Chapter 12

Nancy's Hands

Graphical User Interfaces

Learning Objectives

- *Create Graphical User Interface (GUI) programs using Windows forms components*
- *Create GUI programs using Windows Presentation Foundation (WPF)*
- *List and describe the parts of a window*
- *Describe how to register event handler methods with component events*
- *State the definition of the term "delegate"*
- *Handle events generated in one object using event handler methods located in another object*
- *Use the Form, TextBox, Button, and Label controls*
- *Automatically arrange controls with the FlowLayoutPanel and TableLayoutPanel controls*
- *Pass references to event handler objects via constructor methods*
- *Manipulate arrays of controls*
- *Add controls to a window's Controls collection*
- *Control programs via menus*
- *Manipulate text in a text box*

Introduction

Nearly every application running on your personal computer (PC) sports a Graphical User Interface (GUI). This chapter shows you how to create GUIs for your programs.

Before we get started, I'd like to share with you some good news and some bad news. First the bad news: An exhaustive treatment of all aspects of Microsoft Windows GUI programming is way beyond the scope of this book. If you want to move beyond what's covered in this chapter, I recommend reading one of the many books available devoted entirely to the subject. A quick search on Amazon should return a handful of popular, up-to-date titles.

Now the good news: You don't need to know a whole lot to create really nice GUIs. Most of the heavy lifting is done for you by the classes found in the System.Windows.Forms namespace. Some of the more important classes to know include *Form*, *TextBox*, *Button*, and *Label*. Add to these an understanding of how events and delegates work and you'll be off to a good start.

I will also teach you how to separate the GUI from other parts of your program. To do this, you'll need to know how to register event handler methods, located in one or more separate classes, with buttons or other GUI components located in your GUI.

An unfortunate mistake many novice programmers make is to rely too heavily upon the GUI designer available in Microsoft Visual Studio. The problem with using the GUI designer is that it makes it difficult to separate concerns. In this chapter, I will show you how to create GUIs by hand. It's not difficult once you get the hang of things. I'll also show you how you can automatically place components within a GUI via layout managers.

For this edition, I've added a section on the Windows Presentation Foundation (WPF), which differs from Forms-based GUIs in that the UI components are specified in XAML markup files, and UI interaction logic is contained in separate but related code-behind files. I will use Visual Studio for the WPF topic because trying to code XAML by hand is a real pain.

After you complete this chapter, the only thing you'll need to do to create spectacular GUIs is to dive deeper into the .NET Framework and use a little imagination.

The Form Class

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include *standard*, *tool*, *borderless*, or *floating* windows. The Form class is also used to create *dialog boxes* and *multiple-document interface* (MDI) windows.

Form Class Inheritance Hierarchy

A Form is a lot of things, as you can see from its inheritance diagram shown in figure 12-1.

Referring to figure 12-1 — A Form is a *ContainerControl*, a *ScrollableControl*, a *Control*, a *Component*, a *MarshalByRefObject*, and ultimately an *Object*. I recommend that you visit the Microsoft Developer Network (MSDN) website and do a little research on the Form class so you can get a better feel for what it can do. A quick review of its methods and properties will give you a few ideas about how you can manipulate the Form class in your programs.

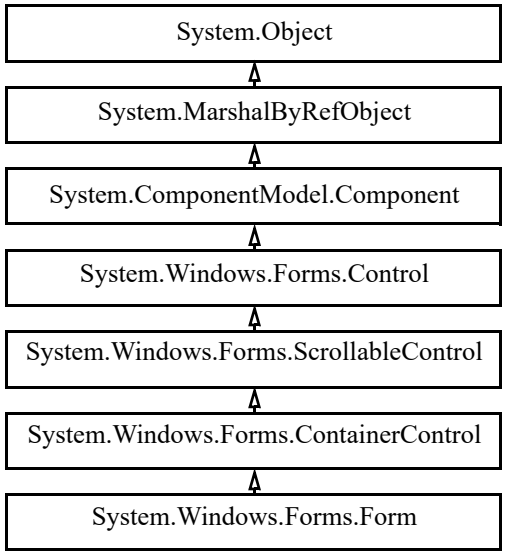

Figure 12-1: Form Class Inheritance Hierarchy

A Simple Form Program

Example 12.1 gives the code for a simple Form-based program. All this program does is display an empty window on the screen. I'll use its output to introduce you to the parts of a standard window.

12.1 SimpleForm.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  public class SimpleForm : Form {
5    public static void Main(){
6      Application.Run(new SimpleForm());
7    }
8  }
```

Referring to example 12.1 — The SimpleForm class extends Form and provides a Main() method. The important point to note here is the use on line 6 of the System.Windows.Forms.Application class to display the form. The Application class's static Run() method starts an *application message loop* on the current thread. Don't worry about threads for now as they are covered in detail in chapter 16. I will discuss messages and the message loop in more detail in the next section.

A Microsoft Windows program is event-driven, meaning that when a window is displayed, it will sit there forever processing events until the application exits. Some of the events are mouse clicks within the window itself or on controls within the window like buttons, text boxes, or menus. Other events may be events sent to the application from other applications, like the operating system, perhaps.

You can compile example 12.1 two ways. If you compile it the way we've been compiling programs up until now, which is just using `csc *.cs`, you will create an application that displays both a window and a command console. Compiling with the `/target:winexe` switch results in an application that displays only the window. The full command required to create a windows executable from example 12.1 is:

```
csc /target:winexe SimpleForm.cs
```

You can compile either way, but you'll find having a console window to display output can come in handy for testing purposes, as you'll see later. Figure 12-2 shows the results of running example 12.1.

Referring to figure 12-2 — When you execute the program either from the command line or by double-clicking, it displays an empty window. It's only empty because I didn't put anything in it, nor did I set any of its properties. However, the empty window has a lot of functionality built in. You can drag the win-

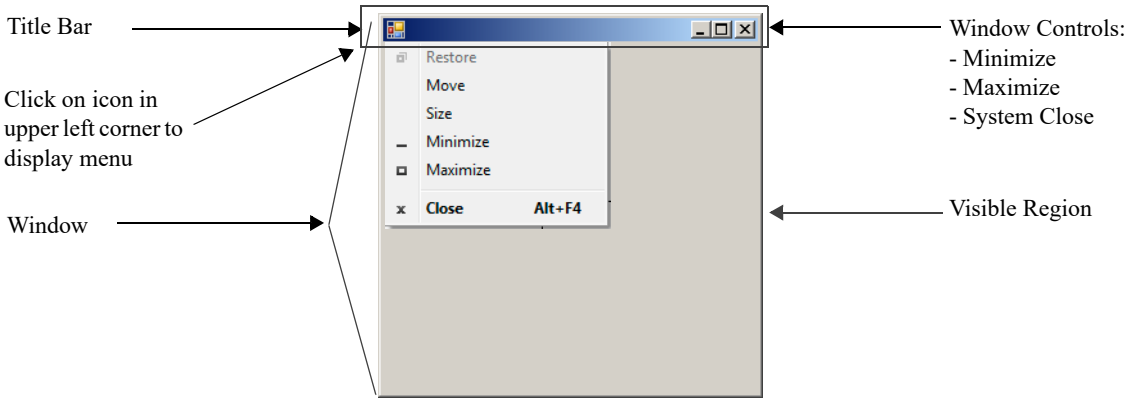

Figure 12-2: Results of Running Example 12.1

dow around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner. Click the icon in the upper left corner of the window to reveal a menu. You can close the window with the keyboard shortcut Alt +F4. It has all the basic functionality you've come to expect from a standard window.

The title bar would have a title in it if I had set the form's *Text* property. I'll show you how to do that in a moment. The window's visible region includes those areas of the window visible to the user. In this case the entire window is visible. If you moved another window over top of this one, then some of it would be visible and some of it would not. Figure 12-3 shows the same window resized smaller and larger.

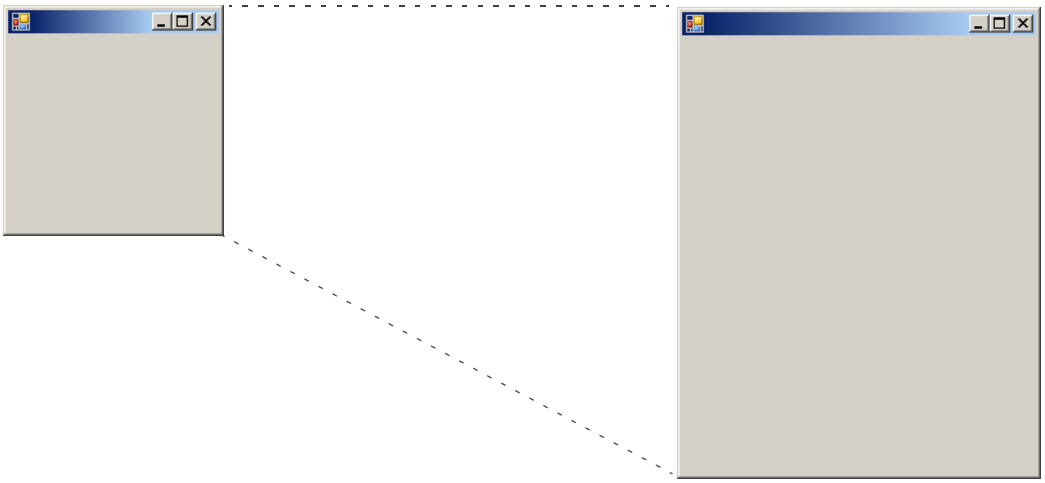

Figure 12-3: A Standard Window can be Resized by Dragging the Lower Right Corner

Quick Review

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include *standard*, *tool*, *borderless*, or *floating* windows. The Form class is also used to create *dialog boxes* and *multiple-document interface* (MDI) windows. A Form is a ContainerControl, a ScrollableControl, a Control, a Component, a MarshalByRefObject, and ultimately an Object.

The Form class provides a lot of functionality right out of the box. You can drag the window around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner or use Alt+F4.

Application Messages, Message Pump, Events, And Event Loop

As I mentioned earlier, Microsoft Windows applications are *event driven*. This means that when a GUI application executes, it sits there patiently waiting for an event to occur such as a mouse click or keystroke. These events are delivered to the application in the form of *messages*. Messages can be generated by the system in response to various types of stimuli including direct user interaction (*i.e.*, mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system-generated messages are placed into a data structure referred to as the *system message queue*. A queue is a data structure that has a first-in-first-out (FIFO) characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window. This process, referred to as the *message pump*, is illustrated in figure 12-4.

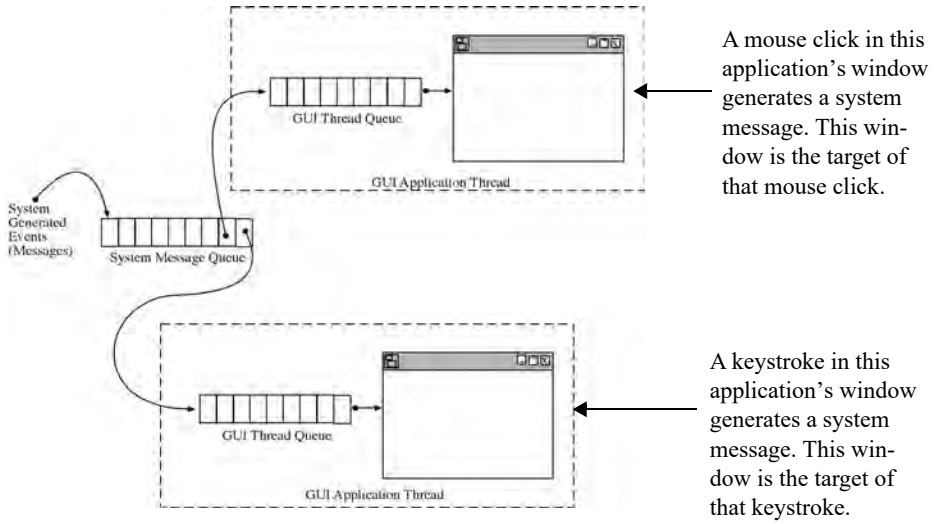

Figure 12-4: Windows Message Routing (Message Pump)

Referring to figure 12-4 — System messages are placed into the system message queue where they wait in line to be processed. The system then examines the data within each message to determine its GUI application target. Each GUI application, which runs in its own thread of execution, has its own message queue. The message is placed in the GUI application's queue where again it waits its turn to be examined and forwarded on to its generating window. **Note:** Applications can have multiple windows open at the same time.

Message Categories

As you can well imagine, many types of events can occur during the execution of a complex GUI application. Each of these application events generates a corresponding system message. The following table lists the system message categories and their message prefixes.

Prefix	Message Category	Prefix	Message Category
ABM	Application Desktop Toolbar	MCM	Month Calendar Control
BM	Button Control	PBM	Progress Bar

Table 12-1: System Message Categories and their Prefixes

Prefix	Message Category	Prefix	Message Category
CB	Combo Box	PGM	Pager Control
CBEM	Extended Combo Box Control	PSM	Property Sheet
CDM	Common Dialog Box	RB	Rebar Control
DBT	Device	SB	Status Bar Window
DL	Drag List Box	SBM	Scroll Bar Control
DM	Default Push Button Control	STM	Static Control
DTM	Date and Time Picker Control	TB	Toolbar
EM	Edit Control	TBM	Trackbar
HDM	Header Control	TCM	Tab Control
HKM	Hot Key Control	TTM	Tooltip Control
IPM	IP Address Control	TVM	Tree-view Control
LB	List Box Control	UDM	Up-down Control
LVM	List View Control	WM	General Window

Table 12-1: System Message Categories and their Prefixes

Messages In Action: Trapping Messages With IMessageFilter

One way to see messages in action is to print them to the console as they occur. The following program is very similar to the SimpleForm code given in example 12.1 in that the MessagePumpDemo class extends Form. It also implements the *IMessageFilter* interface, which declares one method named *PreFilterMessage()*. The PreFilterMessage() method's implementation begins on line 6. All it does in this simple example is write the incoming message to the console.

12.2 MessagePumpDemo.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class MessagePumpDemo : Form, IMessageFilter {
5
6     public bool PreFilterMessage(ref Message m){
7       Console.WriteLine(m);
8       return false;
9     }
10
11    public static void Main(){
12      MessagePumpDemo mpd = new MessagePumpDemo();
13      Application.AddMessageFilter(mpd);
14      Application.Run(mpd);
15    }
16  }
```

Referring to example 12.2 — Any class that implements IMessageFilter can be used as a message filter. In this example, the MessagePumpDemo class uses an instance of itself as a message filter with a call to the Application.AddMessageFilter() method. To see the messages being printed to the console, compile this program into an ordinary console executable file. Figure 12-5 shows the results of running this program.

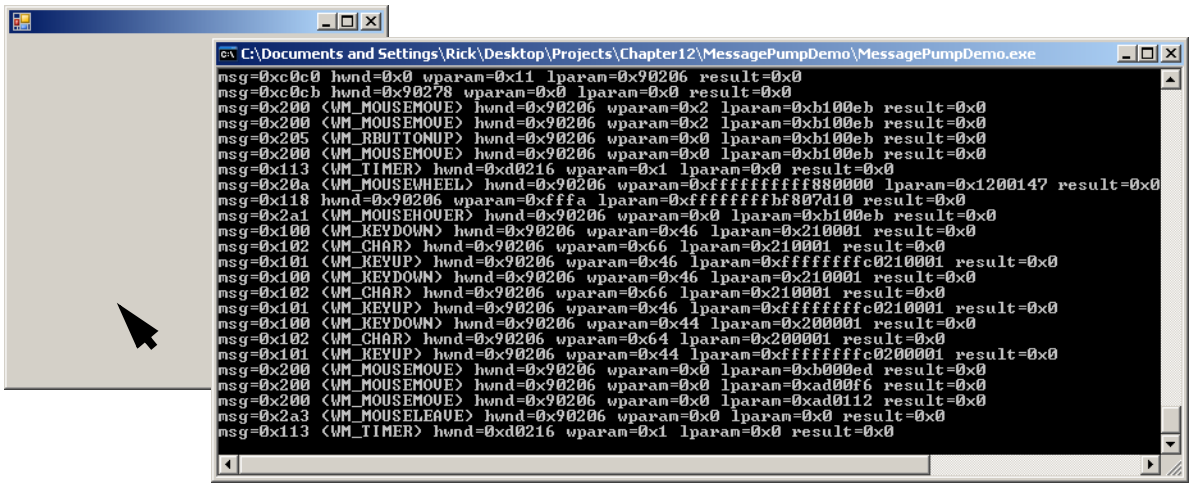

Figure 12-5: Results of Running Example 12.2

Referring to figure 12-5 — Cursor movement within the application window, not the console, causes the generation of WM_MOUSEMOVE messages. Note that since the window has no additional components like buttons or text boxes, almost all the messages generated belong to the WM (general window) category. Scrolling the mouse wheel causes a WM_MOUSEWHEEL message. Keystrokes cause a sequence of messages including WM_KEYDOWN, WM_CHAR, and WM_KEYUP. The best way to see these messages in action is to run this application and experiment with different types of mouse and keyboard entry along with window movement and resizing.

Final Thoughts On Messages

The information presented in this section falls into the category of "nice to know". Unless you're writing complex GUI applications that need to filter system messages you can safely ignore them. What you'll most likely do is create windows that contain various components, like text boxes, buttons, labels, menus, etc. These components, and indeed, forms as well, can respond to certain events. For example, buttons have (among others) the Click event. If you want a button to do something in response to a mouse click on it, you will need to create what is referred to as an event handler method and register it with the button's Click event. When the button is clicked, its event handler method, or methods if there is more than one, is called.

So, although the system is creating, sending, and responding to messages, you will think in terms of components and the events they can respond to. I will show you how to do this shortly, but first, I want to show you a few things about screen coordinates.

Quick Review

Microsoft Windows applications are *event-driven*. When launched, they wait patiently for an event to occur such as a mouse click or keystroke. Events are delivered to the application in the form of *messages*. Messages can be generated by the operating system in response to various types of stimuli, including direct user interaction (i.e., mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system generated messages are placed into a data structure referred to as the *system message queue*. A queue is a data structure that has a FIFO characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window.

Screen And Window (Client) Coordinate System

When working with GUIs you'll need to be aware to two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as *client coordinates*.

A window is drawn upon a computer screen at a certain position. The placement of the window's upper left corner falls on a certain point within the screen's coordinate system. The basic unit of measure for a screen is the *pixel*. The screen coordinate system is illustrated in figure 12-6.

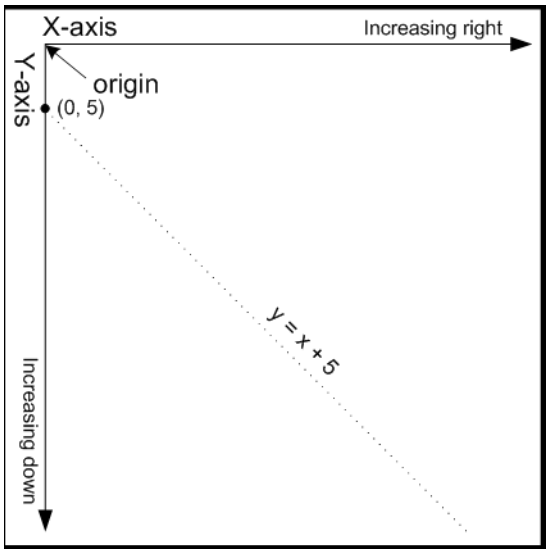

Figure 12-6: Screen Coordinate System

Referring to figure 12-6 — The origin of the screen, or the point where the value of both its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) coordinate pairs.

Windows have a coordinate system similar to screen coordinates as is shown in figure 12-7.

Referring to figure 12-7 — The window is placed on the screen at position (100, 275). This is the location of its upper left corner. The *origin* of the window is its upper left corner, meaning that components drawn within the window are placed with respect to the window's origin. The button drawn in the window is placed at position (125, 125).

Windows, and the components drawn within them, have *height* and *width*. The bounds of a component are the location of its upper left corner together with its width and height. If a window is placed at position (100, 275) and is 600 pixels wide and 350 pixels high, then its bounds are (100, 275, 600, 350). Example 12.3 gives a short program that prints the bounds of a window in response to user input.

12.3 ShowBounds.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class ShowBounds : Form, IMessageFilter {
5
6     public bool PreFilterMessage(ref Message m){
7       Console.WriteLine(this.Bounds);
8       return false;
9     }
10
11    public static void Main(){
```


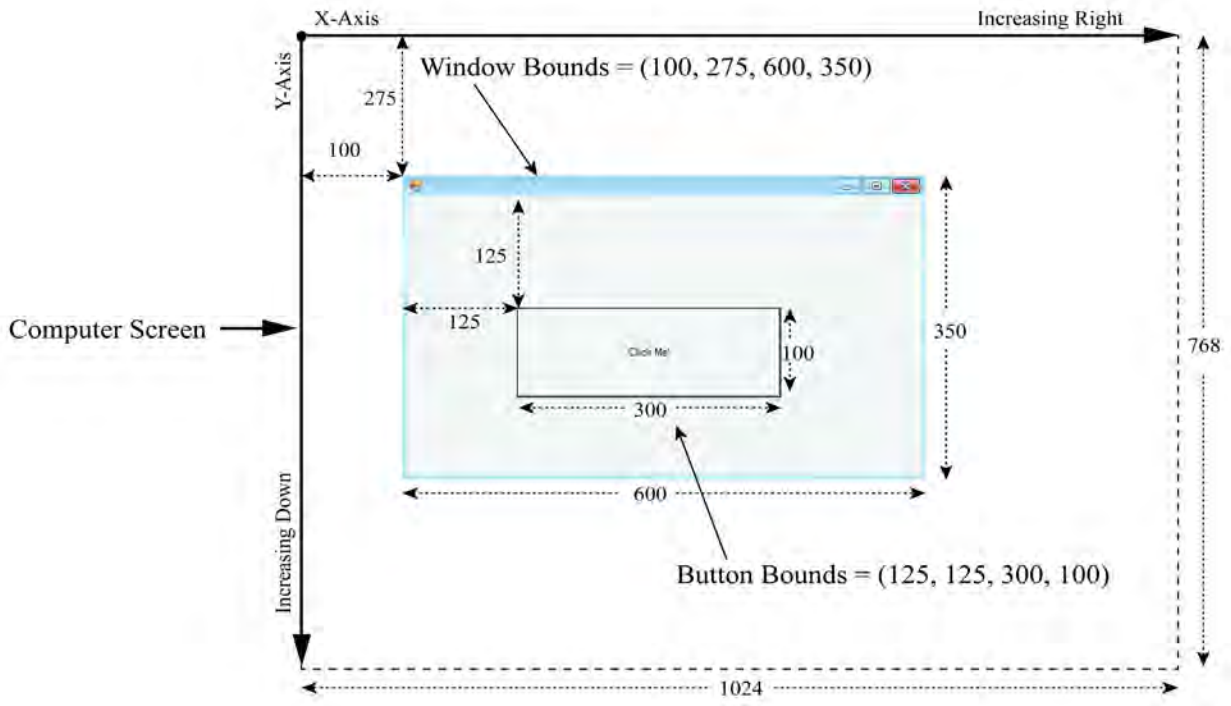

Figure 12-7: Window Coordinates

```
12      ShowBounds sb = new ShowBounds();
13      Application.AddMessageFilter(sb);
14      Application.Run(sb);
15    }
16  }
```

Referring to example 12.3 — This program is just a slight modification to the MessagePumpDemo program. The ShowBounds class extends Form and implements the IMessageFilter interface. The PreFilterMessage() method has been modified to print the window's Bounds property. Figure 12-8 shows the results of running this program.

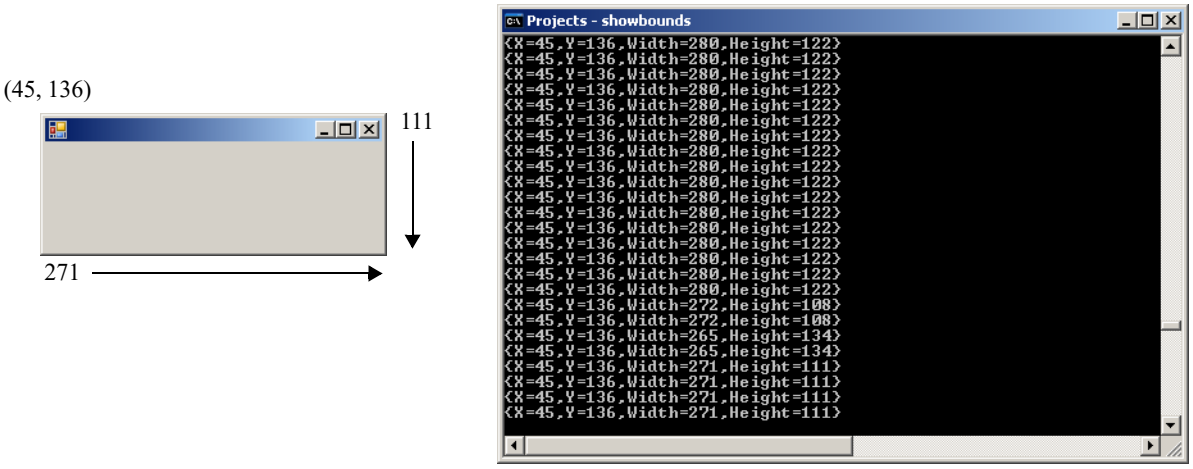

Figure 12-8: Results of Running Example 12.3

Referring to figure 12-8 — The output shown is the result of dragging the window through various sizes. Its final screen position is (45, 136); its final width is 271 pixels wide, and it is 111 pixels high. Thus, the bounds of this particular window are (45, 136, 271, 111), as is shown in the console window's final lines of output.

Quick Review

There are two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as *client coordinates*. The basic unit of measure upon a screen is the *pixel*. The origin of the screen, or the point where the value of both its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) coordinate pairs. Windows have a coordinate system similar to the screen, with their *origin* located in the upper left corner of the window. Windows, and the components drawn within them, have *height* and *width*. The *bounds* of a component are the location of its upper left corner together with its width and height.

Manipulating Form Properties

The Form class provides many properties, methods, and events that make it easy to manipulate them in your programs. In fact, as you saw in figure 12-1, the Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

In this section I'd like to demonstrate a few helpful form properties. Before going on though, I'd like to say that there are way too many Form class members to demonstrate them all in one section, or even one chapter. I recommend you take the time now, if you haven't already done so, to review the Form class documentation on the MSDN website and get a feel for all the things you can do to a form. I will demonstrate the use of other Form members when their use becomes appropriate in the text.

When you display a window, it's usually nice to give it a title. You can set a window's title bar text via its *Text* property. If you want to change a window's background color set its *BackColor* property. If you want to set a window's background image, do so via its *BackgroundImage* property.

The use of each of these properties requires the help of additional .NET Framework classes or structures including System.Drawing.*Color*, System.Drawing.*Image*, and System.Drawing.*BitMap*. Example 12.4 gives a short program that shows how to set a window's title bar text, backgound color, and its background image.

12.4 FormPropertiesDemo.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class FormPropertiesDemo : Form {
6     public static void Main(String[] args){
7       FormPropertiesDemo fpd = new FormPropertiesDemo();
8       if(args.Length > 0){
9         try{
10          Image image = new Bitmap(args[0]);
11          fpd.BackgroundImage = image;
12          fpd.Size = image.Size;
13        }catch(Exception){
14          //ignore for now
15        }
16      }else{
17        fpd.BackColor = Color.Black;
18      }
19
20      fpd.Text = "Form Properties Demo";
21      Application.Run(fpd);
22
23    } // end Main()
```



```
24  } // end class
```

Referring to example 12.4 — The FormPropertiesDemo class extends Form. Notice that I have used the String array version of the Main() method. This program can be run two ways: 1) by providing the name of an image file to use as the window background image, or 2) with no command line input, in which case the window's background color is set to black.

The code that creates the image and sets the window's BackgroundImage property is enclosed in a `try/catch` block that ignores the generated exception. If an exception does occur, the window is displayed with its default background color and no image. It would be easy, however, to add some code to the body of the *catch* clause that sets the background image to some default image.

Notice on line 10 that the image is created with the help of the Bitmap class. The Bitmap class has many overloaded constructor methods that make it easy to create images from different sources. The version used here creates an image from a string that represents the image filename. The string can be just the name of the file, in which case the program expects to find the image file in the default or working directory. (*i.e.*, The directory in which it executes.) The string can also be a complete path name. Notice on line 12 that the size of the window is set to the size of the image.

If the program is run with no command-line argument or by being double-clicked, then its background color is set to black with the help of the Color structure on line 17. The Color structure defines many public properties that represent different colors. In this example, I used Color.Black to set the window's BackColor property.

Lastly, I set the window's title bar text on line 20 via its Text property. I then display the form and kick off the GUI application execution thread with the Application.Run() method.

Figure 12-9 shows the results of running this program via the command line given the name of an image file. (You can download this image, WCC_2.jpg, from the PulpFreePress.com or Warrenworks.com websites, or you can use one of your own images.)

Figure 12-9: Running Example 12.4 via the Command Line with the Name of the Image WCC_2.jpg

Figure 12-10 shows the results of running example 12.4 with no image name or by double-clicking the executable file.

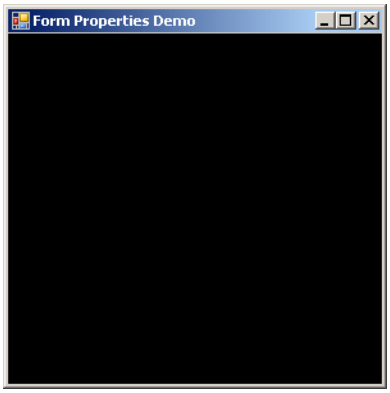

Figure 12-10: Running Example 12.4 with No Image

Quick Review

The Form class provides many properties, methods, and events which make it easy to manipulate them in your programs. The Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

Setting a control's properties often requires the use of other classes, structures, or enumerations found in the .NET Framework. A few of these include System.Drawing.*Point*, System.Drawing.*Rectangle*, System.Drawing.*Color,* System.Drawing.*Bitmap*, and System.Drawing.*Image*. The type of property determines what type of object you must use to set the property.

Adding Components To Windows: Button, TextBox, And Label

In this section I show you how to add components to windows. The components I use to explain the concepts include Button, TextBox, and Label. You'll find these, and many other components, in the System.Windows.Forms namespace. You'll also need the System.Drawing.*Point* structure to help place components in absolute positions within the window. Study the code given in example 12.5.

12.5 ComponentDemo.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5
6   public class ComponentDemo : Form {
7     private Button _button1;
8     private TextBox _textbox1;
9     private Label _label1;
10
11    public ComponentDemo(int x, int y, int width, int height){
12      this.Bounds = new Rectangle(x, y, width, height);
13      this.Text = "Component Demo";
14      InitializeComponents();
15    }
16
17    public ComponentDemo():this(100, 200, 400, 200){ }
18
19    private void InitializeComponents(){
20      _label1 = new Label();
```



```
21      _label1.Text = "This is a Label!";
22      _label1.Location = new Point(25, 25);
23
24      _button1 = new Button();
25      _button1.Text = "Click Me!";
26      _button1.Location = new Point(125, 25);
27
28      _textbox1 = new TextBox();
29      _textbox1.Text = "some default text";
30      _textbox1.Location = new Point(225, 25);
31
32      this.Controls.Add(_label1);
33      this.Controls.Add(_button1);
34      this.Controls.Add(_textbox1);
35    }
36
37    public static void Main(){
38      Application.Run(new ComponentDemo());
39    } // end Main()
40  } // end class
```

Referring to example 12.5 — The ComponentDemo class extends Form and declares three private component members: *_button1*, *_textbox1*, and *_label1*. It declares two constructors. The first constructor, beginning on line 11, declares four parameters that are used to set the window's Bounds property. Notice that the Bounds property must be set with the help of a *Rectangle* structure. Alternatively, you could set the window's Location, Height, and Width properties separately. On line 13, the window's title bar text is set via its Text property. And lastly, on line 14, the InitializeComponents() method is called.

The InitializeComponents() method begins on line 19 and creates and initializes the window's _button1, _textbox1, and _label1 components. Notice how each component's Location property is set with the help of the Point structure.

On lines 32 through 34, each component is added to the window by adding it to the window's *Controls* collection. Figure 12-11 shows the results of running this program.

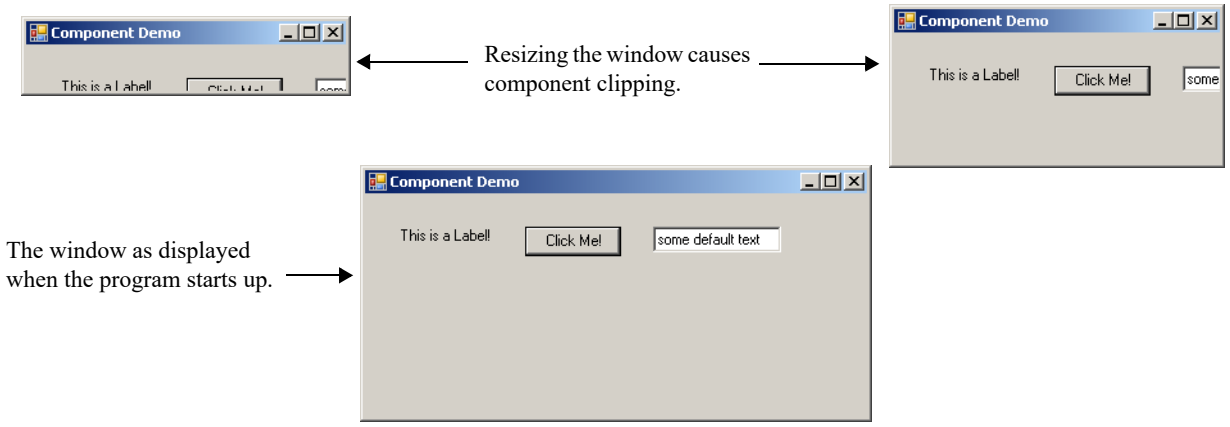

Figure 12-11: Results of Running Example 12.5

Referring to figure 12-11 — Notice the effects of setting the Text property for each component. Also note how the absolute placement of the components affects the window's appearance when it's resized. It's tedious to place components in specific positions within a window. If you're not careful, you can cover one component with another and wonder where it disappeared to. In a moment, I'll show you how to use layout panels to automatically place components within a window. But first, I want to show you how to make the button actually do something when it's clicked.

Quick Review

To add a control like a Button or TextBox to a window, you must first declare and create the control, set its properties, and then add the control to the window's Controls collection. The absolute placement of controls can be tedious. Use the System.Drawing.Rectangle structure to set a control's Bounds property. You may alternatively set a control's Top, Left, Width, and Height properties separately.

Registering Event Handlers With GUI Components

The whole point of creating a GUI is to have it respond to user interaction. As it stands now, the program shown in example 12.5 simply displays a window with a label, a button, and a text box. Although you can type in the text box and click the button, nothing else happens. Let's change that by adding an event handler method and registering it with the button's Click event.

Delegates And Events

All System.Windows.Forms GUI controls have event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event (and also via its MouseClick event!) GUI components can respond to many types of events. Table 12-2 offers an incomplete listing of some common events associated with the Control class, from which most Windows forms components inherit.

Event	Description	Delegate Type
BackColorChanged	Occurs when the BackColor property changes	System.EventHandler
BackgroundImageChanged	Occurs when the BackgroundImage property changes	System.EventHandler
Click	Occurs when control is clicked.†	System.EventHandler
DoubleClick	Occurs when control is double clicked	System.EventHandler
GotFocus	Occurs when the control receives focus	System.EventHandler
MouseClick	Occurs when the control is clicked by the mouse††	System.Windows.Forms.MouseEventHandler
MouseDoubleClick	Occurs when the control is double-clicked by the mouse	System.Windows.Forms.MouseEventHandler
MouseDown	Occurs when the mouse pointer is over the control and the mouse button is pressed	System.Windows.Forms.MouseEventHandler
MouseEnter	Occurs when the mouse pointer enters the control	System.EventHandler
MouseLeave	Occurs when the mouse pointer leaves the control	System.EventHandler

Table 12-2: Partial Listing of Control Events

Event	Description	Delegate Type
MouseMove	Occurs when the mouse pointer moves over the control	System.Windows.Forms.MouseEventHandler
MouseUp	Occurs when the mouse pointer is over the control and the mouse button is released	System.Windows.Forms.MouseEventHandler
Paint	Occurs when the control is redrawn	System.Windows.Forms.PaintEventHandler
† A Click event can be caused by pressing the Enter key †† A MouseClick is of type MouseEventHandler which uses MouseEventArgs to convey additional mouse information to the event handler method		

Table 12-2: Partial Listing of Control Events

The important thing to note about the events listed in table 12-2 is that different types of events are handled by different types of event handlers. An event handler type is defined or specified by a *delegate* type. A delegate provides a specification for a method signature. For example, the System.EventHandler delegate specifies a method with the following signature:

```
void EventHandler(Object sender, EventArgs e)
```

This means that if you want to register a method to respond to Click events on a control, the method must have the same signature as the event's delegate type. The best way to see all this work is to look at some code. Example 12.6 expands on the previous example by adding an event handler method for the _button1 component. When the button is clicked the text appearing in the text box is used to set the label's text.

12.6 ComponentDemo.cs (Mod 1)

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5
6   public class ComponentDemo : Form {
7     private Button _button1;
8     private TextBox _textbox1;
9     private Label _label1;
10
11    public ComponentDemo(int x, int y, int width, int height){
12      this.Bounds = new Rectangle(x, y, width, height);
13      this.Text = "Component Demo";
14      InitializeComponents();
15    }
16
17    public ComponentDemo():this(100, 200, 400, 200){ }
18
19    private void InitializeComponents(){
20      _label1 = new Label();
21      _label1.Text = "This is a Label!";
22      _label1.Location = new Point(25, 25);
23
24      _button1 = new Button();
25      _button1.Text = "Click Me!";
26      _button1.Location = new Point(125, 25);
```



```
27      _button1.Click += ButtonClickHandler;
28
29      _textbox1 = new TextBox();
30      _textbox1.Text = "some default text";
31      _textbox1.Location = new Point(225, 25);
32
33      this.Controls.Add(_label1);
34      this.Controls.Add(_button1);
35      this.Controls.Add(_textbox1);
36    }
37
38
39    public void ButtonClickHandler(Object sender, EventArgs e){
40      _label1.Text = _textbox1.Text;
41    }
42
43
44    public static void Main(){
45      Application.Run(new ComponentDemo());
46    } // end Main()
47  } // end class
```

Referring to example 12.6 — I made only two minor modifications to the previous program. The first addition appears on line 27 where an event handler is registered with the _button1.Click event. Notice the use of the '+=' operator. Note that Click events require event handler methods with a signature of the System.EventHandler delegate type. The ButtonClickHandler() method, defined starting on line 39, conforms to the required signature, namely, it takes two parameters — one of type Object and the other of type EventArgs. (You could use the lower-case object if you want.) The name of the method and its parameters can be just about anything you can think of but I recommend keeping the parameter names the same and choose a method name that makes it easy to identify it as an event handler method. In this example I chose the name ButtonClickHandler.

Notice in the body of the ButtonClickHandler() method how the text box text is assigned to the label text. Let's see this program in action. Figure 12-12 gives the results of running the code.

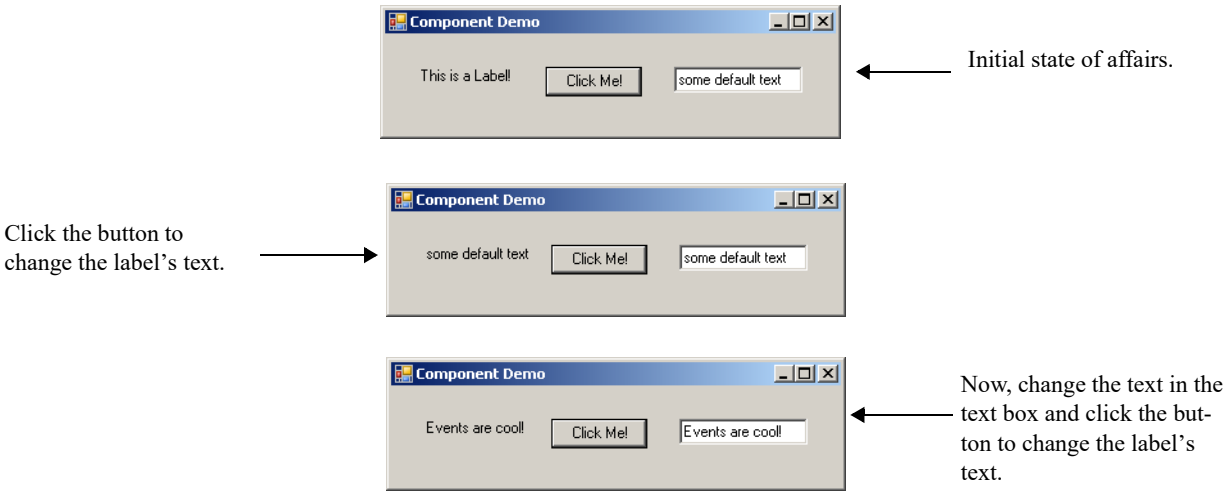

Figure 12-12: Results of Running Example 12.6 with Different Text in the TextBox

Referring to figure 12-12 — The top image shows the state of affairs when the program first executes. When you click the button the label text changes to say "some default text", which is the text initially loaded into the text box. In the bottom image the text "Events are cool!" is entered into the text box, the button clicked, and the label's text changed again.

Quick Review

The whole point of creating a GUI is to have it respond to user interaction. All System.Windows.Forms GUI controls have event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event.

A *delegate* declares a new type in the form of a method signature. Events are class members declared to have a certain delegate type, meaning that a method assigned to handle that event must have the specified delegate's method signature.

Use the '+=' operator to assign an event handler method to a control's event. Give your event handler method's names that clarify their role as event handlers.

Handling GUI Component Events In Separate Objects

In this section I am going to teach you a most critical skill, one that will liberate your thinking and take your programming skills to new heights. I'm going to show you how to handle GUI component events in separate objects. To do this you'll need to know how to do the following things:

- Create a stand-alone, non-application GUI class that extends Form.
- Create a separate application class that uses the services of the GUI class. This application class will also contain the required event handler code.
- Create GUI class constructors that take a reference to the object that contains the event handler code.
- Register a component's event with the event handler code via the supplied reference.
- Create appropriate methods or properties in the GUI class that allow horizontal manipulation of private GUI components.

Figure 12-13 gives the UML class diagram for the sample program I'm going to use to show you how to do these things.

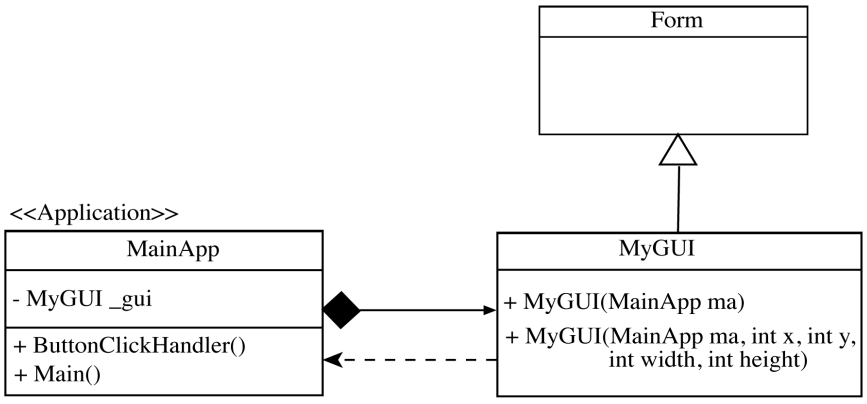

Figure 12-13: UML Class Diagram Showing Separate GUI and Application/event handler Classes

Referring to figure 12-13 — The MyGUI class extends Form. I've shown the constructors for the MyGUI class. Note that one of the parameters in each constructor is of type MainApp. That's why there's a dependency between class MyGUI and MainApp. It's through this parameter that the MainApp class passes an instance of itself when it creates an instance of MyGUI. The MyGUI class then uses the MainApp reference to access the MainApp.ButtonClickHandler() method.

The MainApp class creates an instance of MyGUI in its constructor method, which you'll see in example 12.7 shortly. That's why there's a black diamond adorning the MainApp class. This two-way dependency is not an ideal situation by any means, but it could be broken with the use of an interface.

The MainApp class pulls double duty in this example. It contains the Main() method and some event handler code in the form of the ButtonClickHandler() method.

Let's now take a look at the code for these two classes. Example 12-7 gives the code for the MyGUI class.

12.7 MyGUI.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5
6   public class MyGUI : Form {
7
8     /** Private GUI Components **/
9     private Button _button1;
10    private TextBox _textbox1;
11    private Label _label1;
12
13    /** Public Properties **/
14    public string TextBoxText {
15      get { return _textbox1.Text; }
16      set { _textbox1.Text = value; }
17    }
18
19    public string LabelText {
20      get { return _label1.Text; }
21      set { _label1.Text = value; }
22    }
23
24    /** Constructors **/
25    public MyGUI(MainApp ma, int x, int y, int width, int height){
26      this.Bounds = new Rectangle(x, y, width, height);
27      this.Text = "MyGUI Window";
28      InitializeComponents(ma);
29    }
30
31    public MyGUI(MainApp ma):this(ma, 100, 200, 400, 200){ }
32
33    /** Other Methods **/
34    private void InitializeComponents(MainApp ma){
35      _label1 = new Label();
36      _label1.Text = "This is a Label!";
37      _label1.Location = new Point(25, 25);
38
39      _button1 = new Button();
40      _button1.Text = "Click Me!";
41      _button1.Location = new Point(125, 25);
42      _button1.Click += ma.ButtonClickHandler;
43
44      _textbox1 = new TextBox();
45      _textbox1.Text = "some default text";
46      _textbox1.Location = new Point(225, 25);
47
48      this.Controls.Add(_label1);
49      this.Controls.Add(_button1);
50      this.Controls.Add(_textbox1);
51    }
52  } // end MyGUI class definition
```


Referring to example 12.7 — The MyGUI class is very similar in structure to example 12.6. The primary difference is that it's no longer an application because I removed the Main() method. I have removed the ButtonClickHandler() method and moved it to the MainApp class. I have also added two public properties named *TextBoxText* and *LabelText* that allow access to the Text properties of the private _textbox1 and _label1 components. I also modified the two constructors by adding a MainApp type parameter, and have added a parameter of type MainApp to the InitializeComponents() method as well. Before I walk you through the operation of this code, take a look at the MainApp code given in example 12.8.

12.8 MainApp.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  public class MainApp {
5
6    private MyGUI _gui;
7
8    public MainApp(){
9      _gui = new MyGUI(this);
10   }
11
12   public void ButtonClickHandler(Object sender, EventArgs e){
13       _gui.LabelText = _gui.TextBoxText;
14   }
15
16   public static void Main(){
17     MainApp ma = new MainApp();
18     Application.Run(ma._gui);
19   } // end Main()
20 } // end MyApp class definition
```

Referring to example 12.8 — The MainApp class has a private field of type MyGUI named *_gui*. The MainApp constructor creates an instance of MyGUI and passes to its constructor a reference to itself via the `this` keyword. The Main() method creates an instance of MainApp and then displays the window with a call to the Application.Run() method passing in the `_gui` reference.

The ButtonClickHandler() method shown on line 12 manipulates the text box and label text via the _gui reference by accessing the two public properties named TextBoxText and LabelText. Note that if you tried to access the MyGUI's _textbox1 and _label1 components directly, you'd get a compiler error because they are private fields, hence the need for public methods or properties to perform the required manipulations.

Let's now walk though the creation of the MyGUI object. When the MainApp constructor calls the MyGUI constructor, it passes in a reference to itself (*i.e.*, a reference to the object that's currently being created) via the `this` pointer. The single-parameter MyGUI constructor takes the reference and passes it on to the five parameter version of the MyGUI constructor. The `x`, `y`, `width`, and `height` parameters set the window's bounds. The `ma` parameter is passed as an argument to the InitializeComponents() method, where it is used to register its ButtonClickHandler() method with the _button1 Click event.

This is some of the most complex code you've encountered so far in this book. And although at this point it may seem difficult to trace through its execution, keep at it until you understand exactly what's happening in the code. Handling GUI events in separate objects is truly a critical programming skill and is a stepping stone to understanding and applying more complex object-oriented programming patterns in your code.

Now, let's see this bad boy run! Figure 12-14 shows the results of running this program.

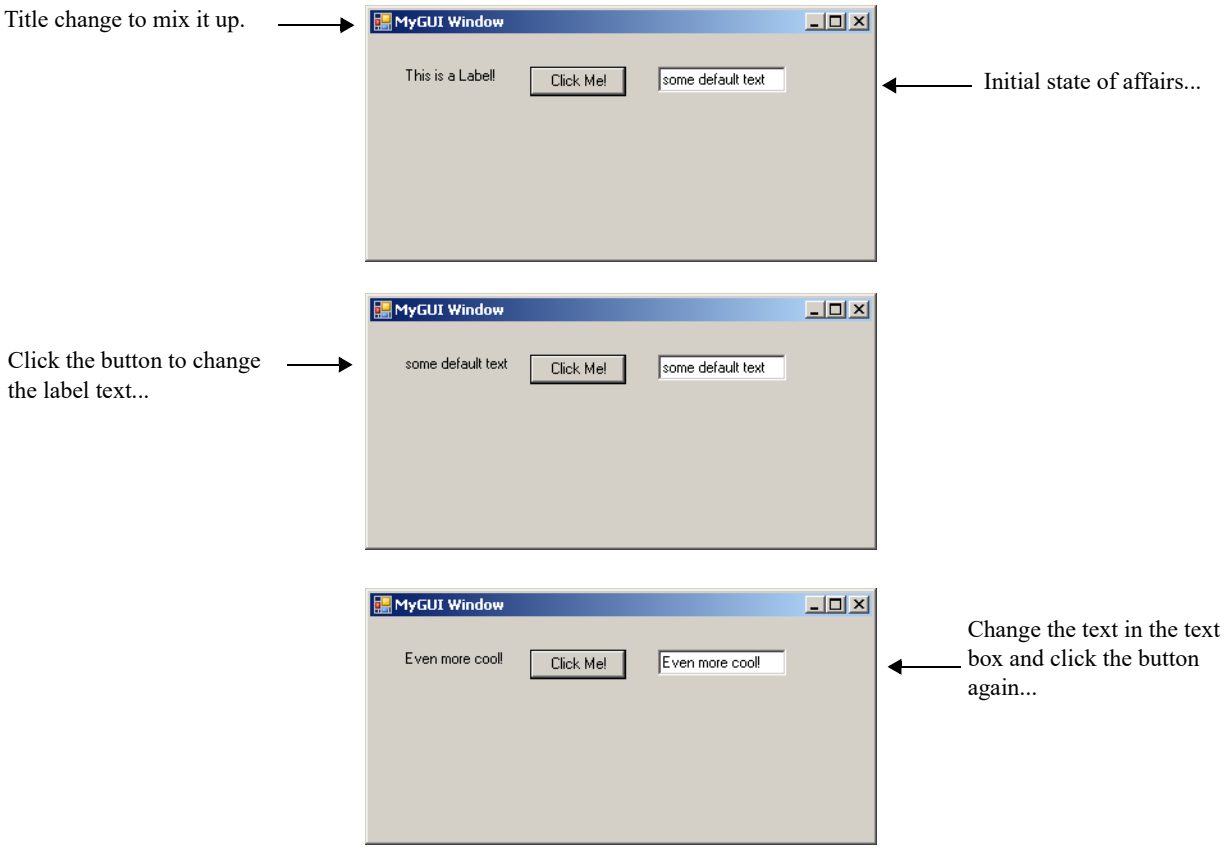

Figure 12-14: Results of Running Example 12.8 — GUI Events Handled in Separate Object

Quick Review

Knowing how to write code so that GUI events generated in one object are handled by event handler methods located in another object is a critical programming skill. To do this you must know how to do the following things: 1) create a stand-alone, non-application GUI class that extends Form, 2) create a separate application class that uses the services of the GUI class; this application class will also contain the required event handler code, 3) create GUI class constructors that take a reference to the object that contains the event handler code, 4) register a component's event with the event handler code via the supplied reference, and 5) create appropriate methods or properties in the GUI class that allow horizontal manipulation of private GUI components.

Layout Managers

As you saw earlier, placing GUI components in a window at absolute positions is tedious at best. While there may be times when you really want to put a component in a particular spot and have it stay there, it's generally preferable to make the window's components adjust their positions automatically to accommodate window resizing and prevent component clipping or hiding. In this section I will show you how to automatically place components on a window via layout panels.

The .NET Framework provides two layout panels: *FlowLayoutPanel* and *TableLayoutPanel*. This may not seem like much, but you really can create complex window layouts using only these two layout panels.

FlowLayoutPanel

The purpose of the FlowLayoutPanel is to dynamically lay out its components either horizontally or vertically. When you resize a window that contains components in a FlowLayoutPanel, those components will float within the panel and readjust their position based on the size of the window. Let's see a FlowLayoutPanel in action.

Examples 12.9 and 12.10 give the code for a program that displays five buttons in a window. The buttons are placed in a FlowLayoutPanel. To make the program more interesting, I have added the capability for each button to display the time it was clicked relative to program launch.

12.9 FlowLayoutGUI.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class FlowLayoutGUI : Form {
6
7     private Button[] _buttonArray;
8     private FlowLayoutPanel _panel;
9
10    public FlowLayoutGUI(MainApp ma, int x, int y, int width, int height){
11      this.Bounds = new Rectangle(x, y, width, height);
12      this.Text = "Flow Layout GUI";
13      InitializeComponents(ma);
14    }
15
16    public FlowLayoutGUI(MainApp ma):this(ma, 100, 200, 425, 150){ }
17
18    public void InitializeComponents(MainApp ma){
19      _panel = new FlowLayoutPanel();
20      _panel.SuspendLayout();
21      _panel.AutoSize = true;
22      _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
23      _panel.WrapContents = true;
24      _panel.Dock = DockStyle.Top;
25
26      _buttonArray = new Button[5];
27
28      for(int i=0; i<_buttonArray.Length; i++){
29        _buttonArray[i] = new Button();
30        _buttonArray[i].Text = "Button " + (i+1);
31        _buttonArray[i].Click += ma.ButtonClickHandler;
32        _panel.Controls.Add(_buttonArray[i]);
33      }
34
35      this.SuspendLayout();
36      this.Controls.Add(_panel);
37
38      _panel.ResumeLayout();
39      this.ResumeLayout();
40    }
41  } // end FlowLayoutGUI class definition
```

Referring to example 12.9 — The FlowLayoutGUI class declares an array of Buttons and one FlowLayoutPanel. Most of the action takes place in the InitializeComponents() method. First, the FlowLayoutPanel is created. Several of its properties are then set, including *AutoSize*, *AutoSizeMode*, *WrapContents*, and *Dock*. Note the call to _panel.SuspendLayout() on line 20. Call the SuspendLayout() method anytime you are modifying several control properties at a time or are adding multiple controls to a control. The SuspendLayout() method suspends the target control's layout logic for improved performance. A call to SuspendLayout() must eventually be followed by a call to ResumeLayout().

The Dock property gets or sets how a control's edges are docked to its containing control and determines how it is resized. Note the use of the DockStyle enumeration to specify which edge to dock. The complete set of DockStyle values include DockStyle.*Bottom*, DockStyle.*Fill*, DockStyle.*Left*, DockStyle.*None*, DockStyle.*Right*, and DockStyle.*Top* as I have used here.

(**Note:** To see the effects of DockStyle.Top you could add a line of code that sets the _panel.BackColor to something different that the color of the window. Then, when you resize the window, you'll see the FlowLayoutPanel react to the changes.)

The button array is created on line 26 followed by a `for` loop that creates and initializes each button. Note that each button is added to the FlowLayoutPanel's Controls array. The buttons will be "flowed" into the FlowLayoutPanel from left to right by default in the order in which the buttons are added. You can change this behavior by setting the panel's *FlowDirection* property with the help of the FlowDirection enumeration whose values include FlowDirection.*BottomUp*, FlowDirection.*LeftToRight*, FlowDirection.*RightToLeft*, and FlowDirection.*TopDown*.

After all the buttons have been added to the FlowLayoutPanel it's time to add the _panel reference to the window's Controls collection. This is done on line 36, and is preceded by a call to the window's SuspendLayout() method. Lastly, the ResumeLayout() method is called on both the _panel and the window.

Example 12.10 gives the code for the MainApp class.

12.10 MainApp.cs

```
1  using System;
2  using System.Windows.Forms;
3
4  public class MainApp {
5
6    private FlowLayoutGUI _gui;
7    private DateTime _appStart;
8
9    public MainApp(){
10     _gui = new FlowLayoutGUI(this);
11     _appStart = DateTime.Now;
12   }
13
14   public void ButtonClickHandler(Object sender, EventArgs e){
15       ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
16   }
17
18   public static void Main(){
19     MainApp ma = new MainApp();
20     Application.Run(ma._gui);
21   } // end Main()
22 } // end MyApp class definition
```

Referring to example 12.10 — The MainApp class declares a FlowLayoutGUI field named _gui and a DateTime field named _appStart. In the MainApp constructor, the _gui reference is initialized to an instance of the FlowLayoutGUI class. The _appStart reference is initialized to DateTime.*Now,* which is simply a DateTime structure that represents the current date and time. The _appStart is then used in the body of the ButtonClickHandler() method to calculate the time span between the time the application started and the time the button was clicked. Note how the sender parameter is used to figure out which control generated the event. The result of subtracting one DateTime object from another results in a *TimeSpan* object. Every time a button in the GUI is clicked, its text is updated with the elapsed time the application has been running. Figure 12-15 shows the results of running this program.

TableLayoutPanel

The TableLayoutPanel lets you divvy up a panel into cells arranged by rows and columns. The order in which controls are added to a TableLayoutPanel is, by default, left-to-right and top-to-bottom. For exam-

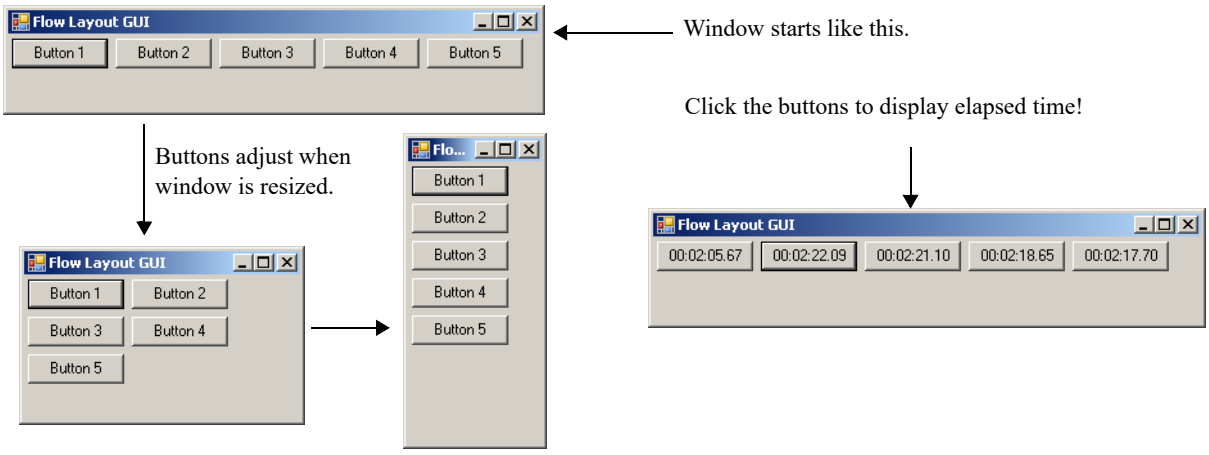

Figure 12-15: Results of Running Example 12.10 — Buttons Adjust when Window is Resized

ple, if you create a TableLayoutPanel with two rows that each contain two columns and you add four buttons to it, the first button will go into cell 0,0, the second onto cell 0,1, the third into cell 1,0, and the last into cell 1,1.

The following program adds a slew of buttons to a TableLayoutPanel. When each button is clicked, its BackColor property is changed along with its Image. The program consists of both examples 12.11 and 12.12.

12.11 TableLayoutGUI.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class TableLayoutGUI : Form {
6
7     Button[,] _floor = new Button[10,10];
8     TableLayoutPanel _panel;
9
10    public TableLayoutGUI(MainApp ma){
11      InitializeComponents(ma);
12    }
13
14    public void InitializeComponents(MainApp ma){
15      _panel = new TableLayoutPanel();
16      _panel.SuspendLayout();
17      _panel.ColumnCount = 10;
18      _panel.RowCount = 10;
19      _panel.Dock = DockStyle.Top;
20      _panel.AutoSize = true;
21      _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
22
23      for(int i = 0; i<_floor.GetLength(0); i++){
24        for(int j = 0; j<_floor.GetLength(1); j++){
25          _floor[i,j] = new Button();
26          _floor[i,j].Click += ma.MarkSpace;
27          _panel.Controls.Add(_floor[i,j]);
28        }
29      }
30
31      this.SuspendLayout();
32      this.Text = "TableLayoutGUI Window";
33      this.Width = 850;
34      this.Height = 325;
35      this.Controls.Add(_panel);
```



```
36      _panel.ResumeLayout();
37      this.ResumeLayout();
38    } // end InitializeComponents() method
39  } // end TableLayoutGUI class definition
```

Referring to example 12.11 — This program declares and creates a two-dimensional array of buttons having 10 rows and 10 columns. In the InitializeComponents() method, the TableLayoutPanel is created and its *RowCount* and *ColumnCount* properties are set to 10 x 10 to match the array's dimensions. The buttons are created in the nested `for` loop that starts on line 23, and added to the panel's Controls collection. Each button's Click event calls the MarkSpace() event handler method located in the MainApp class. Example 12.12 gives the code for the MainApp class.

12.12 MainApp.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class MainApp {
6
7     private TableLayoutGUI _gui;
8     private Bitmap _bitmap;
9     public MainApp(){
10      _gui = new TableLayoutGUI(this);
11      _bitmap = new Bitmap("rat.gif");
12    }
13
14    public void MarkSpace(Object sender, EventArgs e){
15      ((Button)sender).BackColor = Color.Blue;
16      ((Button)sender).Image = _bitmap;
17    }
18
19    public static void Main(){
20      MainApp ma = new MainApp();
21      Application.Run(ma._gui);
22    } // end Main()
23  } // end MainApp class definition
```

Referring to example 12.12 — The MainApp class declares a TableLayoutGUI field named _gui and a Bitmap field named _bitmap. The MainApp constructor initializes the _gui and _bitmap references. The image used in this example is named "rat.gif" and is expected to be in the program's execution directory. (**Note:** The rat.gif image can be downloaded from the PulpFreePress.com or Warrenworks.com websites, or you can use your own image.)

When a button is clicked in the TableLayoutGUI window, the MarkSpace() method sets its BackColor and Image properties. Figure 12-16 shows the results of running this program.

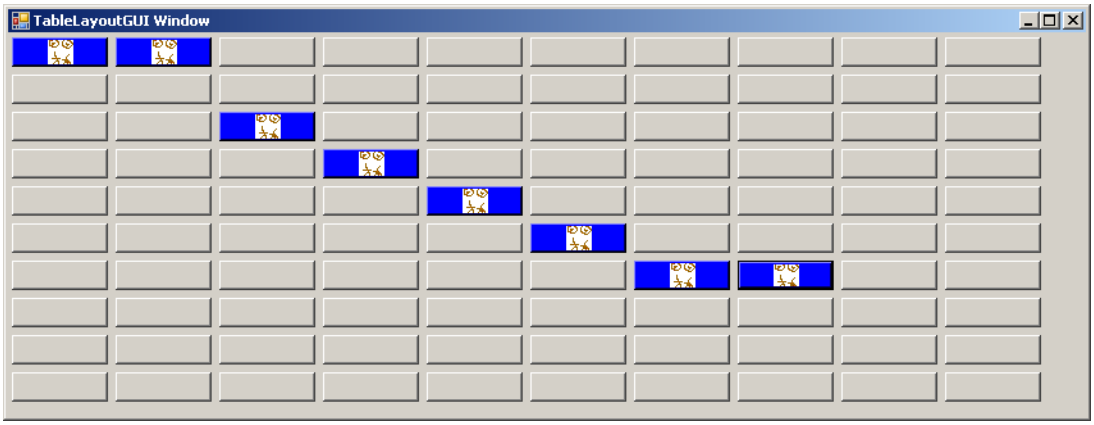

Figure 12-16: Results of Running Example 12.12 After Several Buttons have been Clicked

Quick Review

Use the FlowLayoutPanel and TableLayoutPanel classes to automatically control the layout of controls in a window. Controls placed in a FlowLayoutPanel flow into the panel from left-to-right by default. Don't forget to set the panel's Dock property.

The TableLayoutPanel places controls into a grid arrangement, where each control occupies a cell that can be accessed via x and y coordinates, much like a two-dimensional array.

You can add panels to panels to create complex GUIs.

Menus

Professional looking GUIs are usually controlled via menus. In this section, I am going to show you how to add menus to your GUIs.

The .NET Framework provides several classes that make adding menus easy. These include the System.Windows.Forms.*MenuStrip* and the System.Windows.Forms.*ToolStripMenuItem*. The example program used to demonstrate the operation of these classes allows the user to dynamically add buttons and text boxes to a window via a menu. The Add menu contains three menu items named Button, TextBox, and Exit. It also contains a menu item separator. Figure 12-17 shows the application's window and menu structure.

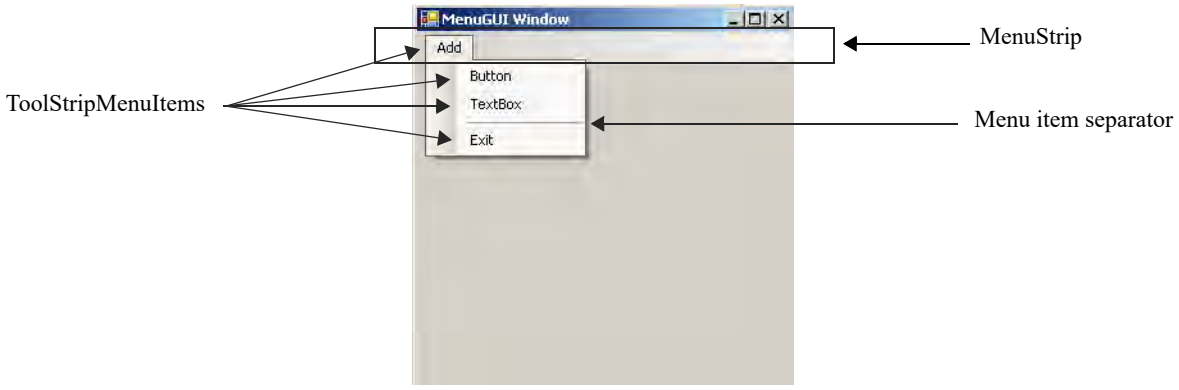

Figure 12-17: Window and Menu Structure of Menu Demo Program

Referring to figure 12-17 — The MenuStrip control is docked at the top of the MenuGUI window. The Add menu item is added to the MenuStrip, and the Button, TextBox, item separator, and Exit menu items are added as submenu items to the Add menu item. Example 12.13 gives the code for this window.

12.13 MenuGUI.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4
5   public class MenuGUI : Form {
6
7     private FlowLayoutPanel _panel;
8
9     public MenuGUI(MainApp ma, int x, int y, int width, int height){
10      this.Bounds = new Rectangle(x, y, width, height);
11      this.Text = "MenuGUI Window";
12      InitializeComponents(ma);
```



```
13   }
14
15   public MenuGUI(MainApp ma):this(ma, 125, 125, 300, 300){ }
16
17
18   private void InitializeComponents(MainApp ma){
19     MenuStrip ms = new MenuStrip();
20
21     ToolStripMenuItem addMenu = new ToolStripMenuItem("Add");
22     ToolStripMenuItem addButtonItem = new ToolStripMenuItem("Button", null,
23                                                             ma.AddButtonItemHandler);
24     ToolStripMenuItem addTextBoxItem = new ToolStripMenuItem("TextBox", null,
25                                                             ma.AddTextBoxItemHandler);
26     ToolStripMenuItem addExitItem = new ToolStripMenuItem("Exit", null,
27                                                             ma.AddExitItemHandler);
28
29     addMenu.DropDownItems.Add(addButtonItem);
30     addMenu.DropDownItems.Add(addTextBoxItem);
31     addMenu.DropDownItems.Add("-"); // <---- use a dash to add menu item separators
32     addMenu.DropDownItems.Add(addExitItem);
33
34     ms.Items.Add(addMenu);
35     ms.Dock = DockStyle.Top;
36     this.MainMenuStrip = ms;
37
38     _panel = new FlowLayoutPanel();
39     _panel.SuspendLayout();
40     _panel.AutoSize = true;
41     _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
42     _panel.WrapContents = true;
43     _panel.Dock = DockStyle.Fill;
44
45     this.SuspendLayout();
46     this.Controls.Add(_panel);
47     this.Controls.Add(ms); // IMPORTANT: Add the MenuStrip last!!
48     _panel.ResumeLayout();
49     this.ResumeLayout();
50   }
51
52   public void AddButton(MainApp ma){
53     Button b = new Button();
54     b.Text = "C# Rocks";
55     b.Click += ma.ButtonClickHandler;
56     _panel.SuspendLayout();
57     _panel.Controls.Add(b);
58     _panel.ResumeLayout();
59   }
60
61   public void AddTextBox(MainApp ma){
62     TextBox t = new TextBox();
63     t.Text = "Default Text";
64     t.Click += ma.TextBoxClickHandler;
65     _panel.SuspendLayout();
66     _panel.Controls.Add(t);
67     _panel.ResumeLayout();
68   }
69 } // end MenuGUI class definition
```

Referring to example 12.13 — The MenuGUI class declares one private FlowLayoutPanel field named _panel. All of the menu items are declared locally within the InitializeComponents() method. You may be asking yourself at this point why not declare all of the GUI components as fields. The answer depends on which components you need to have access to after the GUI is rendered. In this example, I am adding buttons and text boxes to the flow layout panel. If I were designing an application that needed to add menu items to the menu strip, then I would most likely declare a MenuStrip field for easy access.

Let's step through the InitializeComponents() method. The first thing I do on line 19 is to declare and create the MenuStrip. Next, on lines 21 through 27, I declare and create the four ToolStripMenuItems. Notice the naming convention I have adopted here to help sort out which sub-items belong to which parent menu item.

The ToolStripMenuItem constructor is overloaded. I have used two versions in this code. The first version used on line 21 takes the name of the menu item. The second type of constructor takes three arguments: *name*, *image*, and *event handler*. I've used "null" to indicate no image. Notice how, by supplying an event handler, menu items are enabled to perform work.

Next, on lines 29 through 32, I add the submenu items to the addMenu item by adding them with its DropDownItems.Add() method. Notice here how a menu item separator is added to a list of menu items by adding a dash "-".

Once the submenu items are added to the parent menu item, the parent menu item is added to the MenuStrip by calling its Items.Add() method. The MenuStrip is docked to the top of the window and then designated as the MenuStrip for this window.

Lines 38 through 43 prepare the FlowLayoutPanel by setting several of its properties. Then, on line 45 the window's layout is suspended with a call to SuspendLayout(), then the panel is added to the window first, followed by the MenuStrip. (**Note:** Always add the MenuStrip last to ensure it displays at the top of the form and does not hide other controls.)

Lastly, the ResumeLayout() method is called on both the panel and the window.

The MenuGUI class contains two other public methods named AddButton() and AddTextBox(). To see these methods in action, let's take a look at the MainApp class given in example 12.14.

12.14 MainApp.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class MainApp {
5
6     private MenuGUI _gui;
7     private DateTime _appStart;
8
9     public MainApp(){
10      _gui = new MenuGUI(this);
11      _appStart = DateTime.Now;
12    }
13
14    public void AddButtonItemHandler(object sender, EventArgs e){
15      _gui.AddButton(this);
16    }
17
18    public void AddTextBoxItemHandler(object sender, EventArgs e){
19      _gui.AddTextBox(this);
20    }
21
22    public void AddExitItemHandler(object sender, EventArgs e){
23      Application.Exit();
24    }
25
26    public void ButtonClickHandler(Object sender, EventArgs e){
27      ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
28    }
29
30    public void TextBoxClickHandler(Object sender, EventArgs e){
31      ((TextBox)sender).Text = (DateTime.Now - _appStart).ToString();
32    }
33
34    public static void Main(){
35      MainApp ma = new MainApp();
```



```
36      Application.Run(ma._gui);
37    }
38  }
```

Referring to example 12.14 — This version of the MainApp class declares two fields, one of type MenuGUI named _gui and the other of type DateTime named _appStart. Its constructor initializes the fields and kicks off the program with a call to Application.Run().

This class also contains five event handler methods, three of which are used by the menu items. The ButtonClickHandler() method is used by all of the buttons that get added to the window, and the TextBoxClickHandler() method is used by all the text boxes.

When this program runs, users can add as many buttons or text boxes as they want by selecting the appropriate menu item. A click on either a button or a text box results in its text being set to the elapsed program run time. Figure 12-18 shows this program in action after several buttons and textboxes have been added to the window and then clicked.

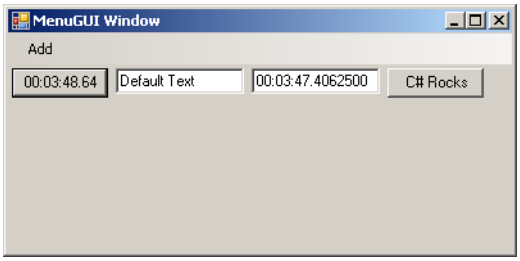

Figure 12-18: Results of Running Example 12.14 and Adding Several Buttons and Text Boxes

Quick Review

Use menus to give your GUI a professional appearance. You can create menus using the MenuStrip and ToolStripMenuItem classes.

A Little More About TextBoxes

Up until now, I've only used TextBox controls in single line mode ideally suited to display one short line of text. The TextBox, however, is a versatile control that can be used to edit multiline text or rich text content. In this section, I want to show you how to display a multi-line TextBox and then, by double-clicking on a line of text in the box, determine the text line number. You'll find this to be a handy little trick to have up your sleeve.

The code for the example program is given in two files. Example 12.15 gives the code for the LineSelectGUI class and example 12.16 gives the code for the MainApp class.

12.15 LineSelectGUI.cs

```
1   using System;
2   using System.Windows.Forms;
3   using System.Drawing;
4   using System.Windows;
5
6   public class LineSelectGUI : Form {
7
8     private TextBox _tb1;
9     private TextBox _tb2;
10    private FlowLayoutPanel _flowLayoutPanel;
11
12    public LineSelectGUI(MainApp ma, int x, int y, int width, int height){
```



```
13      this.Bounds = new Rectangle(x, y, width, height);
14      this.Text = "LineSelectGUI Window";
15      InitializeComponents(ma);
16    }
17
18    public LineSelectGUI(MainApp ma):this(ma, 125, 125, 375, 200){ }
19
20    public void InitializeComponents(MainApp ma){
21      _flowLayoutPanel = new FlowLayoutPanel();
22      _flowLayoutPanel.SuspendLayout();
23      _flowLayoutPanel.Height = 56;
24      _flowLayoutPanel.Width = 20;
25      _flowLayoutPanel.AutoSize = true;
26      _flowLayoutPanel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
27      _flowLayoutPanel.WrapContents = true;
28      _flowLayoutPanel.Dock = DockStyle.Top;
29
30      _tb1 = new TextBox();
31      _tb1.Multiline = true;
32      _tb1.Height = 150;
33      _tb1.Width = 200;
34      _tb1.DoubleClick += ma.DoubleClickHandler;
35
36      _tb2 = new TextBox();
37      _flowLayoutPanel.Controls.Add(_tb1);
38      _flowLayoutPanel.Controls.Add(_tb2);
39
40      this.SuspendLayout();
41      this.Controls.Add(_flowLayoutPanel);
42
43      _flowLayoutPanel.ResumeLayout();
44      this.ResumeLayout();
45    }
46
47    public void ShowLineNumber(){
48      int index = _tb1.SelectionStart;
49      int line_number = _tb1.GetLineFromCharIndex(index);
50      _tb2.Text = ("Line Number is: " + line_number);
51    }
52  } // end LineSelectGUI class definition
```

Referring to example 12.15 — The LineSelectGUI class declares two TextBox fields named _tb1 and _tb2 and one FlowLayoutPanel field named _flowLayoutPanel. In the InitializeComponents() method the FlowLayoutPanel is created and initialized. Next, the first TextBox field, _tb1, is created and initialized to become a multiline TextBox by setting its *MultiLine* property to true. Setting its *WrapContents* property to true makes text wrap automatically to the next line. Finally, on line 34, the MainApp.DoubleClickHandler() method is registered with _tb1's DoubleClick event.

The second TextBox is created on line 36, and on lines 37 and 38 both TextBoxes are added to the FlowLayoutPanel.

The LineSelectGUI class defines a method named ShowLineNumber() starting on line 47. The ShowLineNumber() method determines the text line number with the help of two TextBox methods. First, the index of the selected text must be determined by calling the TextBox.SelectionStart() method. The index of the selected text is then used in a call to the TextBox.GetLineFromCharIndex() method.

The ShowLineNumber() method is called within the body of the MainApp.DoubleClickHandler() method. Example 12.16 gives the code for the MainApp class.

12.16 MainApp.cs

```
1   using System;
2   using System.Windows.Forms;
3
4   public class MainApp {
```



```
5
6     private LineSelectGUI _gui;
7
8     public MainApp(){
9       _gui = new LineSelectGUI(this);
10    }
11
12    public void DoubleClickHandler(Object sender, EventArgs args){
13      _gui.ShowLineNumber();
14    }
15
16    public static void Main(){
17      MainApp ma = new MainApp();
18      Application.Run(ma._gui);
19    }
20  }
```

Referring to example 12.16 — The MainApp class declares one field of type LineSelectGUI named _gui. The MainApp constructor initializes the _gui field.

A double-click within the multiline text box results in a call to the DoubleClickHandler() method, which in turn calls the _gui.ShowLineNumber() method. Figure 12-19 shows the results of running this program.

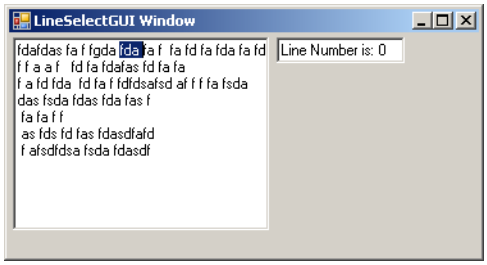

Figure 12-19: Results of Running Example 12.16 — Double-clicking the First Line

Quick Review

TextBoxes can be used in single-line or multiline mode. To create a multiline text box set its *Multiline* property to true. To determine the line number for a line of text in a text box, use the TextBox.SelectionStart() method to first get the index of the selected character, then use the TextBox.GetLineFromCharIndex() method to get the text line number.

The Rhythm Of Coding GUIs

You may have noticed by now that there is a certain rhythm associated with writing GUI code by hand. Regardless of how complex you make your GUI, getting into the rhythm can make writing the code much less tedious and lots of fun. The rhythm goes something like this:

- Start by sketching a layout of your GUI. This will be a tremendous help when you start coding.
- Declare components like dialog windows, panels, menus, buttons, text boxes, labels, etc.
- Initialize the components in a constructor or in another method that's called by the constructor.
- If using absolute positioning, set the component's placement within the window.
- Register event handlers.
- Set other component properties as required.

- Add components to panels.
- Add panels to a form.

Windows Presentation Foundation (WPF) – A Breathless Introduction

The Windows Presentation Foundation, or simply, the WPF, enables you to declaratively build GUI applications using a Microsoft variant of Extensible Application Markup Language (XAML) where you declare GUI components and their properties in .xaml files and the C# code that services UI events is contained within separate but related source files (.cs) referred to as code-behinds.

The WPF will force a few changes in the way I present the topic in this section as compared with Windows Forms programming discussed previously. First, I'll be using Visual Studio 2017 to develop, build, and run the examples. Trying to write WPF code without the help of an IDE is not easy nor recommended. Second, the basics of handling GUI events still apply, so I'm not going to rehash those topics. Finally, I'll be demonstrating some very basic concepts in this section, and the GUIs that result will be rustic at best. If the WPF is something you want to explore more deeply, please seek out a book or resource dedicated to the topic.

A Simple WPF Application

Using Visual Studio 2017 or newer, create a new **WPF App (.NET Framework)** project. For this example, I will name the solution *SimpleWPFApp*, and the solution's first project I'll name *PresentationLayer* as figure 12-20 illustrates.

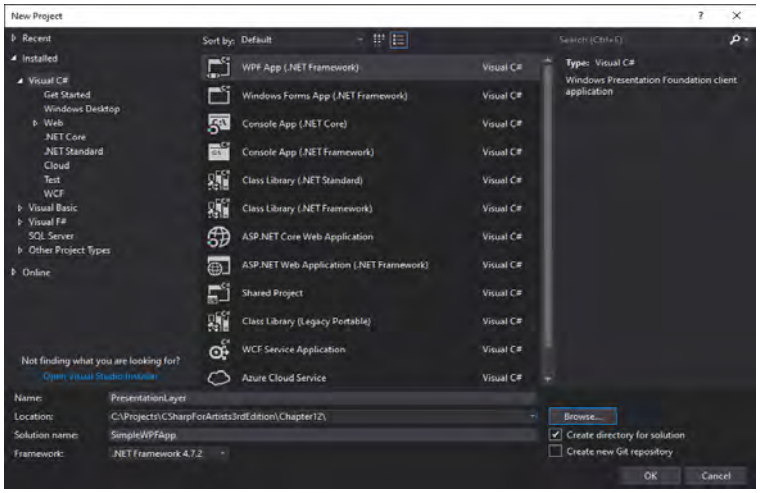

Figure 12-20: Creating WPF Project

Referring to figure 12-20 — This approach to solution and project naming simply reflects my preference for changing the default choices pre-populated by Visual Studio when you first create a project. It's good to remember that when starting a Visual Studio "project" from scratch, the project is contained within a "solution". A solution can contain one or more projects. I prefer to give a name to the solution that is distinct from all of its projects. Figure 12-21 shows what the Solution Explorer pane looks like when I've created the project.

Referring to figure 12-21 — Notice the PresentationLayer project is contained within the SimpleWPFApp solution. You can now add more projects to the solution as required while maintaining your sanity. Figure 12-22 shows the initial state of the project when first launched.

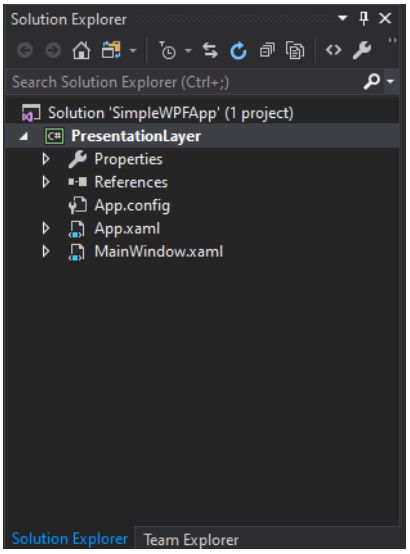

Figure 12-21: Solution and Project Layout in Solution Explorer

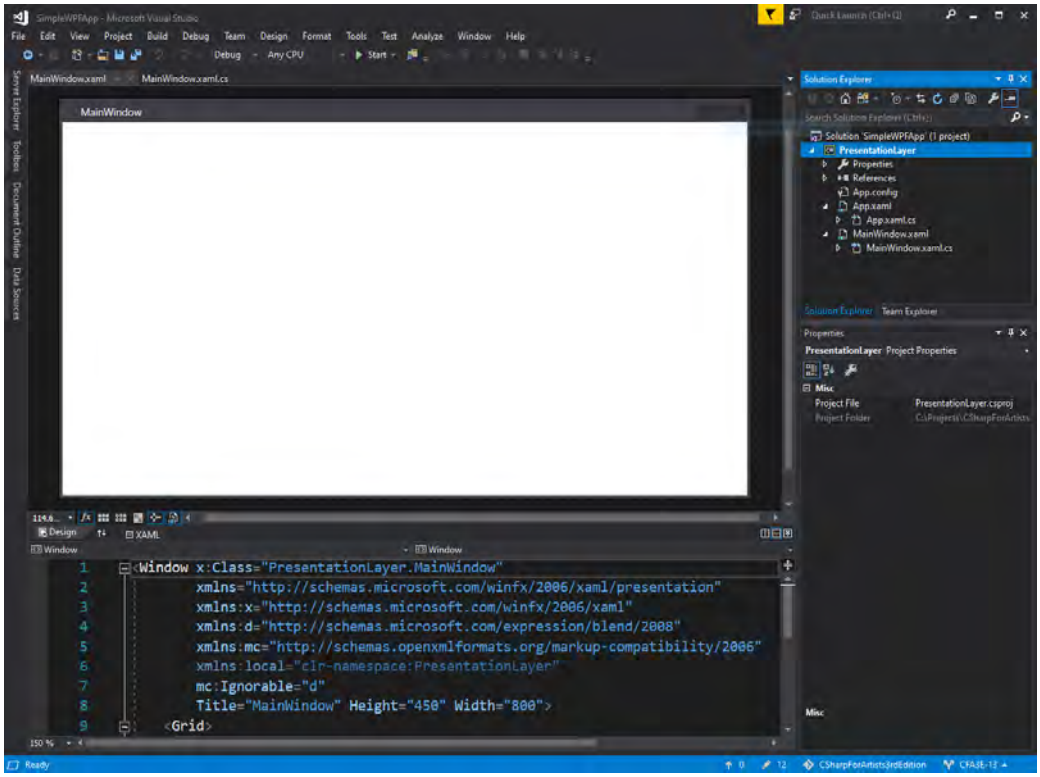

Figure 12-22: Initial Project Workspace

Referring to figure 12-22 — Visual Studio initializes the new project with several files that enable you to build and run the project right from the start, so let's do this. On the toolbar below the main menu click the green triangle **Start** button to the right of "Any CPU" to build and run the project. Figure 12-23 shows the MainWindow running in debug mode.

Referring to figure 12-23 — The black, rectangular toolbox shown top-center of the window is a set of debugging aids. The debugging toolbox is not present when the application is launched from the desktop.

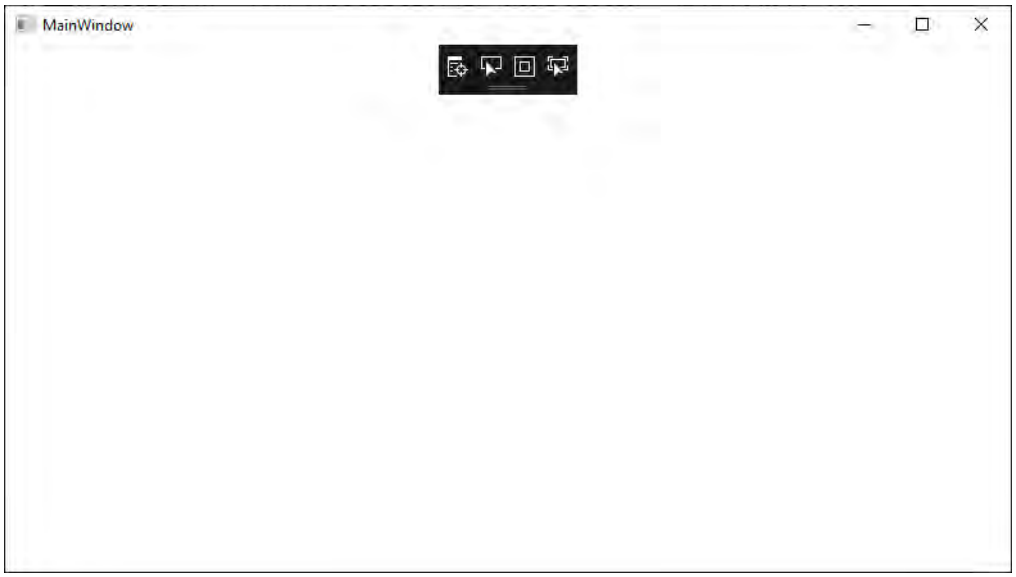

Figure 12-23: MainWindow Running in debug Mode

To stop the debugging session, close the window or click the orange rectangle located on the toolbar below the main Visual Studio menu. This returns you to the project workspace. We are now ready to dive deeper into how this simple WPF application works.

XAML & Code-Behind Structure

The new WPF project created above comes with a set of default XAML and code-behind file necessary to run the program and launch the MainWindow. These files include App.xaml and its code-behind App.xaml.cs, and MainWindow.xaml and its code-behind MainWindow.xaml.cs. Notice the naming pattern? A few words of warning before getting started. WPF is an abstraction layer over components located in the Systems.Windows namespace. To see what's really going on underneath the covers you'll need to drill down in the Visual Studio Solution Explorer pane. Also, when renaming or otherwise manipulating project files, always do so within Visual Studio. Directly manipulating project files at the file system level, like opening a project folder and renaming files directly, will break the project. You've been warned. Now, let's discuss each of these files in more detail, starting with App.xaml.

12.17 App.xaml

```
1  <Application x:Class="PresentationLayer.App"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:local="clr-namespace:PresentationLayer"
5               StartupUri="MainWindow.xaml">
6      <Application.Resources>
7
8      </Application.Resources>
9  </Application>
```

Referring to example 12.17 — On line 1, `Application` is of type System.Windows.Application, and the `x:Class` property is set to a class named `App`, which is a partial class, as you'll see below, that extends System.Windows.Application. The `<Application.Resources>` tag represents a collection (dictionary of key/value pairs) available for use by the application. I'll show you how to define and use an application resource here shortly.

While we're discussing the App.xaml file, select the App.xaml file in the Solution Explorer pane to display its File Properties as is shown in figure 12-24.

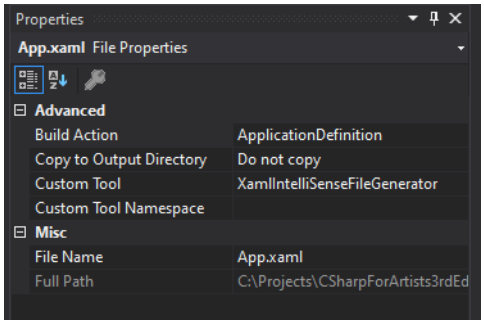

Figure 12-24: App.xaml File Properties

Referring to figure 12-24 — In the Advanced section, note that Build Action is set to ApplicationDefinition. This is the build action assigned to the XAML file that defines the application, which in this case, is App.xaml.

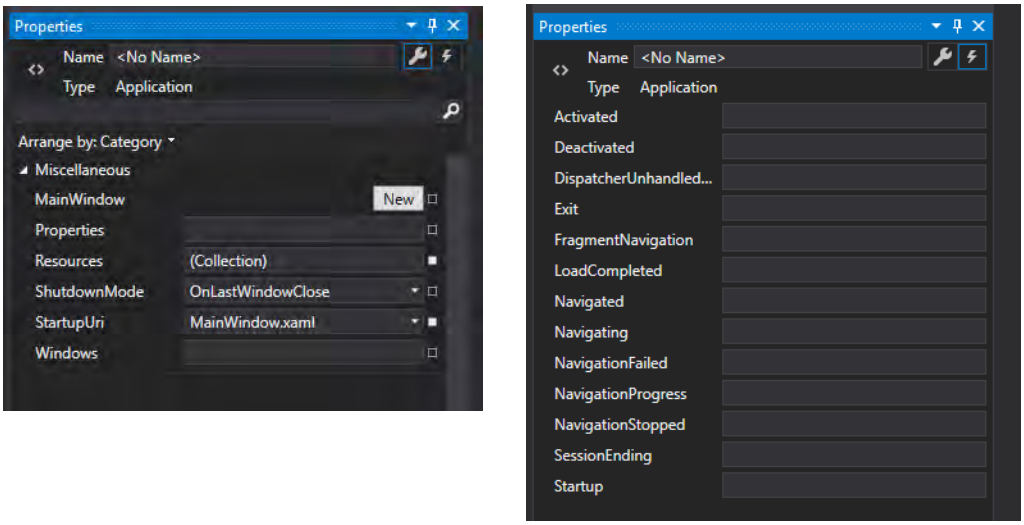

Figure 12-25: App.xaml Properties and Events

Referring to figure 12-25 — Note that Properties are viewed by clicking on the wrench icon in the upper right-hand corner, while Events are revealed by clicking on the lightening bolt icon. In the Properties pane, note that StartupUri is set to MainWindow.xaml, and ShutdownMode is set to OnLastWindowClose, which means the application will shut down when the last window is closed or the Shutdown() method is called.

Referring now to the Events pane, you can provide custom event handlers for the listed events. For example, when the application receives the Activated event, you might perform a check to ensure all resources like files and data connections are still valid and take appropriate action if not.

To keep things in context, remember that the App.xaml file contains an **<Application>** tag. *Application* is of type *System.Windows.Application*. If you look up the System.Windows.Application class on Microsoft Docs, you'll see the correspondence between the properties and events shown in the panes above with those listed in the documentation. You can set properties in either the XAML file or its code-behind. You can set and manipulate properties in the code-behind, and respond to events by implementing the appropriate event handler code in the code-behind as well. You'll see examples of this later in the section. Speaking of code-behinds, let's look now at the App.xaml.cs file.

12.18 App.xaml.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Configuration;
4  using System.Data;
5  using System.Linq;
6  using System.Threading.Tasks;
7  using System.Windows;
8
9  namespace PresentationLayer {
10   /// <summary>
11   /// Interaction logic for App.xaml
12   /// </summary>
13   public partial class App : Application {
14   }
15 }
```

Referring to example 12.18 — Though quite sparse, there's a lot going on here. First, the App.xaml markup file and its code-behind, App.xaml.cs, are compiled together to fully implement the behavior defined in both files. Second, there's a lot of boilerplate code hidden from you. To reveal it, you'll need to expand the App.xaml.cs file in Solution Explorer to see the Main() and InitializeComponent() methods, which are located in a file named App.g.i.cs, which is automatically generated when the project builds. (**NOTE:** This file is located in the project's `../obj/debug/` or `../obj/release/` directory, depending on which target (debug or release) is used to build the solution.) I'm diving into the weeds here, but it's super helpful to understand how Visual Studio relates the XAML file to its code-behind and what sections of code are automatically generated and off limits, and which sections of code are safe to edit. The biggest source of anxiety among programmers new to Visual Studio is the seemingly impossible task of recovering from a simple project editing or configuration mistake which can break things to the point of having to abandon the project altogether in favor of starting from scratch. Understanding how Visual Studio works goes a long way towards alleviating that anxiety. (Well, that was a bit touchy-feely.)

Let's quickly examine the MainWindow.xaml and its code-behind.

12.19 MainWindow.xaml

```
1  <Window x:Class="PresentationLayer.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6          xmlns:local="clr-namespace:PresentationLayer"
7          mc:Ignorable="d"
8          Title="MainWindow" Height="450" Width="800">
9      <Grid>
10
11     </Grid>
12 </Window>
```

Referring to example 12.19 — The top-most tag in this markup file is of type *System.Window*, and its code-behind is the MainWindow class as set by the `x:Class` attribute. On line 8, the MainWindow's Title property is set to "MainWindow", along with its initial Height and Width. You can also access and manipulate these properties, along with any events a Window can respond to, in the code-behind.

If you examine MainWindow.xaml's File Properties, you'll see the Build Action is set to Page, which means the XAML markup file will be compiled into a binary .baml file for faster runtime loading.

12.20 MainWindow.xaml.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows;
```



```
7   using System.Windows.Controls;
8   using System.Windows.Data;
9   using System.Windows.Documents;
10  using System.Windows.Input;
11  using System.Windows.Media;
12  using System.Windows.Media.Imaging;
13  using System.Windows.Navigation;
14  using System.Windows.Shapes;
15
16  namespace PresentationLayer {
17    /// <summary>
18    /// Interaction logic for MainWindow.xaml
19    /// </summary>
20    public partial class MainWindow : Window {
21      public MainWindow() {
22        InitializeComponent();
23      }
24    }
25  }
```

Referring to example 12.20 — Note that MainWindow is a partial class that extends System.Windows.Window. It provides a default constructor method that calls the InitializeComponent() method. The InitializeComponent() method is auto-generated, so no need to go hunting for that since you're not supposed to edit it anyway.

Now that you have a feeling for how things are connected in a newly-created Visual Studio WPF project, it's time to make some modifications and get a feel for how to customize the application.

Adding Functionality

In this section I'll walk through how to create an application with WPF. The launch point will be the SimpleWPFApp created above. I'll show you how to set and modify properties, add UI components to windows, create new windows, menus, and menu items, and how to respond to UI events in the code-behinds. This simple application isn't going to do anything particularly useful, rather, the point of this section is to provide a contrast between Windows Forms-based GUI development covered earlier in the chapter, vs. WPF-based GUI development. Note that both technologies provide a visual means to design and layout the user interface.

Layer Responsibility

Before getting started, I'd like to emphasize that the primary concern of the PresentationLayer project will be that of implementing the application user interface and reacting to user interface events. For really small applications, you might decide to implement all functionality in one project, including business logic and data access. There's nothing wrong with that approach for small projects, but larger, sophisticated applications demand a separation between application layers, and organizationally it's considered best practice to maintain a tight as focus as possible on project responsibilities. I have a series of videos on YouTube devoted to this topic: https://www.youtube.com/watch?v=fNjD_KhGHRM&t=0s

Application Modifications

The application modifications I want to focus on include:

- Use an Application resource to show a splash screen when the application launches
- Add menus and menu items to a window
- Respond to menu events with event handlers in the code-behind

This should be enough to give you a good overview of how GUI development using WPF differs from Windows Forms.

Application Resources and Splash Screens

The App.xaml file contains an `<Application.Resources>` tag which is a dictionary of key/value pairs. Resources defined here are available application-wide. You can also store resources in a resources file (.resx). I'll demonstrate both approaches. The PresentationLayer project stores resources in the Properties.Resources.resx file. Also found in the Properties namespace is the Settings.settings file. The difference between resources and settings is resources are packaged along with the application and are non-trivial to change. Settings are contained within configuration files and its values can be easily modified between application runs to suit the environment. I'll focus on resources here and introduce you to settings later in the book when we use configuration files to hold database connection strings and other configurable items. So, let's use resources to help build splash screens.

You've seen a splash screen. It's a window that briefly appears when an application launches and either disappears on its own or as the result of some user action. Splash screens provide an interesting discussion point related to WPF because, as with most software development tasks, there are several ways to skin the cat here. Apologies to any cat lovers for that one.

Creating A Simple Splash Screen

The WPF provides a simple way to specify a splash screen. It's so simple, I'm not even going to bother showing you how to do it, rather, I'll leave it as an exercise, and just briefly describe the steps involved.

Step 1: Add an image to the PresentationLayer project. Acceptable formats include: JPEG, PNG, GIF, BMP, TIFF - something along those lines. I recommend creating a new folder within the project named either *Media* or *Images* and add the image to the folder. Make sure the image is sized to the appropriate dimensions, because the easy way doesn't allow you to alter the image before it displays.

Step 2: Right-click the image and from the **Build Action** drop-down select **Splash-Screen**.

Bada-Bing-Bada-Boom! — You're smokin' cigarettes. Now, run the application. The image should appear before the main window. If you used a rather large image, you'll realize it as soon as it appears on the screen. Now, let's create a custom splash screen that offers more control over how long the image appears on the screen and its fadeout time.

Creating A Custom Splash Screen

In this example, I'll show you how to create and display a custom splash screen with a little bit of code in the code-behind. Before proceeding, right-click the image you used earlier to create the simple splash screen and from the **Build Action** drop-down select **Resource**, then follow the general steps below to create a custom splash screen. I will start by adding an Application Resource that contains the path to the image file.

Step 1: Add Reference To App.xaml

Open the App.xaml and within the `<Application.Resources>` tags add a string entry that represents the path to the image you want to use for the splash screen. You'll also need to add a reference to the System namespace as is shown in example 12.21.

12.21 App.xaml

```
1  <Application x:Class="PresentationLayer.App"
2               xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3               xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4               xmlns:local="clr-namespace:PresentationLayer"
5               xmlns:sys="clr-namespace:System;assembly=mscorlib"
6               StartupUri="MainWindow.xaml">
```



```
7       <Application.Resources>
8           <sys:String x:Key="splash_image">Images/PFP-Banner-Small.png</sys:String>
9       </Application.Resources>
10  </Application>
```

Referring to example 12.21 — First, on line 5, to gain access to the String type I needed to add an XML namespace declaration named `sys` (xmlns:sys). Then, on line 8, I used the `sys` prefix to declare a String resource with the key "splash_image" and value "Images/PFP-Banner-Small.png", which is the path to the image.

Step 2: Edit MainWindow Code-Behind

In the MainWindow.xaml.cs code-behind file, create a private method that will contain the code for the custom splash screen. Example 12.22 shows how I implemented the method.

12.22 ShowSplashImage() Method

```
1   private void ShowSplashImage() {
2     BitmapImage img = new BitmapImage();
3     img.BeginInit();
4     img.UriSource = new Uri((string)Application.Current.Resources["splash_image"],
5                             UriKind.RelativeOrAbsolute);
6     img.EndInit();
7     SplashScreen sc = new SplashScreen(img.UriSource.ToString());
8     sc.Show(false, true); // autoClose:false, topMost:true
9     Thread.Sleep(600);
10    sc.Close(TimeSpan.FromSeconds(5));
11  }
```

Referring to example 12.22 — The ShowSplashImage() method creates a BitmapImage and, on lines 4 and 5, sets its UriSource property to the images's file path. This value is obtained by searching the `Application.Current.Resources` dictionary using the "splash_image" key. The calls to img.BeginInit() and img.EndInit() are important because omitting them breaks the code. Once the image's UriSource is set, a new SplashScreen object is created using the image's UriSource string value. A call is then made to the SplashScreen's Show() method with the first parameter, *autoClose* set to `false`, and the second parameter, *topMost*, set to `true`. The call to Thread.Sleep() on line 9 allows the splash screen to display for 600 milliseconds, a fair amount of time, and finally, the call to SplashScreen's Close() method allows the image to fade out over a period of five seconds. OK, with this method in place, we need some way to automatically call it when the application launches. That's the topic of Step 3.

Step 3: Automatically Calling The ShowSplashImage() Method

Like the simple splash screen, you may want the custom splash screen to display when the application launches. As it turns out, this is easy-peasy to do with events. For this step, you can add an event handler method in the MainWindow's code-behind, and have it call the ShowSplashImage() method. Then, you'll need to associate the event with the event handler method, and you can do that in the MainWindow.xaml file. First, though, a word of caution. You can approach this task in several different ways. You could use the properties panel to add an event handler. This will automatically create the method stub in the code-behind and modify the related xaml file as well. Once you know what you're doing, all this auto-generated, make-you-life-easy stuff can actually begin to irritate the shit out of you...it does me sometimes. Anyway, you've been briefed, now let's get crackin'.

First, you'll need to select an appropriate event to use to show the splash screen. You can actually have a lot of fun with this as there are many events in the life of an application or window to which you can respond. I'm going to use the Window.Loaded event, and I will implement the event handler method in the code-behind first. Example 12.23 gives the code.

12.23 Window_Loaded() Event Handler

```
1   private void Window_Loaded(object sender, EventArgs e) {
2     ShowSplashImage();
3   }
```

Referring to example 12.23 — If you're used to writing event handlers for Windows Forms GUIs, you're in familiar territory. The Window.Loaded event is fired when a window is laid out, rendered, and ready to respond to user interaction. As this is the application's main window, this will be a good event to use to display the splash screen. You could also respond to the Application.StartUp event. I'll leave that as an exercise. Example 12.24 shows the required modification to the MainWindow.xaml file.

12.24 MainWindow.xaml

```
1   <Window x:Class="PresentationLayer.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6       xmlns:local="clr-namespace:PresentationLayer"
7       mc:Ignorable="d"
8       Title="MainWindow" Height="450" Width="800"
9       WindowStartupLocation="CenterScreen"
10      Loaded="Window_Loaded" >
11
12      <Grid>
13
14      </Grid>
15  </Window>
```

Referring to example 12.24 — On line 10, the Window.Loaded event is associated with the Window_-Loaded() event hander declaratively within the `<Window>` tag. Note, too, on line 9, I've set the `Window-StartupLocation` to `CenterScreen`. OK, time to test the custom splash screen. Run the application. If you used the image I supplied with the project you should see something resembling figure 12-26 when the application launches and the main window displays.

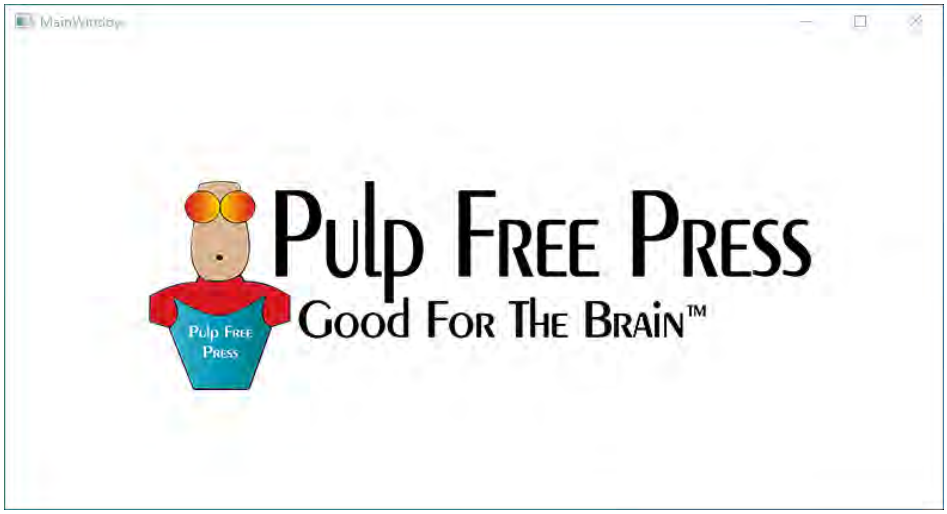

Figure 12-26: Custom Splash Screen

Referring to figure 12-26 — When you launch the application, the splash screen will display and slowly fade away, leaving the main window. In the next section, I'll take a slightly different approach to a splash screen effect, and show you how to create a custom window that shows a short movie.

Creating A Custom Splash Window

The SplashScreen class is limited to displaying only images. What if you'd rather show an animated splash screen or a short movie? For these situations you'll need to create and display a custom window and use a MediaElement tag to play the video. This approach involves custom coding in the code-behind, and manual manipulation of the MediaElement component. Thankfully, the WPF provides the heavy lifting, you just need to supply the creativity.

Earlier I showed you how to use Application resources by adding a string entry to the `<Application.Resources>` tag in the App.xaml file. This time I will take a different approach in this regard as well, and add an entry to the PresentationLayer's Properties.Resources.resx file which will be a string path value to the movie's location within the project. Again, if you're creating this from scratch as you follow along in the book, create a new folder within the project named Videos and add your video there, or add it to the Media folder you may have created earlier. The names of these folders matters little; what does matter is that you remember where you put them. You could even download them from the Internet when the application runs, but that's something I'll leave for you to explore on your own.

Step 1: Add Image Path To Project Resources.resx File

Open the project's Resources.resx file and add the following entry:

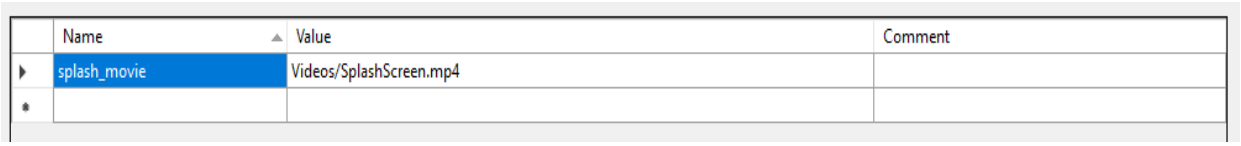

Figure 12-27: Adding Movie Location Path to Project Resources File

Referring to figure 12-27 — I've added an MP4 movie file named "SplashScreen.mp4" to the Videos folder. The entry in the Project Resources file adds a string resource "Videos/SplashScreen.mp4" and associates it with the key "splash_movie". As you'll soon see, accessing resources defined within a resources file is different from accessing resources defined within `<Application.Resources>`.

Step 2: Create Custom Window

Again, before proceeding, it's helpful to consider project organization. I created a new folder within the project named Windows. As I write this, in retrospect, I should have shown a bit more naming finesse, but as it stands, a folder essentially introduces a new namespace, and I just need to fully qualify the namespace when I refer to a custom window class that resides therein. Just something to be aware of when you set out to organize your projects.

In the Windows folder I created a new WPF window and named it **Intro.xaml**. This also creates the associated code-behind. Starting first with the Intro.xaml file, you need to add a `<MediaElement>` tag and set its `x:Name` attribute as shown in example 12.25.

12.25 Intro.xaml

```
1   <Window x:Class="PresentationLayer.Windows.Intro"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6           xmlns:local="clr-namespace:PresentationLayer.Windows"
7           mc:Ignorable="d"
8           Background="Transparent"
9           ResizeMode="NoResize"
10          Title="Simple WPF Splash Screen"
11          Height="225" Width="400"
```



```
12          WindowStartupLocation="CenterOwner"
13          ShowInTaskbar="False"
14          Topmost="True"
15          WindowStyle="None"
16          MouseDown="Window_MouseDown" >
17    <Window.Effect>
18      <DropShadowEffect/>
19    </Window.Effect>
20    <Grid>
21      <MediaElement x:Name="SplashMovie" LoadedBehavior="Manual"/>
22    </Grid>
23  </Window>
```

Referring to example 12.25 — I've set several attributes on the `<Window>` element to remove the border, prevent resizing, and to generally make it look like a splash screen. On line 16, I've set the MouseDown event to call the Window_MouseDown event handler, which I'll define below. On line 18, I've added a window effect `<DropShadowEffect>`, and finally, on line 21, I've added a `<MediaElement>` tag and set it's `x:Name` attribute to "SplashMovie", and its `LoadedBehavior` to Manual.

Next, we need to edit the Intro window's code-behind so it will play the movie when the Intro window is created, and close when a user clicks on the window. The code is shown in example 12.26 below.

12.26 Intro.xaml.cs

```
1   using System;
2   using System.Windows;
3   using System.Windows.Input;
4
5   namespace PresentationLayer.Windows {
6     /// <summary>
7     /// Interaction logic for Intro.xaml
8     /// </summary>
9     public partial class Intro : Window {
10      public Intro() {
11        InitializeComponent();
12        SplashMovie.Source = new Uri(Properties.Resources.splash_movie,
13                                     UriKind.RelativeOrAbsolute);
14        SplashMovie.Play();
15      }
16
17      private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
18        this.Close();
19      }
20    }
21  }
```

Referring to example 12.26 — I've placed the code to play the movie in the Intro window's constructor method. I could have created a separate private method and called that method from the constructor, but the approach I took above will work just fine. First, on line 12, after the InitializeComponent() method, I access the MediaElement by name, SplashMovie, and set its Source property by creating a new Uri. Note how I'm accessing the project resource named "splash_movie" (i.e., Properties.Resources.splash_movie). (**Note:** Compare this method of resource access with that of Application.Current.Resources["splash_image"]. This is a dictionary lookup, where as Properties.Resources.*splash_movie* is accessing a property.) Finally, on line 14, the move is played. A user can close the Intro window at any time by clicking on the window, which fires the mousedown event, which is handled by the Window_MouseDown method starting on line 17.

You're not done yet! One more edit. The Intro window is complete, but you need a way to open that window when the application launches or the MainWindow displays. You can use the same event handler method utilized in the previous section to show the Intro window. Here's the modified MainWindow code-behind.

12.27 MainWindow.xaml.cs (mod 1)

```
1   using System;
2   using System.Threading;
3   using System.Windows;
4   using System.Windows.Media.Imaging;
5
6   namespace PresentationLayer {
7     /// <summary>
8     /// Interaction logic for MainWindow.xaml
9     /// </summary>
10    public partial class MainWindow : Window {
11      public MainWindow() {
12        InitializeComponent();
13      }
14
15      private void Window_Loaded(object sender, EventArgs e) {
16        ShowIntroWindow();
17      }
18
19      private void ShowIntroWindow() {
20        try {
21          Window intro = new Windows.Intro();
22          intro.Owner = this;
23          intro.ShowDialog();
24        } catch (Exception) {
25          Console.WriteLine("Problem loading Intro Window");
26        }
27      }
28
29      private void ShowSplashImage() {
30        BitmapImage img = new BitmapImage();
31        img.BeginInit();
32        img.UriSource = new Uri((string)Application.Current.Resources["splash_image"],
33                        UriKind.RelativeOrAbsolute);
34        img.EndInit();
35        SplashScreen sc = new SplashScreen(img.UriSource.ToString());
36        sc.Show(false, true); //autoClose:false, topMost:true
37        Thread.Sleep(600);
38        sc.Close(TimeSpan.FromSeconds(5));
39      }
40    } // End class
41  }
```

Referring to example 12.27 — I've added a method named ShowIntroWindow() beginning on line 19, which creates a new Intro window, sets its Owner to the containing window (this), then shows the window as a modal dialog, meaning no user input will be accepted anywhere else in the application until the Intro window is dismissed. I've modified the Window_Loaded() event handler and replaced the original call to ShowSplashImage() with ShowIntroWindow(). Figure 12-28 shows the results of running the application.

Menus, Menu Items, and Menu Events

The final WPF topic I'd like to cover in this chapter is how to add menus and respond to menu events. To demonstrate these concepts I'll add a menu bar to the MainWindow with two menus: File and Help. I'll add several menu items to each menu, but will only implement event handling on the Help menu items.

Add Menus and Menu Items

Open the MainWindow.xaml file and make the following modifications as shown in example 12.28.

Figure 12-28: Using Intro Window as a Splash Screen — Playing a Movie

12.28 MainWindow.xaml (Menu Mods)

```
1   <Window x:Class="PresentationLayer.MainWindow"
2           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6           xmlns:local="clr-namespace:PresentationLayer"
7           mc:Ignorable="d"
8           Title="MainWindow" Height="450" Width="800"
9          WindowStartupLocation="CenterScreen"
10         Loaded="Window_Loaded"
11         MouseDown="Window_MouseDown">
12
13    <Grid>
14      <DockPanel>
15        <Menu DockPanel.Dock="Top">
16          <MenuItem Header="_File">
17            <MenuItem Header="_Open"/>
18            <MenuItem Header="_Close"/>
19            <MenuItem Header="_Save"/>
20          </MenuItem>
21
22          <MenuItem Header="_Help">
23            <MenuItem Header="_About"/>
24            <MenuItem Header="_Show Splash Image"/>
25            <MenuItem Header="_Pulp Free Press"/>
26          </MenuItem>
27        </Menu>
28      </DockPanel>
29    </Grid>
30  </Window>
```

Referring to example 12.28 — I've added a `<DockPanel>` tag on line 14, followed by a `<Menu>` tag and set its `DockPanel.Dock` attribute to "Top". The first `<MenuItem>` tag starts on line16. Note that menu items that show up on the menu bar encompass sub-menu items, which are defined as child tags to the parent `<MenuItem>` tag. I have added underscore characters '_' to each Header value to indicate menu item accelerator keys. (Alt + F for File, O for Open, etc.) There's no action associated with any of these

menu commands just yet, so let's take care of that. Edit the MainWindow code-behind and make it look like example 12.29.

12.29 MainWindow.xaml.cs (Menu Event Handler Mods)

```
1   using System;
2   using System.Threading;
3   using System.Windows;
4   using System.Windows.Media.Imaging;
5
6   namespace PresentationLayer {
7     /// <summary>
8     /// Interaction logic for MainWindow.xaml
9     /// </summary>
10    public partial class MainWindow : Window {
11      public MainWindow() {
12        InitializeComponent();
13      }
14
15      private void Window_Loaded(object sender, EventArgs e) {
16        ShowIntroWindow();
17      }
18
19      private void Show_Splash_Image(object sender, EventArgs e) {
20        ShowSplashImage();
21      }
22
23      private void Show_Intro_Window(object sender, EventArgs e) {
24        ShowIntroWindow();
25      }
26
27      private void Goto_PulpFreePress(object sender, EventArgs e) {
28        System.Diagnostics.Process.Start(@"https://pulpfreepress.com");
29      }
30
31      private void Window_MouseDown(object sender, EventArgs e) {
32        ShowSplashImage();
33      }
34
35      private void ShowIntroWindow() {
36        try {
37          Window intro = new Windows.Intro();
38          intro.Owner = this;
39          intro.ShowDialog();
40        } catch (Exception) {
41          Console.WriteLine("Problem loading Intro Window");
42        }
43      }
44
45      private void ShowSplashImage() {
46        BitmapImage img = new BitmapImage();
47        img.BeginInit();
48        img.UriSource = new Uri((string)Application.Current.Resources["splash_image"],
49                        UriKind.RelativeOrAbsolute);
50        img.EndInit();
51        SplashScreen sc = new SplashScreen(img.UriSource.ToString());
52        sc.Show(false, true); //autoClose:false, topMost:true
53        Thread.Sleep(600);
54        sc.Close(TimeSpan.FromSeconds(5));
55      }
56
57    } // End class
58  }
```


Referring to example 12.29 — I've added three event handlers. Show_Splash_Image(), Show_Intro_Window(), and Goto_PulpFreePress(), which is an added bonus because in this method I open a browser using a call to System.Diagnostics.Process.Start() and pass in a URL to the website. The Show_Splash_Image() event handler simply calls the ShowSplashImage() method. That's the nice thing about organizing your code into methods. You can call them when you need them and reduce code repetition. The same holds true for the Show_Intro_Window() event handler. Now, let's associate these event handlers with menu items. Return to MainWindow.xaml and edit the corresponding menu item tags. Example 12.30 shows the modified code.

12.30 MainWindow.xaml (Menu Event Handler Mods)

```
<Window x:Class="PresentationLayer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PresentationLayer"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800"
       WindowStartupLocation="CenterScreen"
       Loaded="Window_Loaded"
       MouseDown="Window_MouseDown">

  <Grid>
    <DockPanel>
      <Menu DockPanel.Dock="Top">
        <MenuItem Header="_File"> <!-- Alt + F -->

          <MenuItem Header="_Open"/> <!-- O -->
          <MenuItem Header="_Close"/> <!-- C -->
          <MenuItem Header="_Save"/> <!-- S -->
        </MenuItem>

        <MenuItem Header="_Help"> <!-- Alt + H -->
          <MenuItem Header="_About" Click="Show_Intro_Window"/> <!-- A -->
          <MenuItem Header="_Show Splash Image" Click="Show_Splash_Image"/> <!-- S -->
          <MenuItem Header="_Pulp Free Press" Click="Goto_PulpFreePress" /> <!-- P -->
        </MenuItem>
      </Menu>
    </DockPanel>
  </Grid>
</Window>
```

Referring to example 12.30 — The Help -> About, Help -> Show Splash Image, and Help -> Pulp Free Press menu items each have their Click events set to call the corresponding event handler in the code-behind. The menu item shortcuts are given in comments to the right of each menu item. The File menu, like I said, has no real functionality at this time. That's it! Run the application and try out the menu items. You can show the Intro "splash screen" movie with Help -> About, show the splash screen image, and goto https://pulpfreepress.com. Before leaving this section, I want to emphasize that I've only scratched the surface of menus and menu events. This topic alone would warrant an entire chapter in a dedicated WPF book, so, have fun, play around, experiment!

Quick Review

The Windows Presentation Foundation (WPF) enables a mixture of declarative and imperative programming paradigms to create modern, Windows graphical user interfaces (GUIs). WPF windows consist of two related files: a XAML file where GUI elements can be added and configured, and a corresponding code-behind where those components can be manipulated and further functionality added with C#.

The primary differences between application resources vs. properties is that resources are packaged along with the application and not easily changed after deployment. Properties can be defined within configuration files and are easily changed to enable application customization based on its deployed environment. Application resources can be specified as an entry inside `<Application.Resources>` tags, or in a project's Resources.resx file. A project can have multiple resource files.

Summary

The Form class, found in the System.Windows.Forms namespace, serves as the basis for all types of windows you might need to create in your application. These include standard, tool, borderless, or floating windows. The Form class is also used to create dialog boxes and multiple-document interface (MDI) windows. A Form is a ContainerControl, a ScrollableControl, a Control, a Component, a MarshalByRefObject, and ultimately an Object. OK, let's start with App.xaml.

The Form class provides a lot of functionality right out of the box. You can drag the window around the screen, resize the window, minimize the window, maximize the window, and close the application by clicking the box with the "X" in the upper right corner.

Microsoft Windows applications are *event-driven*. When launched they wait patiently for an event such as a mouse click or keystroke to occur. Events are delivered to the application in the form of *messages*. Messages can be generated by the operating system in response to various types of stimuli, including direct user interaction (*i.e.,* mouse movement, clicking, scrolling, keystrokes, etc.), or by other applications.

The majority of system generated messages are placed into a data structure referred to as the *system message queue*. A *queue* is a data structure that has a first-in-first-out (FIFO) characteristic. Messages are placed into the queue as they occur. Each message contains data that includes the type of event, target window, cursor coordinates, etc. The system then examines each message and routes it to its target window.

There are two types of coordinates: *screen coordinates* and *window coordinates*. Window coordinates are also referred to as client coordinates. The basic unit of measure for a screen is the *pixel*. The origin of the screen, or the point where both the value of its x and y coordinates equal 0, is located in the upper left corner of the screen. The value of the x coordinate increases to the right, while the value of the y coordinate increases down. Points upon the screen are given in (x, y) coordinate pairs. Windows have a coordinate system similar to the screen, with their origin located in the upper left hand corner of the window. Windows, and the components drawn within them, have height and width. The bounds of a component are the location of its upper left corner together with its width and height.

The Form class provides many properties, methods, and events which make it easy to manipulate them in your programs. The Form class gets most of its functionality via its inheritance hierarchy. What you can do to a form you can also do, for the most part, to other controls, containers, and components.

Setting a control's properties often requires the use of other classes, structures, or enumerations found in the .NET Framework. A few of these include System.Drawing.Point, System.Drawing.Rectangle, System.Drawing.Color, System.Drawing.Bitmap, and System.Drawing.Image. The type of property will determine what type of object you must use to set the property.

To add a control like a Button or TextBox to a window, you must first declare and create the control, set its properties, and then add the control to the window's Controls collection. The absolute placement of controls can be tedious. Use the System.Drawing.Rectangle class to set a control's Bounds property. You may alternatively set a control's Top, Left, Width, and Height properties separately.

The whole point of creating a GUI is to have it respond to user interaction. All System.Windows.Forms GUI controls have Event members. An *event* is something an object can respond to. For example, a Button can respond to a mouse click via its Click event.

A *delegate* declares a new type in the form of a method signature. Events are class members declared to have a certain delegate type, meaning that a method assigned to handle that event must have the specified delegate's method signature.

Use the '+=' operator to assign an event handler method to a control's event. Give your event handler methods names that clarify their role as event handlers.

Knowing how to write code so that GUI events generated in one object are handled by event handler methods located in another object is a critical programming skill. To do this you, must know how to do the following things: 1) create a stand-alone, non-application GUI class that extends Form, 2) create a separate application class that uses the services of the GUI class; this application class will also contain the required event handler code, 3) create GUI class constructors that take a reference to the object that contains the event handler code, 4) register a component's event with the event handler code via the supplied reference, and 5) create appropriate methods or properties in the GUI class that allows horizontal manipulation of private GUI components.

Use the FlowLayoutPanel and TableLayoutPanel classes to automatically control the layout of controls in a window. Controls placed in a FlowLayoutPanel flow into the panel from left-to-right, top-to-bottom by default. Don't forget to set the panel's Dock property.

The TableLayoutPanel is used to place controls into a grid arrangement, where each control occupies a cell that can be accessed via x and y coordinates, much like a two-dimensional array.

You can add panels to panels to create complex GUIs.

Use menus to give your GUI a professional appearance. You can create menus using the MenuStrip and ToolStripMenuItem classes.

TextBoxes can be used in single-line or multiline mode. To create a multiline text box set its Multiline property to true. To determine the line number for a line of text in a text box use the TextBox.SelectionStart() method to first get the index of the selected character, then use the TextBox.GetLineFromCharIndex() method to get the text line.

Getting into the rhythm of writing GUI code can make writing the code much less tedious and lots of fun.

The Windows Presentation Foundation (WPF) enables a mixture of declarative and imperative programming paradigms to create modern, Windows graphical user interfaces (GUIs). WPF windows consist of two related files: a XAML file where GUI elements can be added and configured, and a corresponding code-behind where those components can be manipulated and further functionality added with C#.

The primary differences between application resources vs. properties is that resources are packaged along with the application and not easily changed after deployment. Properties can be defined within configuration files and are easily changed to enable application customization based on its deployed environment. Application resources can be specified as an entry inside `<Application.Resources>` tags, or in a project's Resources.resx file. A project can have multiple resource files.

Skill-Building Exercises

1. **API Drill:** Visit the .NET API documentation located on the MSDN website and explore the System.Windows.Forms namespace. List and briefly describe the purpose of each class, structure, enumeration, and delegate.

2. **API Drill:** Visit the .NET API documentation located on the MSDN website and explore the System.Drawing namespace. List and briefly describe the purpose of each class, structure, enumeration, and delegate.

3. **API Drill:** Visit the .NET API documentation located on https://docs.microsoft.com and research the System.Windows.Forms.Control class. List and briefly describe the purpose of each of its members.

4. **Code Drill:** Experiment with control docking. First, visit https://docs.microsoft.com and research the purpose of the Control.Dock property. Write a short program that puts several buttons in a window. Set each button's Dock property using the DockStyle enumeration. Note the effects each different DockStyle location has upon the buttons.

5. **Code Drill:** Experiment with control anchoring. First, visit https://docs.microsoft.com and research the purpose of the Control.Anchor property. Write a short program that puts several buttons in a window. Set each button's Anchor property using the AnchorStyles enumeration. Note the effects that different AnchorStyles have upon the buttons.

6. **Code Drill:** Write a program that creates 10 buttons, puts them into a FlowLayoutPanel, and displays them in a window. Experiment with setting the different FlowLayoutPanel properties and note the effects.

7. **Code Drill:** Practice creating complex GUI layouts by writing a program that uses a combination of FlowLayoutPanels and TableLayoutPanels. Add buttons to each panel and then add one type of panel to another.

8. **Code Drill:** Draw a complex user interface on a napkin or piece of paper. Include buttons, single-line and multiline textboxes, and labels. Divide the user interface into different areas, where one area might have only the multiline text box and another area the buttons, and still another area the labels and single-line textboxes. When you finish your drawing write a program that displays the controls in a window according to your plan. You don't need to worry about responding to user events or functionality. This is just a control placement exercise.

9. **API/Code Drill:** Explore the System.Windows.Forms namespace and select several controls, like CheckBox or RadioButton for example, that weren't covered in this chapter. Research their functionality and write a short program that uses them in a window.

10. **API/Code Drill (WPF):** Visit https://docs.microsoft.com and browse the Windows Presentation Foundation documentation.

Suggested Projects

1. **Programming - Robot Rat Redeux:** Revisit the Robot Rat project presented in chapter 3 and give it a graphical user interface. You may take several approaches to this project. For example, you can represent the floor as an array of labels or buttons placed within a TableLayoutPanel. This grid of controls would appear in one section of the user interface. Another section of the user interface would contain a set of buttons that allowed users to control the robot rat's movements. Separate the user interface code from the event handler code. Another approach would be to move an image of a robot rat around a window. This would require you to research how to display images directly in a window and update the window via its Paint event when the image is moved.

2. **Programming - Calculator App:** Write a program that mimics the operation of a hand-held calculator.

At a minimum implement the add, subtract, multiply, and divide operations. You may lay out the calculator's user interface any way you want, but one approach would be to put the display in the top section and a grid of buttons in the bottom section. You might even have separate sections for the numbers and the function keys. Separate the user interface code from the event handler code.

3. **WPF:** Implement Suggested Projects 1 and 2 above using the Windows Presentation Foundation.

4. **WPF Application Resources:** Looking at the ShowSplashImage() method given in example 12.27, convert the hard-coded numeric values 600 and 5 to application resources.

Self-Test Questions

1. How many different types of windows can be created with the Form class?

2. How are operating system messages generated and sent to a GUI application?

3. How are controls added to forms?

4. What are the differences between *screen coordinates* and *client coordinates*?

5. What does the term *origin* mean?

6. What four pieces of data define the bounds of a control?

7. What's the purpose of a delegate type?

8. How are delegates and event handler method signatures related? How are delegates and events related?

9. What operator do you use to assign an event handler method to a control's event?

10. Briefly describe the general steps required to respond to a control's event with an event handler located in a different object.

11. What's the purpose of the Control.SuspendLayout() method? What method should be called to resume layout?

12. What's the difference between the FlowLayoutPanel and TableLayoutPanel?

13. How can you change the direction in which controls flow into a FlowLayoutPanel?

14. How do controls flow into a TableLayoutPanel?

15. List acceptable image file formats for use with a WPF splash screen.

16. In your own words, describe how WPF GUI development differs from Window.Forms development.

References

Microsoft Developer Network (MSDN) .NET Framework Documentation: https://docs.microsoft.com/en-us/dotnet/framework/

Microsoft Docs — XAML Overview (WPF .NET): https://docs.microsoft.com/en-us/dotnet/desktop/wpf/fundamentals/xaml?view=netdesktop-5.0

Microsoft Docs — User Interface Principles: https://docs.microsoft.com/en-us/windows/win32/appuistart/-user-interface-principles

Microsoft Docs — System.Windows Namespace: https://docs.microsoft.com/en-us/dotnet/api/system.windows?view=netframework-4.7.2

Notes