "Select File");
            toolTip.SetToolTip(closeButton, "Close Application");
        }
    }
}
```

💾 Save your work, then go back to the Design Editor and select the Form by clicking on it. In the Properties Window, click the Event icon ⚡. Scroll through the event list, select the **Load** event, and change the dropdown at the right to **FileInfo_Load**. Click ▶ from the Studio button bar to test your changes. Hover over each Button control, and confirm that the Tooltips appear. Click the **Close** button—it doesn't work yet because we haven't written the code for it. To close the program and return to Studio, click the form's ☒ button.

Did you notice how many events there were for just the Form? We'll add more and more events as the lesson progresses. Let's discuss how this code works.

OBSERVE: Adding Tooltips

```
.
.
.
        private void FileInfo_Load(object sender, EventArgs e)
        {
            ToolTip toolTip = new ToolTip();
            toolTip.SetToolTip(findFileButton, "Select File");
            toolTip.SetToolTip(closeButton, "Close Application");
        }
.
.
.
```

We added the **FileInfo_Load** event handler code, and mapped this event handler using the Event property page of the form. When you clicked on the Load event in the event list, you may have noticed the Description at the bottom of the Properties Window for this event: "Occurs whenever the user loads the form." So, when the form is loaded, our Tooltip code will execute.

Tip If the Property Description window is not visible, right-click anywhere in the Property Window and select the **Description** item.

To add a Tooltip to a control, we first have to create a **Tooltip** object, which we do using the C# reserved word **new**, creating an instance of the **Tooltip** class as a variable named **toolTip**. We next use this instance variable, calling the **SetToolTip** method for each control we want to have a Tooltip (in our case, the two Button controls). The first parameter of the **SetToolTip** method is the control that will get the Tooltip (**findFileButton** first, then **closeButton**). The second parameter is the text of the Tooltip to display (**Select File** and **Close Application**).

Wiring the Buttons and the OpenFileDialog

Next, let's add code to make both of the Button controls work. For the closeButton, we've seen the code to close a form, so we'll add that method call. For the findFileButton, we'll use the OpenFileDialog object to bring up a file browsing dialog box, allowing the user to select a file.

In the Solution Explorer, right-click **FileInfo.cs** and select **View Code**. Modify the file as shown:

CODE TO TYPE: FileInfo.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 FileInfo
{
    public partial class FileInfo : Form
    {
        private OpenFileDialog openFileDialog1 = new OpenFileDialog();

        public FileInfo()
        {
            InitializeComponent();
        }

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

        private void FileInfo_Load(object sender, EventArgs e)
        {
            ToolTip toolTip = new ToolTip();
            toolTip.SetToolTip(findFileButton, "Select File");
            toolTip.SetToolTip(closeButton, "Close Application");
        }

        private void findFileButton_Click(object sender, EventArgs e)
        {
            // Set file dialog defaults.
            openFileDialog1.Multiselect = false;
            openFileDialog1.Title = "Select File";
            openFileDialog1.Filter = "All files (*.*)|*.*";
            openFileDialog1.FilterIndex = 1;
            openFileDialog1.RestoreDirectory = true;
            openFileDialog1.FileName = "";

            // Display file dialog.
            DialogResult dialogResult = openFileDialog1.ShowDialog();

            // Check if user clicked OK button.
            if (dialogResult == DialogResult.OK)
            {
                string fullFileName = openFileDialog1.FileName;
            }
        }
    }
}
```

Let's discuss how this code works:

OBSERVE: FileInfo.cs

```
.
.
.
        private OpenFileDialog openFileDialog1 = new OpenFileDialog();
.
.
.
```

We declared a class member variable **openFileDialog1**, of type OpenFileDialog, and assigned it the results of once again calling the **new** operator. So, that makes **openFileDialog1** an instance variable. We could have looked in the ToolBox, and under Dialogs, we would see this same object: OpenFileDialog. In our code, we created an instance of this object, but we could have dragged the control from the Toolbox. Either method works. We'll practice dragging such controls and components later.

OBSERVE: FileInfo.cs, continued

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

We added the **Close()** statement to the closeButton event handler.

Note

Because we simply typed in the code, the closeButton_Click method is not "wired" to handle the mouse click for this Button control. If you double-click on a control, you will get the default event handler name, for the default event, for this control. So, to make the closeButton work properly, either double-click the closeButton on the Form, or use the Event property ⚡ as we did for the findFileButton, selecting the correct event handler for the correct Button.

OBSERVE: FileInfo.cs, continued

```
.
.
.
        private void findFileButton_Click(object sender, EventArgs e)
        {
            // Set file dialog defaults.
            openFileDialog1.Multiselect = false;
            openFileDialog1.Title = "Select File";
            openFileDialog1.Filter = "All files (*.*)|*.*";
            openFileDialog1.FilterIndex = 1;
            openFileDialog1.RestoreDirectory = true;
            openFileDialog1.FileName = "";

            // Display file dialog.
            DialogResult dialogResult = openFileDialog1.ShowDialog();

            // Check if user clicked OK button.
            if (dialogResult == DialogResult.OK)
            {
                string fullFileName = openFileDialog1.FileName;
            }
        }
.
.
.
```

In the event handler for the findFileButton, we set a number of properties of the **openFileDialog1** instance variable: preventing the user from selecting multiple files, setting the title of the dialog box when it appears, setting the text that appears in the file filter, setting which filter to use by default, preventing any default file name from appearing, and setting a property that will restore the dialog to whatever default directory was set before we used it. We then call the **ShowDialog** method of the **openFileDialog1** variable, that displays the dialog. Focus and control of our application will remain with this dialog until the user closes the dialog. When the dialog is closed, we'll be able to know which button the user clicked, using the **dialogResult** variable. We use the **dialogResult** variable to test if the user clicked the OK button by comparing the result variable to a constant value **DialogResult.OK**. If the user did click the OK button, then we use the **Filename** property to extract the full name of the file the user selected (path and file name). Not too bad!

> **Note**
>
> Remember, when you do a comparison, to use the equality test operator **==** and not the assignment operator **=**, a very common mistake (or typo)! If you use the assignment operator, then no matter which button the user clicked, the code would always think the OK button was clicked. It turns out that in this specific code, Studio won't let you use the assignment operator (incompatible types), but you may not always be that fortunate!

Getting File Information

To complete the code, we need to add code to retrieve the information about the file the user selected. Let's do that next.

In the Solution Explorer, right-click **FileInfo.cs** and select **View Code**. Modify the existing code as shown (our file is getting a bit long, so as elsewhere, we'll abbreviate the listing):

CODE TO TYPE: FileInfo.cs

```
.
.
.
        private void findFileButton_Click(object sender, EventArgs e)
        {
            // Set file dialog defaults.
            openFileDialog.Multiselect = false;
            openFileDialog.Title = "Select File";
            openFileDialog.Filter = "All files (*.*)|*.*";
            openFileDialog.FilterIndex = 1;
            openFileDialog.RestoreDirectory = true;
            openFileDialog.FileName = "";

            // Display file dialog.
            DialogResult dialogResult = openFileDialog.ShowDialog();

            // Check if user clicked OK button.
            if (dialogResult == DialogResult.OK)
            {
                string fullFileName = openFileDialog.FileName;
                System.IO.FileInfo fileInfo = new System.IO.FileInfo(fullFileNam
e);
                fileNameTextBox.Text = fileInfo.Name;
                pathTextBox.Text = fileInfo.DirectoryName;
                sizeTextBox.Text = fileInfo.Length.ToString();
                createdTextBox.Text = fileInfo.CreationTime.ToString();
                updatedTextBox.Text = fileInfo.LastWriteTime.ToString();
                archiveCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO
.FileAttributes.Archive);
                normalCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.
FileAttributes.Normal);
                readOnlyCheckBox.Checked = CheckFileAttribute(fileInfo, System.I
O.FileAttributes.ReadOnly);
                compressedCheckBox.Checked = CheckFileAttribute(fileInfo, System
.IO.FileAttributes.Compressed);
                encryptedCheckBox.Checked = CheckFileAttribute(fileInfo, System.
IO.FileAttributes.Encrypted);
                systemCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.
FileAttributes.System);
                hiddenCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.
FileAttributes.Hidden);
                directoryCheckBox.Checked = CheckFileAttribute(fileInfo, System.
IO.FileAttributes.Directory);
            }
        }

        private bool CheckFileAttribute(System.IO.FileInfo fileInfo, System.IO.F
ileAttributes fileAttribute)
        {
            return ((fileInfo.Attributes & fileAttribute) == fileAttribute);
        }

        private void FileInfo_Load(object sender, EventArgs e)
        {
            ToolTip toolTip = new ToolTip();
            toolTip.SetToolTip(findFileButton, "Select File");
            toolTip.SetToolTip(closeButton, "Close Application");
        }
.
.
.
```

Click ▶ from the Studio button bar to test your changes. Click the **Select File** icon, select a file (for example, under **Computer**, double-click the folder that contains your user name with **\\bean\winusers**, double-click your **My Documents** folder, and then select the **PersonalInfo-SelfPortrait** image file we used in an earlier lesson) and click **OK**. You should see the information about the file you selected. You can select another file and get new information, too. When you finish, click **Close**. Not too bad!

Let's discuss how it works.

OBSERVE: FileInfo.cs

```
.
.
.
        private void findFileButton_Click(object sender, EventArgs e)
        {
            // File dialog settings
            openFileDialog.Multiselect = false;
            openFileDialog.Title = "Select File";
            openFileDialog.Filter = "All files (*.*)|*.*";
            openFileDialog.FilterIndex = 1;
            openFileDialog.RestoreDirectory = true;
            openFileDialog.FileName = "";

            // Display file dialog
            DialogResult dialogResult = openFileDialog.ShowDialog();

            // Check if user clicked OK button
            if (dialogResult == DialogResult.OK)
            {
                string fullFileName = openFileDialog.FileName;
                System.IO.FileInfo fileInfo = new System.IO.FileInfo(fullFileName);
                fileNameTextBox.Text = fileInfo.Name;
                pathTextBox.Text = fileInfo.DirectoryName;
                sizeTextBox.Text = fileInfo.Length.ToString();
                createdTextBox.Text = fileInfo.CreationTime.ToString();
                updatedTextBox.Text = fileInfo.LastWriteTime.ToString();
                archiveCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Archive);
                normalCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Normal);
                readOnlyCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.ReadOnly);
                compressedCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Compressed);
                encryptedCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Encrypted);
                systemCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.System);
                hiddenCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Hidden);
                directoryCheckBox.Checked = CheckFileAttribute(fileInfo, System.IO.FileAttributes.Directory);
            }
        }

        private bool CheckFileAttribute(System.IO.FileInfo fileInfo, System.IO.FileAttributes fileAttribute)
        {
            return ((fileInfo.Attributes & fileAttribute) == fileAttribute);
        }
.
.
.
```

Seem like a lot of code? We can break the new code down into three components: creating an instance variable **fileInfo** that is populated with information about the file name stored in the **fullFileName** variable, extracting information from the **fileInfo** object and assigning that data to our different user interface elements, and the adding a private method **CheckFileAttribute** to make it easier to extract bit information from the **fileInfo** object. Whew! Let's talk about each of these points.

fileInfo is an object that is part of the System.IO assembly. We again used the **new** keyword to create the object, and passed into the constructor the name of the file the user selected from the open file dialog. We next extracted the file information from the **fileInfo** object, and assigned the appropriate values to our user interface elements. Some of the elements returned a non-string value, so we had to use the ToString() method to convert the information to a string for our user interface elements. For our CheckBox controls, we used the Checked property: setting the property to true checks the check box, and setting it false unchecks the check box.

Finally, some of the information we needed from the **fileInfo** object had to do with attributes of the file, such as whether or not it was read-only. The **fileInfo** object uses bits to store this information, requiring us to use bit ANDing and comparison. What does that mean? The information for different attributes is stored as a pattern of bits (those 0s and 1s we always hear about related to computers), and by using the & C# bit AND operator, and a specific bit pattern, if that pattern of bits is within the value, if we compare the results of the bit AND operation and it matches the pattern, we know the original pattern was within the value. Sound confusing? For now, let's just stick to understanding the syntax.

We created a convenience method, **CheckFileAttribute**, to make it easier to check the different attributes. Also, without the **CheckFileAttribute** method, we would have had to repeat the attribute comparison code. Whenever you see yourself repeating code, think about creating a method that encapsulates that code. The **CheckFileAttribute** method takes the **fileInfo** object as the first parameter, and the attribute we want to check against as the second parameter, returning the results of the bit AND and comparison operation as a boolean value. We use this boolean result to set the appropriate CheckBox control.

Form Icon

The last thing we need to do to our application is add the icon that you should see at the top left corner of the Form.

Right-click the link 174_magnify_uncompressed.ico) and select **Save Target As...**. Save the file in your **My Documents** folder, which should be the default. This icon is another file available from Visual Studio.

Click on the form to select it, and in the Properties Window, select the **Icon** property. Click on the ellipsis (...) to the right of the Icon property, and use the Open dialog box to select the file **174_magnify_uncompressed.ico** file from your **My Documents** folder. Click **Open**.

The icon in the title bar of the form should now be changed! So, with this last step, you've completed the application!

Functional Requirements

Now that we've created our FileInfo application, and are more sensitive to using standards as we create our user interface, let's discuss the functionality of our application. What does our application do?

Try to produce a list that summarizes what the FileInfo application does.

Let's see how close you came to our list.

- Select a file.
- List the file name selected.
- List the path of the selected file.
- List attributes of the selected file: archive, normal, read-only, compressed, encrypted, system, hidden, directory.
- Close the application.

Were you close? As you can see, the functional list is just that: a list of the application's functions and what they do. As part of planning your application, knowing in advance what you are trying to create helps you to create the correct final product. Our sample application was pretty simple, so we specified a lot of details. In larger, more complex applications, you may have much broader definitions in your functional list, such as "create an application that displays all available information and attributes of a user-selected file." But, if you were creating an application with hundreds of Forms or Windows, a brief functional list is all you might start out with, slowly expanding the details of each area as you work through that section. Often, a brief functional definition is termed a *Statement of Purpose*, and if you review our final brief statement, that statement certainly qualifies.

As we continue to develop software, try to think in terms of functionality first. You could create a basic *Statement of Purpose*, followed by a more detailed functional list. Then, you might create mockups, also known as *prototypes* in computer science, of your user interfaces (usually with no code, or just enough to allow you to navigate through your user interface to see the prototype).

Finally, to add the final emphasis to this section on functional requirements, a topic we'll be going over in more depth, remember the carpenter's rule: measure twice, cut once! That same principle applies to software engineering: plan before you code!

Next

That's it for this lesson! Now that you have completed this lesson, complete any associated projects, then quizzes.

Designing A Program

Introduction

As we learned in the previous lesson, creating software is more than writing code, but is instead an act of engineering, and as such requires planning. As we continue with the lessons and develop the skills needed for software engineering, we'll add a framework to help us envision and implement the planning and development process. We will eventually create formalized documents, such as requirements design documents, but for now, we want to discuss the basic concepts behind such documents: what a programming application should do, and how we will make the code do what it should do. We will also introduce a method of mapping functionality to objects by creating a task list, link these tasks to objects, and then map both to events. This model is known as a *Task Object Event (TOE)* chart. Finally, we'll introduce the concepts of incremental development, and planning and tracking the development process using milestones.

Functional Requirements

The What

When creating software, even if you're just creating something for fun (do people really do that?), the software needs a purpose: what will the software do? Consider the following:

- Who will use the software?
- What would these users want the application to do?
- Can we create a list of these application features?
- Are there any special considerations for this software, such as what operating system, security issues, etc.?
- What is the timeline for completing the software?
- If you plan on selling the software, what about marketing issues?

Note For fun, feel free to take a look at the Wikipedia article for Product Requirements Document.

This list shows only some of the many issues to think about when you consider what the software you create should do. As you've worked through the lessons, we've described in general the sample coding you would be doing during the lesson, then moved directly to writing the code. You, however, have not really had much chance to create a list of functionality from such general descriptions, nor determined how you might code the application based on this functionality list. For the purposes of this lesson, we will build upon the simple web browser application we created in a previous lesson, adding features that will enable us to search the web page source code for specific text.

In fact, let's create a single *Statement of Purpose*: Create an application that enables loading a web page, displays the source code for that web page, and allows case-sensitive and case-insensitive searching within the web page source code, highlighting any located search text.

From this statement of purpose, let's create a list of what this application should do. You try first, then you can review our *functional list*.

Review the Statement of Purpose given above, and put together a functional list of what the program should do.

Here is our list.

- Provide a means for entering a web page URL.
- Provide a means for navigating to the user-entered web page URL.
- Display the web page contents.
- Display the source of the web page contents.
- Provide a means of switching between the web page contents and the source of the web page.
- Provide a means for entering search text.
- Provide a means for selecting case-sensitive or case-insensitive searching.

- Provide a means for searching for the user-entered search text using case selection.
- Highlight the user-entered text if found.
- Alert the user if the text is not found.
- Display the position of the found text within the web page source code.
- Alert the user to any current activity, such as loading page, or searching for text.
- Provide a means of exiting the application.

What a list! Were there any items in the list you hadn't thought about? Are you already envisioning what such an application might look like? Wait—not yet! Let's first discuss the *alert* items. What are they for? Whenever you create an application, you should consider notifying the user if the application is busy (one of those 20 tips). All of us tend to be very impatient, and unless we're told that something is going on, we'll assume it isn't. For this application, there might be a delay as the web page is found and loaded, so we need to let the user know. To be consistent, we also will alert the user when a search is conducted. For this application, the search will most likely be so fast the alert may never be seen by the user, but that doesn't mean we shouldn't display it! The user's computer might be running slow, or they might loaded a really large web page, and the search might just take a few seconds.

Next, let's take a look at creating a TOE (no foot jokes, please!).

Tasks, Objects, Events (TOE)

A Task Object Event (TOE) chart is simply a three-column chart that lists the tasks your application (or a part of your application) will perform, the object involved with that task, and any events that the object might require to perform the task. A TOE chart is an organizational tool for helping you to create your application. We will use our functional list from above for the tasks, adding the objects by name, and adding any related events. Here's our TOE chart for this application.

Task	Object	Event
Provide a means for entering a web page URL.	urlTextBox	None
Provide a means for navigating to the user-entered web page URL.	loadPageButton	Click
Display the web page contents.	webBrowser	DocumentCompleted
Display the source of the web page contents.	richTextBox	None
Provide a means of switching between the web page contents and the source of the web page.	tabControl	None
Provide a means for entering search text.	searchTextBox	None
Provide a means for selecting case-sensitive or case-insensitive searching.	matchCaseCheckBox	None
Provide a means for searching for the user-entered search text using case selection.	searchButton	Click
Highlight the user-entered text if found.	richTextBox	None
Alert the user if the text is not found.	positionLabel	None
Display the position of the found text within the web page source code.	positionLabel	None
Alert the user to any current activity, such as loading page, or searching for text.	statusLabel webBrowser searchButton	None Navigating Click
Provide a means of exiting the application.	exitButton	Click

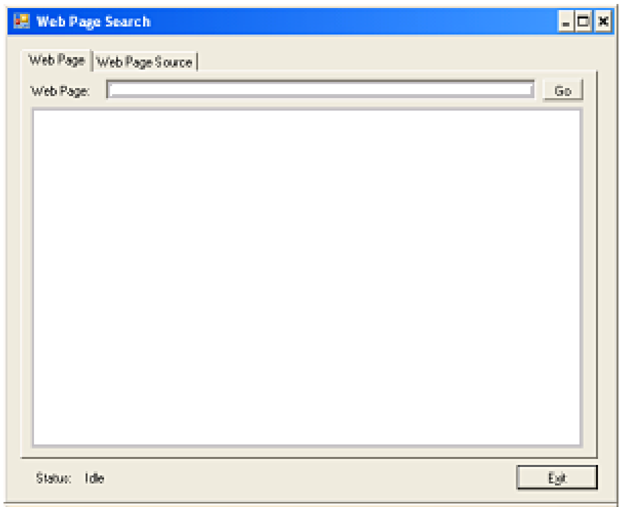
Note Some labels might be empty in the initial state; remember to set their **AutoSize** properties to **false**.

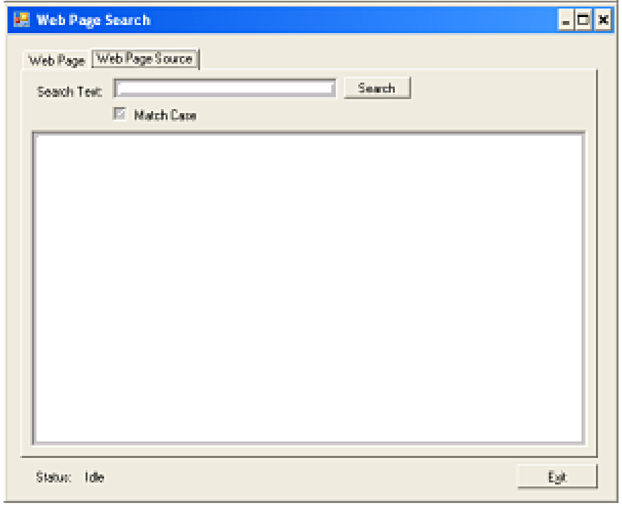
How's that for matching up the tasks of the application to the objects they relate to, and any necessary related events? Not bad! So, taking the functional list and the TOE, let's discuss the design next.

Design

As we mentioned earlier, the *design* is the "how" portion of the planning. It is very common to see a prototype (or mockup) of different user interfaces as part of the functional component of the planning, but doing so does borrow somewhat from the how (design). That's okay, it works. As you look at the TOE chart, you'll see that we have definitely thought about what elements the user interface (UI) is going to need to deliver on the required functionality. A UI mockup that is meant solely to convey the functionality as it relates to the other functional elements is definitely part of any functional requirements document. When that prototype begins to focus on what the final user interface will look like, it belongs with the design components. Confused? Don't be! For now, we're simply going to present our UI mockup, based on our TOE.

The two UI mockups below use another new control: a tab control. You've probably seen it in other applications. We'll use it to let the user switch between the web page content and the source for the web page content, where the user can search for text in the content. Because we have two tab pages, we have to show both mockup screens.





So, with a Statement of Purpose, Functional Requirements, the TOE chart, and the mockups, we're ready to get coding!

From Mockup to UI Prototype

To begin, let's create the mockup.

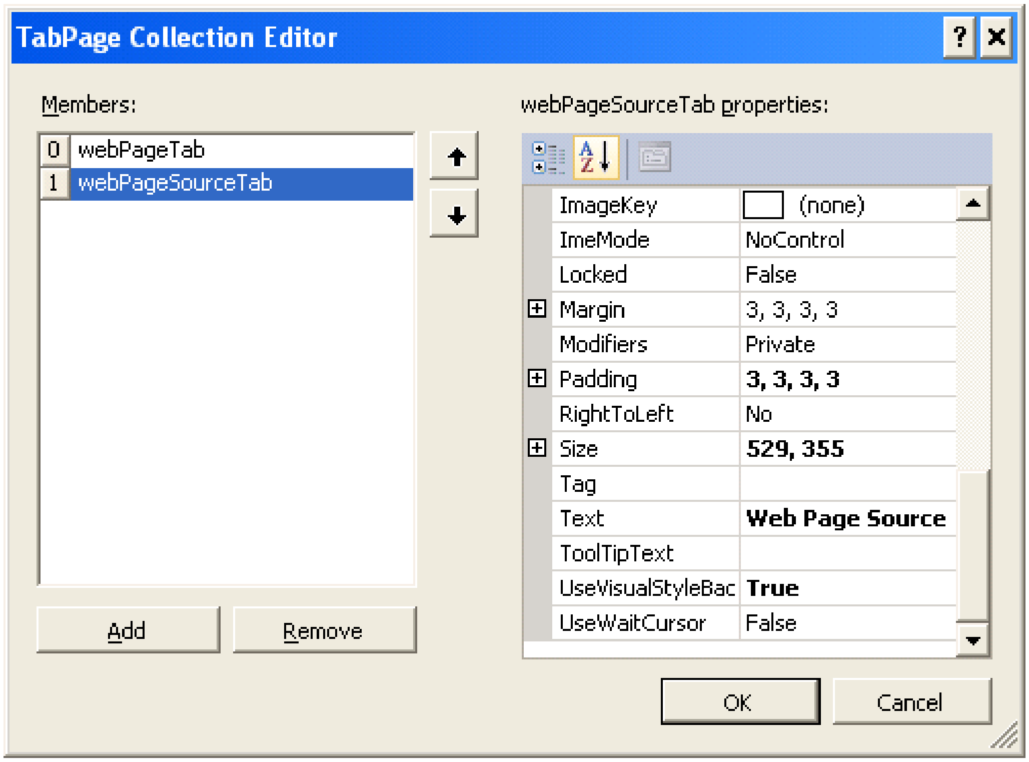
Create a new Visual Studio Windows Forms Application by selecting **File | New | Project**. In the New Project dialog box, change the Name of the project to **WebPageSearch**; all other items in this dialog should keep their default settings. Click **OK**. In the Solution Explorer, click the **Form1.cs** entry, pause, and click it again, and replace the highlighted text, **Form1.cs**, with **WebPageSearch.cs**. Click to save the project.

Just as with the last lesson, we won't walk through adding all of the screen elements in detail; instead, you'll use the mockups as your guide for placing the different elements, and the TOE chart to give you their Name properties. One thing you should be aware of: the positionLabel is *not* visible on the mockup, but is a Label control placed to the right of the searchButton of the Web Page Source Tab control. We will give you guidance on setting up the Tab control. We're also using a RichTextBox control, rather than a text box, so we can highlight the search text as it's found.

Tip

Drag the Tab control onto the Form first, and make sure, as you add the controls for each page of the Tab control, that you drag them onto that page, and not onto the form itself. Also, when adding the WebBrowser control, or any control, the control may take up the entire form. This is because the control is automatically docking itself within the containing Form control. To undock the WebBrowser control, click the black popup menu indicator (▸) located at the top right of the WebBrowser control, and select **Undock in Parent Container**.

Before dragging other controls onto the Form, drag a Tab control onto the Form. In the Properties Window, click the ellipsis to the right of the TabPages property. Click **tabPage1** in the Members list and change the Text property to **Web Page**, and the Name property to **webPageTab**. Click **tabPage2** in the Members list, then change the Text property to **Web Page Source**, and the Name property to **webPageSourceTab** (click **OK** when you finish):



Members:

0 webPageTab
1 webPageSourceTab

webPageSourceTab properties:

ImageKey	(none)
ImeMode	NoControl
Locked	False
⊞ Margin	3, 3, 3, 3
Modifiers	Private
⊞ Padding	**3, 3, 3, 3**
RightToLeft	No
⊞ Size	**529, 355**
Tag	
Text	**Web Page Source**
ToolTipText	
UseVisualStyleBac	**True**
UseWaitCursor	False

Add the rest of the controls onto the Form, setting their Names and Text properties using the mockups and TOE chart.

Not too bad! Next, let's add the code to make our application perform the tasks from the TOE chart. But, before we add the code, let's investigate the concept of incremental development.

Incremental Development

What is incremental development? Just as the name implies, it's developing something incrementally; in other words, in small steps. What steps have we seen so far, and what will we be adding next? See the list below.

- Statement of Purpose
- Functional requirements
- UI Mockup
- TOE Chart
- UI Prototype
- Coding Prototype
- Testing

Did you realize we had so many steps? With incremental development, each step should be reviewed by all interested parties (often referred to as *stakeholders*) before you complete it and proceed to the next step. Sometimes, an individual step will raise questions, and require modifications to prior steps, which means you should go back to that former step, make the necessary change, and then again have the stakeholders review the step before moving on. Also, within a step, you may iterate through the step, having the stakeholders conduct a review through each iteration. For example, the process of converting the UI Mockup to the UI Prototype may raise issues that make it impossible to create the UI Prototype exactly like the UI Mockup. You don't have to change the UI Mockup, although the TOE Chart will probably change, but you would iterate through the UI Prototype, and possibly the TOE Chart, steps, until all the stakeholders were satisfied with the results. That's modern software engineering! Bear in mind, we have *not* added all of the possible steps, just those steps that apply to our application development. Most software development companies or departments have extensive software engineering models that are strictly enforced to ensure the software they produce meets the needs of all of the stakeholders!

Note

The term *prototype* is meant to indicate that a product is an early version. After the first iteration, the term prototype is removed, and a versioning (numbering) system is used. Typically, only the first iteration of a product, or portion of a product, is termed a prototype. The term *mockup* is often used when an attempt is made to explain functionality and possible design ideas, and does not require that the UI presented use real UI elements (such as TextBox or Button controls), although most do.

So, we've completed all of the steps down to Coding Prototype, and since we're the only interested stakeholders, we approve the prior steps, and are ready to proceed!

Adding the Code

The work of adding the code to create our Coding Prototype can be divided into phases:

- Add the application exit.
- Code the web browser navigation.
- Code the web browser source into the rich text control.
- Add the search and highlight capability.
- Add the alert/status updates.

Let's proceed!

Add the application exit.

Modify **WebPageSearch.cs** as shown below.

CODE TO TYPE: WebPageSearch.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 WebPageSearch
{
    public partial class WebPageSearch : Form
    {
        public WebPageSearch()
        {
            InitializeComponent();
        }

        private void ~~label1~~exitButton_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

Remember to go to the Design view, click the Exit button, and add exitButton_Click to the Click Event in the Properties Window. 💾 and test your changes using ▶. We've discussed the code to close a Form before, so we'll skip the explanation and move on to the next step.

Code the web browser navigation

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

CODE TO TYPE: WebPageSearch.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 WebPageSearch
{
    public partial class WebPageSearch : Form
    {
        public WebPageSearch()
        {
            InitializeComponent();
        }

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

        private void loadPageButton_Click(object sender, EventArgs e)
        {
            if (urlTextBox.Text.Length > 0)
            {
                webBrowser.Navigate(urlTextBox.Text);
            }
        }
    }
}
```

Go back to the Design Editor and select the **loadPageButton** control by clicking on it. In the Properties Window, click ⚡. Scroll through the event list for the button, select the **Click** event, and change the dropdown at the right to **loadPageButton_Click**. Save 💾 and test with the ▶ button.

Let's discuss how this code works.

OBSERVE: WebPageSearch.cs

```
.
.
.
        private void loadPageButton_Click(object sender, EventArgs e)
        {
            if (urlTextBox.Text.Length > 0)
            {
                webBrowser.Navigate(urlTextBox.Text);
            }
        }
.
.
.
```

We added code to the **loadPageButton_Click** event that tests if **urlTextBox.Text** has any content, and if so, calls the **Navigate** method of the **webBrowser** control, using the **urlTextBox.Text** contents as a parameter. The **webBrowser** control will be populated with the contents of the web page.

Code the web browser source into the rich text control.

Modify **WebPageSearch.cs** as shown:

CODE TO TYPE: WebPageSearch.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 WebPageSearch
{
    public partial class WebPageSearch : Form
    {
        public WebPageSearch()
        {
            InitializeComponent();
        }

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

        private void loadPageButton_Click(object sender, EventArgs e)
        {
            if (urlTextBox.Text.Length > 0)
            {
                webBrowser.Navigate(urlTextBox.Text);
            }
        }

        private void webBrowser_DocumentCompleted(object sender, WebBrowserDocum
entCompletedEventArgs e)
        {
            richTextBox.Text = webBrowser.DocumentText.ToString();
        }
    }
}
```

Go back to the Design Editor, and select the **webBrowser** control by clicking on it. In the Properties Window, click the ⚡ icon. Scroll through the event list for the Browser, select the **DocumentCompleted** event, and change the dropdown at the right to **webBrowser_DocumentCompleted**. 💾 and test with ▶.

Let's discuss how this code works.

OBSERVE: WebPageSearch.cs

```
.
.
.
        private void webBrowser_DocumentCompleted(object sender, WebBrowserDocum
entCompletedEventArgs e)
        {
            richTextBox.Text = webBrowser.DocumentText.ToString();
        }
.
.
.
```

The **webBrowser_DocumentCompleted** event is executed when the contents of the web page finish loading. When this happen, we extract the web page source contents using the **DocumentText** method of the **webBrowser** control. We assign the web page source contents to the **Text** property of the **richTextBox** control. After you load a web page, if you switch to the **Web Page Source** tab, you will see that the RichTextBox control is populated with the web page source code (which can be quite messy looking, depending on which site you visit).

Add the search and highlight capability.

Modify **WebPageSearch.cs** as shown:

CODE TO TYPE: WebPageSearch.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 WebPageSearch
{
    public partial class WebPageSearch : Form
    {
        private int _startSearchPosition = 0;

        public WebPageSearch()
        {
            InitializeComponent();
        }

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

        private void loadPageButton_Click(object sender, EventArgs e)
        {
            if (urlTextBox.Text.Length > 0)
            {
                webBrowser.Navigate(urlTextBox.Text);
            }
        }

        private void webBrowser_DocumentCompleted(object sender, WebBrowserDocum
entCompletedEventArgs e)
        {
            richTextBox.Text = webBrowser.DocumentText.ToString();
        }

        private void searchButton_Click(object sender, EventArgs e)
        {
            string searchString = searchTextBox.Text;
            int position = -1;

            if (searchString.Length > 0)
            {
                if (matchCaseCheckBox.Checked)
                {
                    position = richTextBox.Find(searchString, _startSearchPositi
on, RichTextBoxFinds.MatchCase);
                }
                else
                {
                    position = richTextBox.Find(searchString, _startSearchPositi
on, RichTextBoxFinds.None);
                }
                if (position >= 0)
                {
                    _startSearchPosition = position + 1;
                    positionLabel.Text = position.ToString();
                    richTextBox.SelectionColor = Color.Red;
                    richTextBox.Select(position, searchString.Length);
                    richTextBox.ScrollToCaret();
                }
                else
                {
                    positionLabel.Text = "Not found";
                    _startSearchPosition = 0;
                }
            }
        }
    }
}
```

Go back to the Design Editor, and select the **searchButton** control by clicking on it (this Button control is on the Web Page Source tab of the TabControl). In the Properties Window, click the ⚡ icon. Scroll through the event list for the button, select the **Click** event, and change the dropdown at the right to **searchButton_Click**. 💾 and then test your changes with ▶.

Let's discuss how this code works.

OBSERVE: WebPageSearch.cs

```
.
.
.
        private int _startSearchPosition = 0;

.
.
.

        private void searchButton_Click(object sender, EventArgs e)
        {
            string searchString = searchTextBox.Text;
            int position = -1;

            if (searchString.Length > 0)
            {
                if (matchCaseCheckBox.Checked)
                {
                    position = richTextBox.Find(searchString, _startSearchPositi
on, RichTextBoxFinds.MatchCase);
                }
                else
                {
                    position = richTextBox.Find(searchString, _startSearchPositi
on, RichTextBoxFinds.None);
                }
                if (position >= 0)
                {
                    _startSearchPosition = position + 1;
                    positionLabel.Text = position.ToString();
                    richTextBox.SelectionColor = Color.Red;
                    richTextBox.Select(position, searchString.Length);
                    richTextBox.ScrollToCaret();
                }
                else
                {
                    positionLabel.Text = "Not found";
                    _startSearchPosition = 0;
                }
            }
        }
    }
}
```

It might seem like a lot of code, but the **searchButton_Click** event handler simply retrieves the **searchString**, sets a default **position** variable to indicate an invalid location for **searchString** in the web page source content, verifies that **searchString** has something in it (Length > 0), searches for the text (case sensitive or case insensitive), and if the string is found, highlights it. A found string also updates the

positionLabel Label control with a numeric value indicating where in the document the **searchString** was found, or to **"Not Found"** if the **searchString** is not found.

We also added a class variable, **_startSearchPosition**, so we can remember where we left off with our previous search. This way, the user can continue to click the **searchButton** Button control and find the next occurrence of **searchString**, moving through the document until no more **searchString** values are found.

When conducting the search, we test the value of the **matchCaseCheckBox** CheckBox control. If the user has checked the box, we specify a case-sensitive parameter, **RichTextBoxFinds.MatchCase**, for the **richTextBox** Find method. If not, we specify **RichTextBoxFinds.None** instead. The second parameter of the **richTextBox** Find method indicates the position to start the next search. By using the **_startSearchPosition** class variable, we can remember that last position, plus one, where we found the last **searchString**.

The highlighting of the found **searchString** uses a number of methods of the **richTextBox** control: the SelectionColor to set the color to red, the Select to do the highlighting, and the ScrollToCaret method to ensure wherever the **searchString** is found, the found text will be visible in the RichTextBox control.

Add the alert/status updates.

Modify **WebPageSearch.cs** as shown:

CODE TO TYPE: WebPageSearch.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 WebPageSearch
{
    public partial class WebPageSearch : Form
    {
        private int _startSearchPosition = 0;

        public WebPageSearch()
        {
            InitializeComponent();
        }

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

        private void loadPageButton_Click(object sender, EventArgs e)
        {
            if (urlTextBox.Text.Length > 0)
            {
                webBrowser.Navigate(urlTextBox.Text);
            }
        }

        private void webBrowser_Navigating(object sender, WebBrowserNavigatingEventArgs e)
        {
            Cursor = Cursors.WaitCursor;
            statusLabel.Text = "Loading...";
        }

        private void webBrowser_DocumentCompleted(object sender, WebBrowserDocumentCompletedEventArgs e)
        {
            richTextBox.Text = webBrowser.DocumentText.ToString();
            statusLabel.Text = "Idle";
            Cursor = Cursors.Default;
        }

        private void searchButton_Click(object sender, EventArgs e)
        {
            string searchString = searchTextBox.Text;
            int position = -1;

            if (searchString.Length > 0)
            {
                statusLabel.Text = "Searching...";
                if (matchCaseCheckBox.Checked)
                {
                    position = richTextBox.Find(searchString, _startSearchPosition, RichTextBoxFinds.MatchCase);
                }
                else
                {
                    position = richTextBox.Find(searchString, _startSearchPosition, RichTextBoxFinds.None);
                }
                if (position >= 0)
                {
                    _startSearchPosition = position + 1;
                    positionLabel.Text = position.ToString();
                    richTextBox.SelectionColor = Color.Red;
                    richTextBox.Select(position, searchString.Length);
                    richTextBox.ScrollToCaret();
                }
                else
                {
                    positionLabel.Text = "Not found";
                    _startSearchPosition = 0;
                }
                statusLabel.Text = "Idle";
            }
        }
    }
}
```

Go back to the Design Editor, and select the webBrowser control by clicking on it. In the Properties Window, click the Lightning Bolt icon. Scroll through the event list for the webBrowser, select the Navigating event, and change the drop-down at the right to **webBrowser_Navigating**. Save and test it before you continue. 💾 and ▶. As noted, unless you load and search a very large or slow web page, you might not see the changign status messages.

Let's discuss how this code works.

OBSERVE: WebPageSearch.cs

```
.
.
.
        private void webBrowser_Navigating(object sender, WebBrowserNavigatingEventArgs e)
        {
            Cursor = Cursors.WaitCursor;
            statusLabel.Text = "Loading...";
        }

        private void webBrowser_DocumentCompleted(object sender, WebBrowserDocumentCompletedEventArgs e)
        {
            richTextBox.Text = webBrowser.DocumentText.ToString();
            statusLabel.Text = "Idle";
            Cursor = Cursors.Default;
        }

        private void searchButton_Click(object sender, EventArgs e)
        {
            string searchString = searchTextBox.Text;
            int position = -1;

            if (searchString.Length > 0)
            {
                statusLabel.Text = "Searching...";
                if (matchCaseCheckBox.Checked)
                {
                    position = richTextBox.Find(searchString, _startSearchPosition, RichTextBoxFinds.MatchCase);
                }
                else
                {
                    position = richTextBox.Find(searchString, _startSearchPosition, RichTextBoxFinds.None);
                }
                if (position >= 0)
                {
                    _startSearchPosition = position + 1;
                    positionLabel.Text = position.ToString();
                    richTextBox.SelectionColor = Color.Red;
                    richTextBox.Select(position, searchString.Length);
                    richTextBox.ScrollToCaret();
                }
                else
                {
                    positionLabel.Text = "Not found";
                    _startSearchPosition = 0;
                }
                statusLabel.Text = "Idle";
            }
        }
    }
}
```

Most of the new code simply **updates the statusLabel control's Text property**. We also change the **Cursor** for the application if something is happening and we want to alert the user that they should wait (patiently!). We use the **WaitCursor** to change the mouse pointer to an hourglass, and then change it back to the **Default**.

We did add a **webBrowser_Navigating** event handler, so we can update the user while the web page contents are being downloaded and loaded into our webBrowser.

That completes the coding. Pretty cool, huh?!

Tracking Development and Milestones

Throughout this lesson, we've worked through the steps of planning and creating the application. Although this application wasn't a very big project, it might have taken you some time to work through all of the steps. You might even have taken a break (or two). Just as we sometimes set goals in our life to achieve specific outcomes, when developing software, we set goals, but we typically call them milestones. When we list the incremental steps, we might include date and time estimates for completing each one. Also, sub-steps within each major step might be given estimated completion dates and times. Establishing milestones is a significant part of the software engineering process. Why? Usually, we're creating software for a reason: to make a personal task easier, as an assignment, or as part of a work-related task. Milestones assist in orchestrating the successful and timely completion of our development efforts. So, for this project, we had seven milestones, and one of the milestones, Coding Prototype, had five milestones. Not too bad!

Another issue that should be raised is the fact that larger software development projects may have multiple resources (developers, architects, UI designers, writers, testers, etc.). Milestones are an integral part of project management, so much so that software like Microsoft Project is used to track the entire development project, including the time element, the tasks, and the resources.

Next

That's it for this lesson! Now that you have completed this lesson, complete any associated projects, then quizzes.

Wrapping It All Up

Introduction

Congratulations, you've completed all of the lessons in this first course! But, wait, what about this final lesson? In this lesson, we will create a final application that pulls together most of the concepts we've learned throughout this course. This lesson will get you started on the final project, leaving the rest of the project for you to complete on your own. Let's get started!

The Project

The Assignment

The final project will be an application that will record movie information. The amount of information is not extensive, but in creating the application, you will create a Movie class and an Actor class. Conceptually, a movie is made up of actors, so a movie object will include an instance of an actor object, as we're going to include the main actor of the movie. We'll also create a **Movies** collection using a new kind of object called an *ArrayList*. You will be able to add a movie (and the actor), and the movie object will be saved to the ArrayList, and a ListBox will display the movie name and the main actor. You will also be able to clear the movie list, which will also clear the ArrayList, as well as clear the movie entry controls.

In creating this final project, we will use the planning tools we've learned so far. You will create the project using the following material:

- Statement of Purpose
- UI Prototype
- Class Design Diagram
- TOE Chart
- Use Case UI Prototype

The last item is new. A *use case* is a description of the behavior of an application as it responds to specific interaction with a user. Use cases typically result in an application state change. Our application has these states:

- Initial state
- Movie entry state
- Movie entry cleared state
- Movie added state
- Movie list cleared state

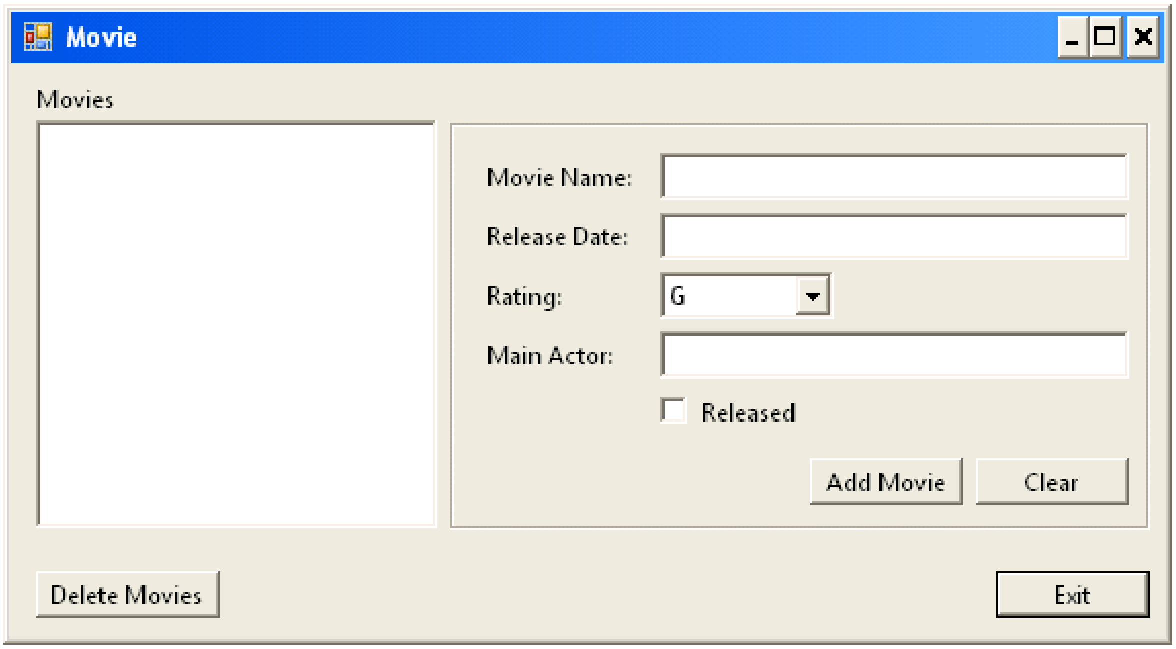
We're going to include one of these states: movie added state. Displaying the state will also show you the status message that should be included to inform the user of what just happened. So, let's get you the material you need to get started!

The Materials

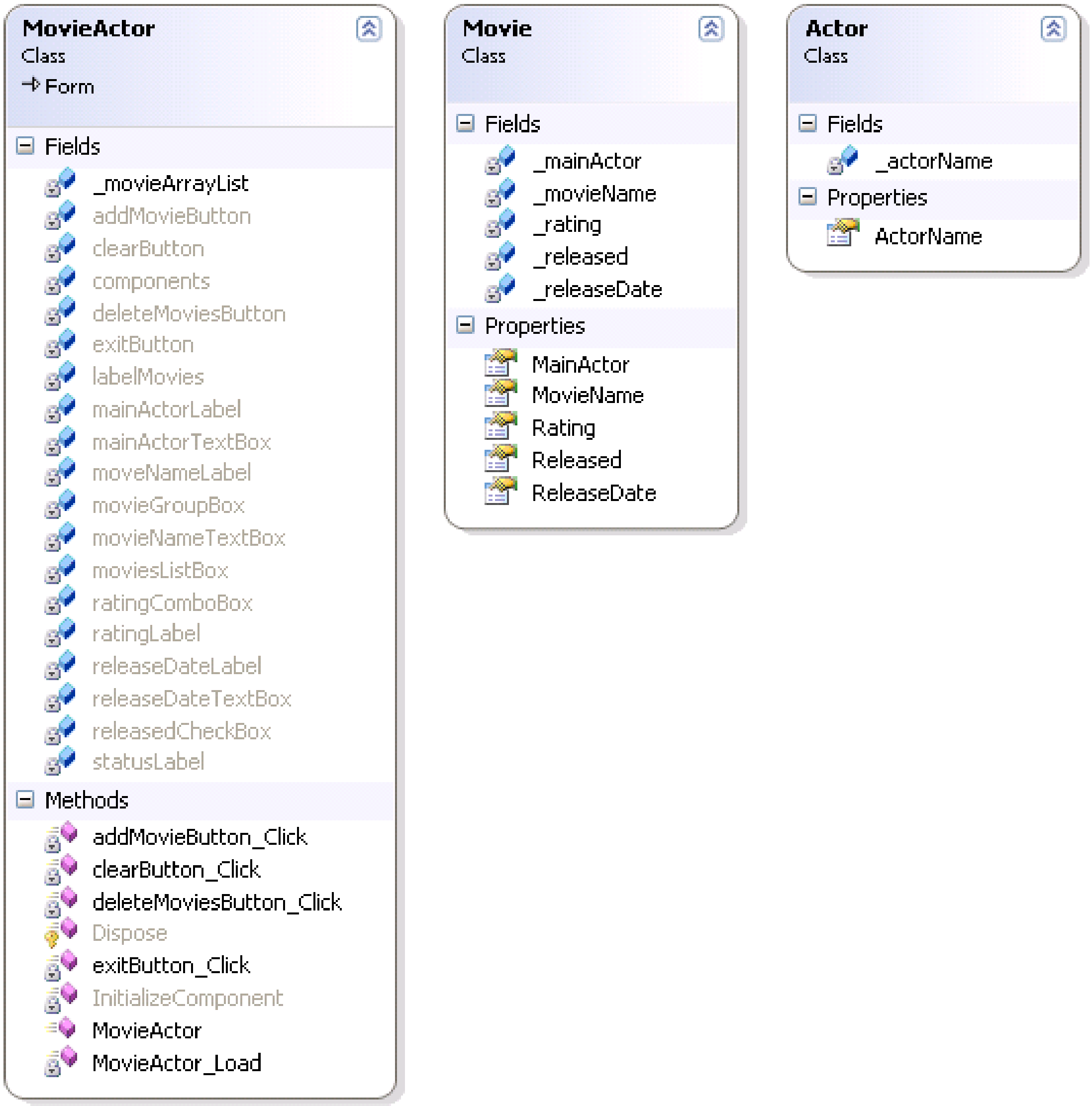
Statement of Purpose

The application creates a list of movies, storing information about each movie (movie name, release date, rating, main actor, and if the movie has been released). Display only the movie name and main actor in the list. The movie entry form must support being reset (cleared), and the movie list must also support being cleared. Include all United States ratings as options.

UI Prototype



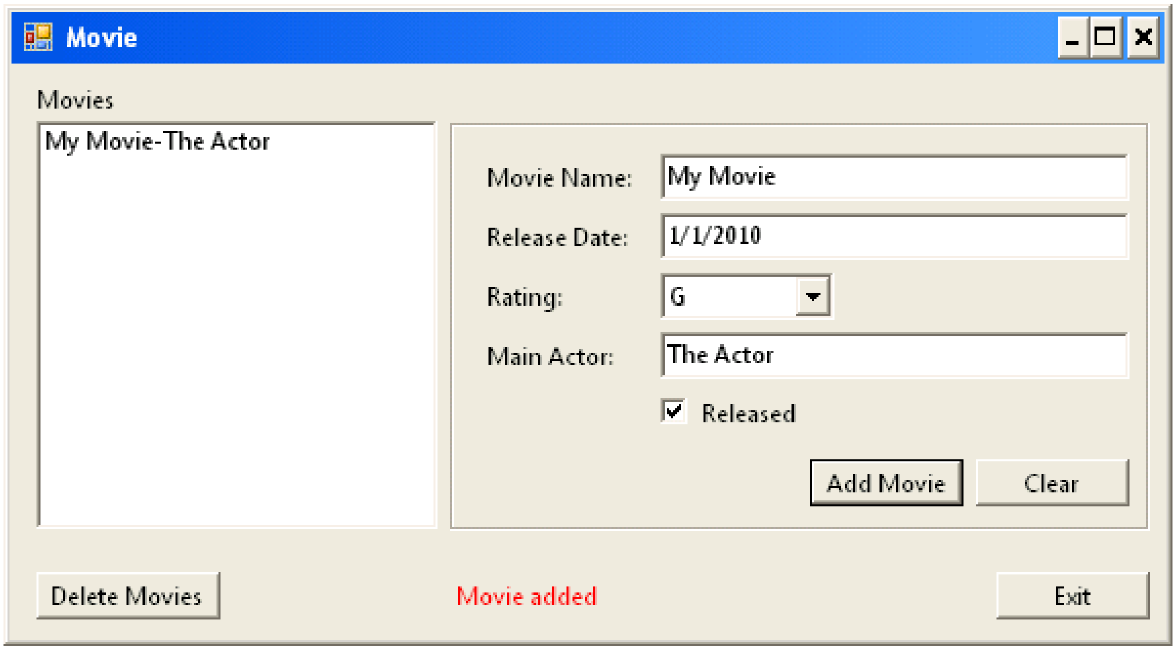
Class Design Diagram



TOE Chart

Task	Object	Event
Set movie name	movieNameTextBox	None
Set movie release date	releaseDateTextBox	None
Set movie rating	ratingComboBox	
Set movie main actor	mainActorTextBox	None
Set movie released status	releasedCheckBox	None
Add movie	addMovieButton moviesListBox _moviesArrayList	Click None None
Clear movie entry controls	clearButton	Click
Clear movie list	deleteMoviesButton moviesListBox _moviesArrayList	Click None None
Exit application	exitButton	Click

Use Case UI Prototype: movie added state



My Movie-The Actor

Movie Name: My Movie

Release Date: 1/1/2010

Rating: G

Main Actor: The Actor

☑ Released

Movie added

The Milestones

That's a lot of information! Let's break up the project into a set of steps, or milestones. Later, we'll explain some of the steps that you might be scratching your head about (the steps marked with a *).

- Create the MovieActor project.
- Rename the default Form
- Set the Form font and point size. *
- Create the user interface. *
- Use the TOE Chart to rename all of the user interface elements.
- Add the System.Collections using statement to support using the C# ArrayList collection. *
- Use Visual Class Designer to add the Actor and Movie class.
- When adding the _mainActor field for the Movie class, make the datatype of the field Actor.
- When adding the MainActor property, do NOT use Refactor/Encapsulate, but instead manually add the property, setting the type to string.
- Use Visual Class Designer to add the ArrayList field to the MovieActor class. *
- Edit the Actor class source code to add default values for fields. *

- Edit the Movie class source code to add default values for fields.
- Edit the Movie class source code to create an instance of the Actor class. *
- Edit the Movie class source code Actor accessor, modifying the get/set values to reference the Actor.ActorName property. *
- Use the Event Property window for the MovieActor Form to add a Form Load event handler. *
- Edit the MovieActor class source code, adding the code to create an instance of the ArrayList collection. *
- Edit the MovieActor class source code, adding the set rating default value functionality. *
- Edit the MovieActor class source code, adding the exit application functionality.
- Edit the MovieActor class source code, adding the clear movie entry functionality.
- Edit the MovieActor class source code, adding the add movie functionality. *
- Edit the MovieActor class source code, adding the clear movie list functionality. *
- Add/verify status display functionality. *
- Test, test, test!

As you read through the milestones, take note of any that you might not have thought of. Spend time thinking 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.

Getting Started

By now, you should be familiar with creating a new project, entering the project name, and changing the default Form name to match the project. We used the Segoe UI 9 point font in earlier lessons, but we set the font for each control. A much easier technique is to set the font and point size for the Form itself, and each control will automatically use this font setting. You'll also notice that we used a GroupBox container control.

The use of a C# ArrayList collection object is new. The ArrayList is part of the System.Collections, so to be able to use the ArrayList data type, we need to include a using statement in our MovieActor class. Also, when we use the Visual Class Designer to add the ArrayList field _moviesArrayList (private class variable) to the MovieActor class, ArrayList will be available in the Class Details Type column.

Note You'll notice that we refer to a MovieActor class. What is this class? The MovieActor class is our Form. Yep, a Form is a class! You probably remember that from earlier lessons, but we wanted to emphasize this fact again.

After using the Visual Class Designer to add the ArrayList field to the MovieActor class, we still need to add code to create an instance of this class as an object. To create an instance, we'll need to use the **new** keyword. Create your MovieActor form and then edit the code as shown to add the default MovieActor class code, with the _moviesArrayList field, the System.Collections assembly for the ArrayList object, and the MovieActor Form Load event handler.

LISTING: MovieActor.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.Collections;

namespace MovieActor
{
    public partial class MovieActor : Form
    {
        private ArrayList _movieArrayList = new ArrayList();

        public MovieActor()
        {
            InitializeComponent();
        }

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

The declaration of the _movieArrayList was added using the Visual Class Designer, and the MovieActor_Load event handler was added using the Event Property window (or by double-clicking anywhere on the Form).

Both the Movie class and the MovieActor class have fields (private member variables) that we should initialize to be consistent, and to indicate that we want these fields to start with a known default value. For both classes, you'll need to add an appropriate initialization code, such as setting strings to empty, and boolean to false (Remember: you should only need to add the highlighted section. All of the rest of the class was generated using the Visual Class Designer):

LISTING: Actor.cs

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

namespace MovieActor
{
    public class Actor
    {
        private string _actorName = "";

        public string ActorName
        {
            get { return _actorName; }
            set { _actorName = value; }
        }
    }
}
```

For the Movie class, you'll need to add initialization code to create an instance of the Actor class. You'll also need to modify the Movie class accessors for the Actor property to get and set the ActorName property of the Actor class. Modify the Actor property of the Movie class as shown below:

LISTING: Movie.cs

```
private Actor _mainActor = new Actor();

public string MainActor
{
    ~~get~~
~~    {~~
~~        throw new System.NotImplementedException();~~
~~    }~~
    get { return _mainActor.ActorName; }
    ~~set~~
~~    {~~
~~    }~~
    set { _mainActor.ActorName = value; }
}
```

The accessor changes allow us to interact with the ActorName property of the Actor class.

For the rating ComboBox, you need to change the DropDownStyle property to **DropDownList**. Use the Items property to set the possible choices for the ratings used in the United States (G, PG, PG-13, R, NC-17). In the MovieActor Form Load event handler, you'll need to set the SelectedIndex property of the ratingComboBox to **0**.

To add a movie, you'll need to create an instance of the Movie class, set the different Movie properties, add the Movie instance to the _moviesArrayList using the Add method, and update the moviesListBox.Items collection with the movie name and main actor using the Add method. The parameter to the _moviesArrayList.Add is the name of the Movie instance object. The parameter to the moviesListBox.Items.Add is a concatenated string that uses the Text values of the movieNameTextBox and mainActorTextBox.

When clearing the movie list, you'll need to clear both the _moviesArrayList and moviesListBox. For the ArrayList, you'll call the Clear method. For the ListBox, you'll call the Items.Clear method.

And finally, for the status, you should update the statusLabel when you clear the movie entry controls, add a movie, or clear the movies.

Wow, right? Well, we can't give you everything! Work through the milestone list, think through what you will need to do, and don't get frustrated! This project combines a lot of material, and completing it will really help you get the "big picture" of planning and executing the development of an application using Studio and C#!

Next

What else? Go to the project section, and complete the MovieActor application. You can do it! Oh, and congratulations for making it!!!