Chapter 12

Pentax 67 / SMC Takumar 55/2.8 / Kodak Tri-X Professional

Fairview Park

Windows Forms Programming

Learning Objectives

- *Create Graphical User Interface (GUI) programs using Windows forms components*
- *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. Go to any good book store and you'll find several on the shelves.

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.

After you complete this chapter, the only thing you'll need to do to create spectacular GUIs is to dive deeper into the .NET API 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.

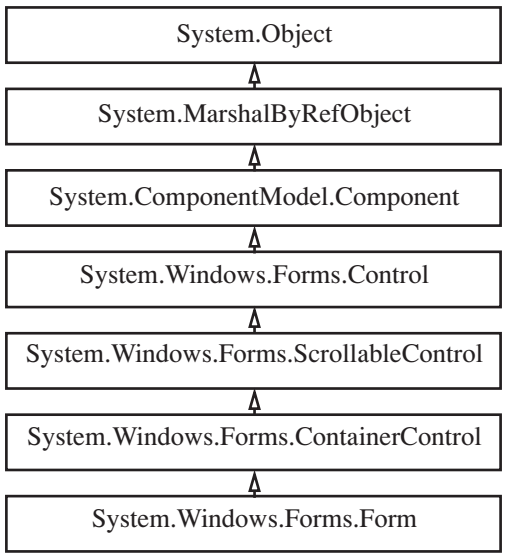

Figure 12-1: Form Class Inheritance Hierarchy

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.

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 untill 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.

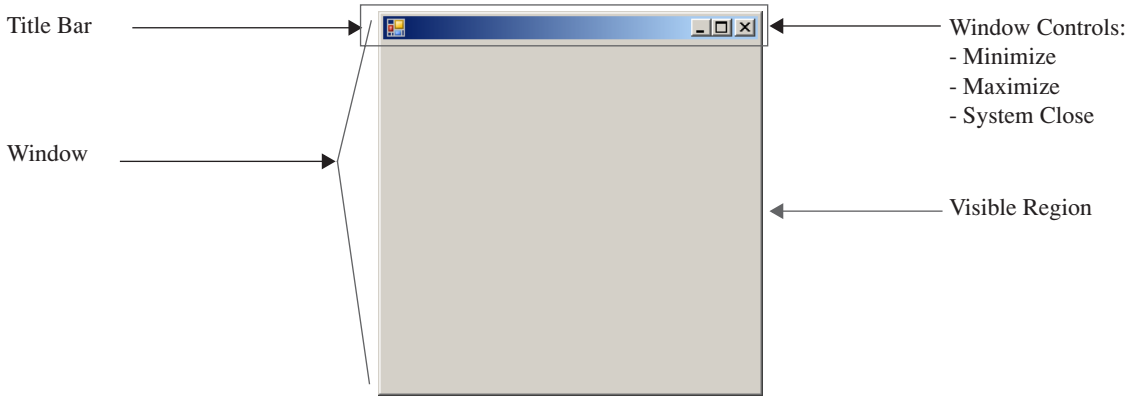

Figure 12-2: Results of Running Example 12.1

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 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. 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.

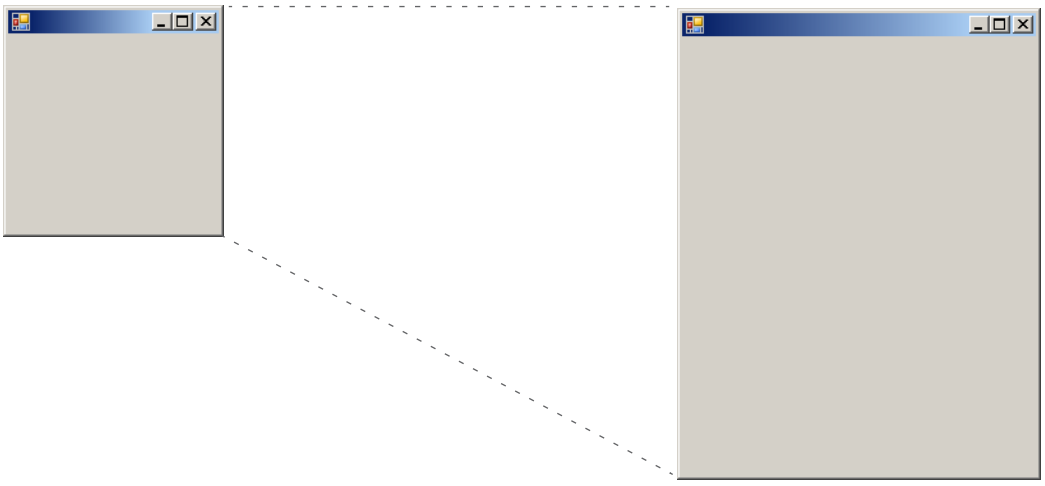

Figure 12-3: A Standard Window can be Resized by Dragging the Lower Right Corner

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.

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.

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.

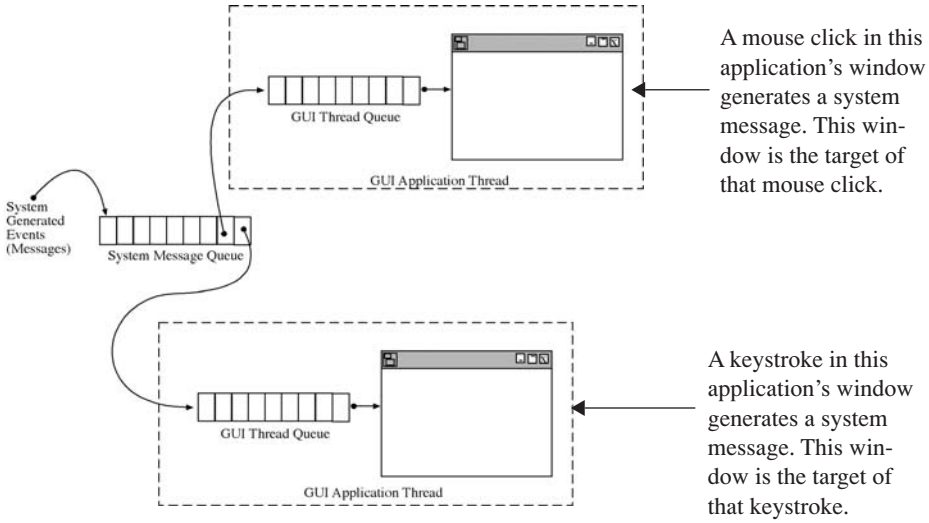

Figure 12-4: Windows Message Routing (Message Pump)

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
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.

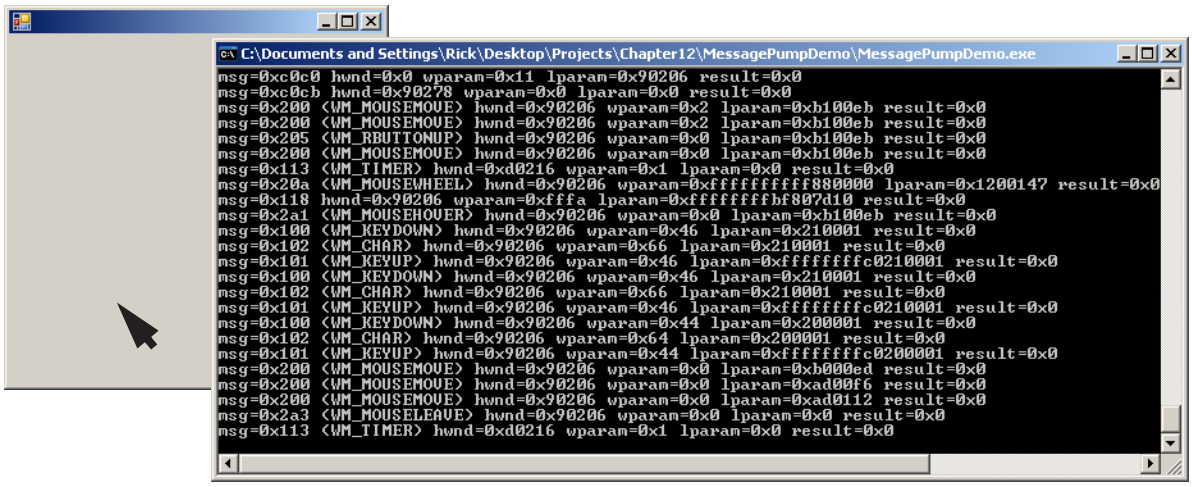

Figure 12-5: Results of Running Example 12.2

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.

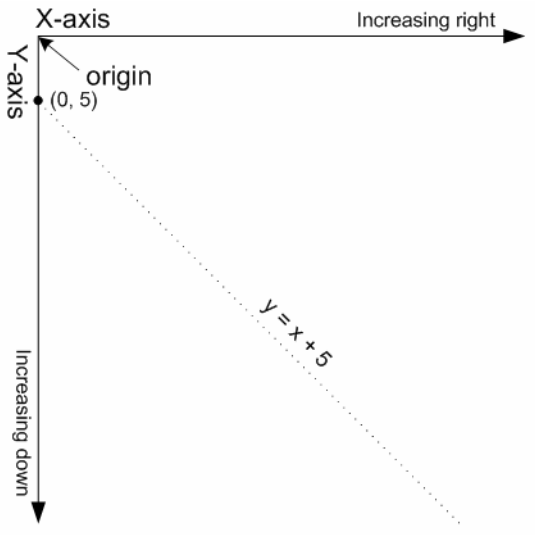

Figure 12-6: Screen Coordinate System

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) pairs.

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

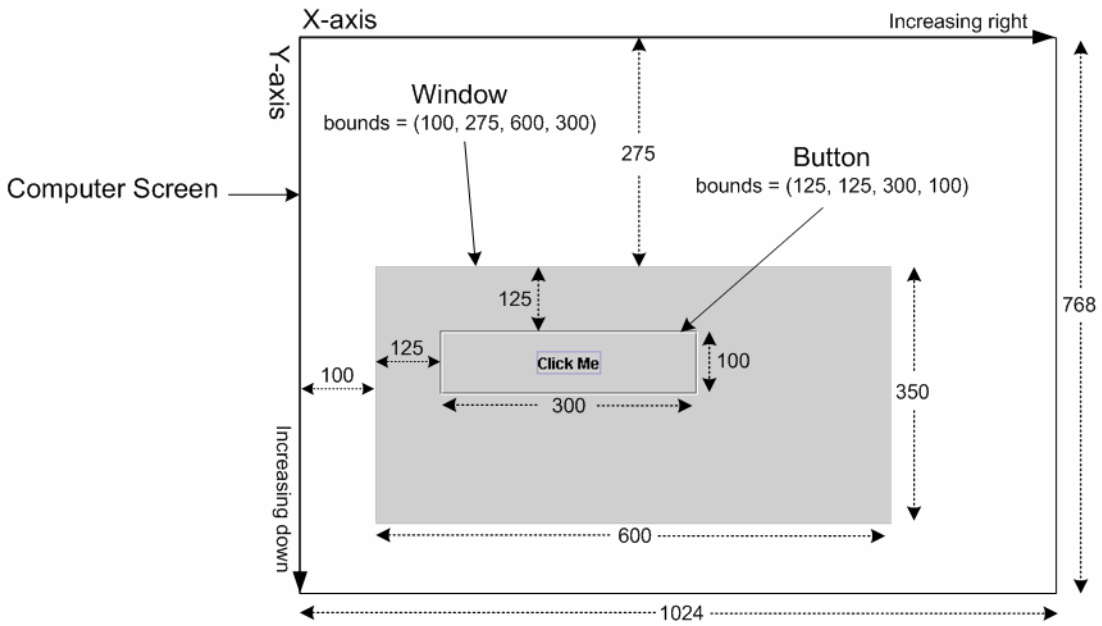

Figure 12-7: Window Coordinates

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 300 pixels wide and 100 pixels high, then its bounds are (100, 275, 300, 100). 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(){
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 previous 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.

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.

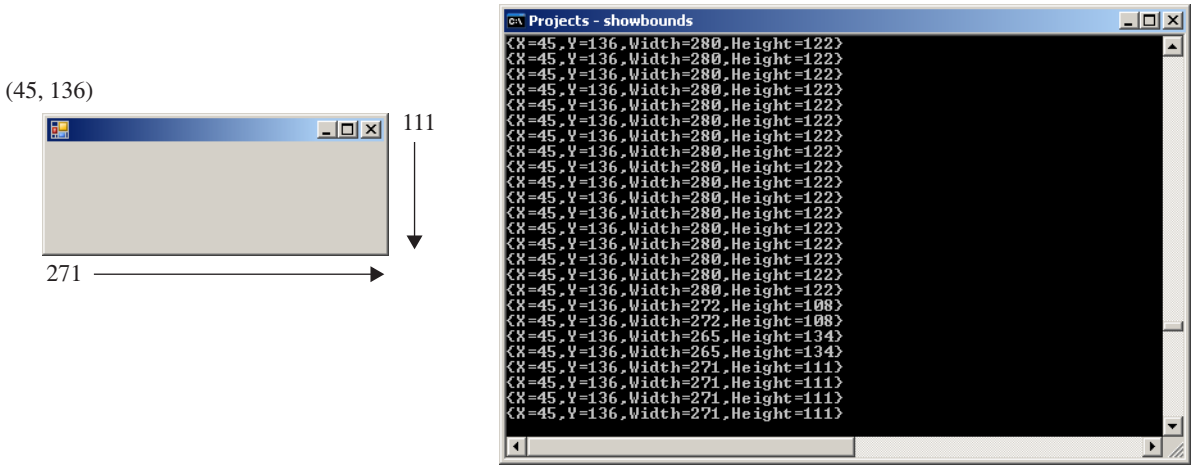

Figure 12-8: Results of Running Example 12.3

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) 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 *BackgroundImgage* 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 FormProperties.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 FormProperties 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 11 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.

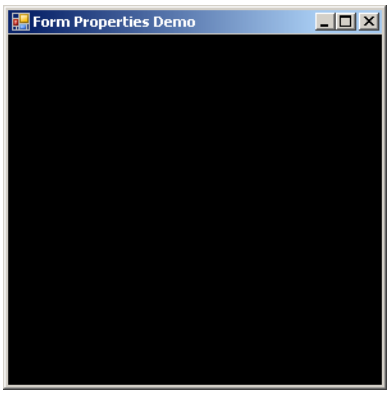

Figure 12-10: Running Example 12.4 with no Image

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.

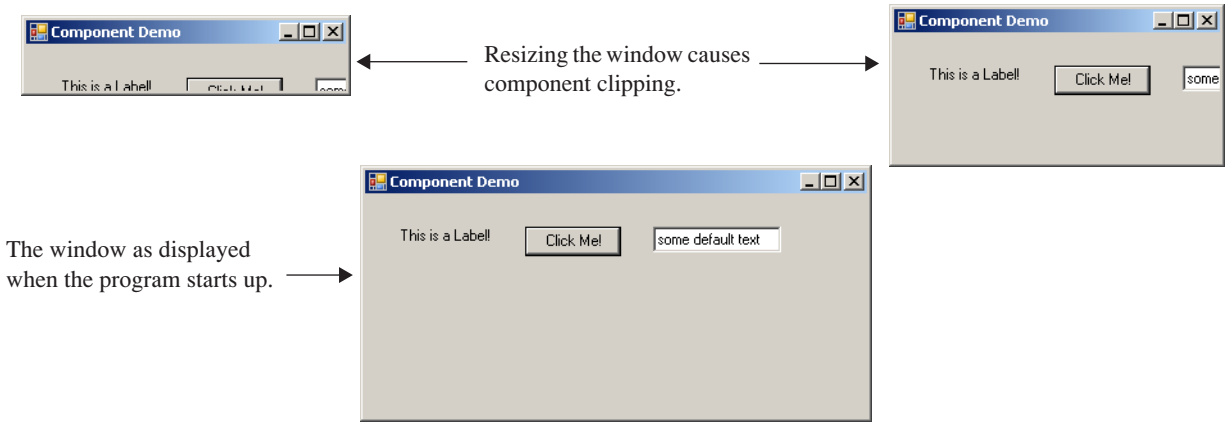

Figure 12-11: Results of Running Example 12.5

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
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 += new EventHandler(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 '+=' and `new` operators. Note that Click events require event handler methods with a signature of the EventHandler delegate type. The ButtonClickHandler() method, defined starting on line 39, sports the required signature, namely, it takes two parameters — one of type Object and the other of type EventArgs. The names 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. 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 tex tbox. 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.

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

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. 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. 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 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.

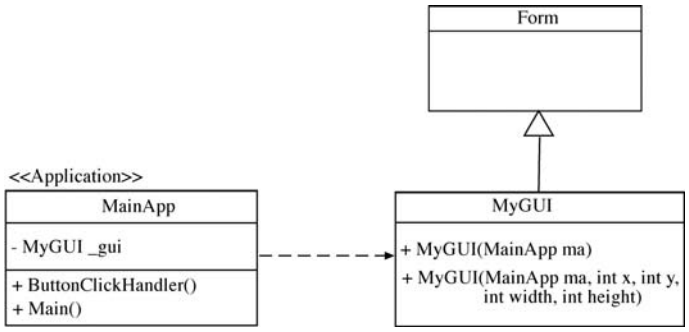

Figure 12-13: UML Class Diagram Showing Separate GUI and Application/Event Handler Classes

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 += new EventHandler(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 Ini-

tializeComponents() 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         Application.Run(_gui);
11      }
12
13      public void ButtonClickHandler(Object sender, EventArgs e){
14           _gui.LabelText = _gui.TextBoxText;
15      }
16
17      public static void Main(){
18           new MainApp();
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. It then displays the window with a call to the Application.Run() method passing in the _gui reference. All the Main() method does is create an instance of MainApp.

The ButtonClickHandler() method shown on line 13 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 though 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.

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 some 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

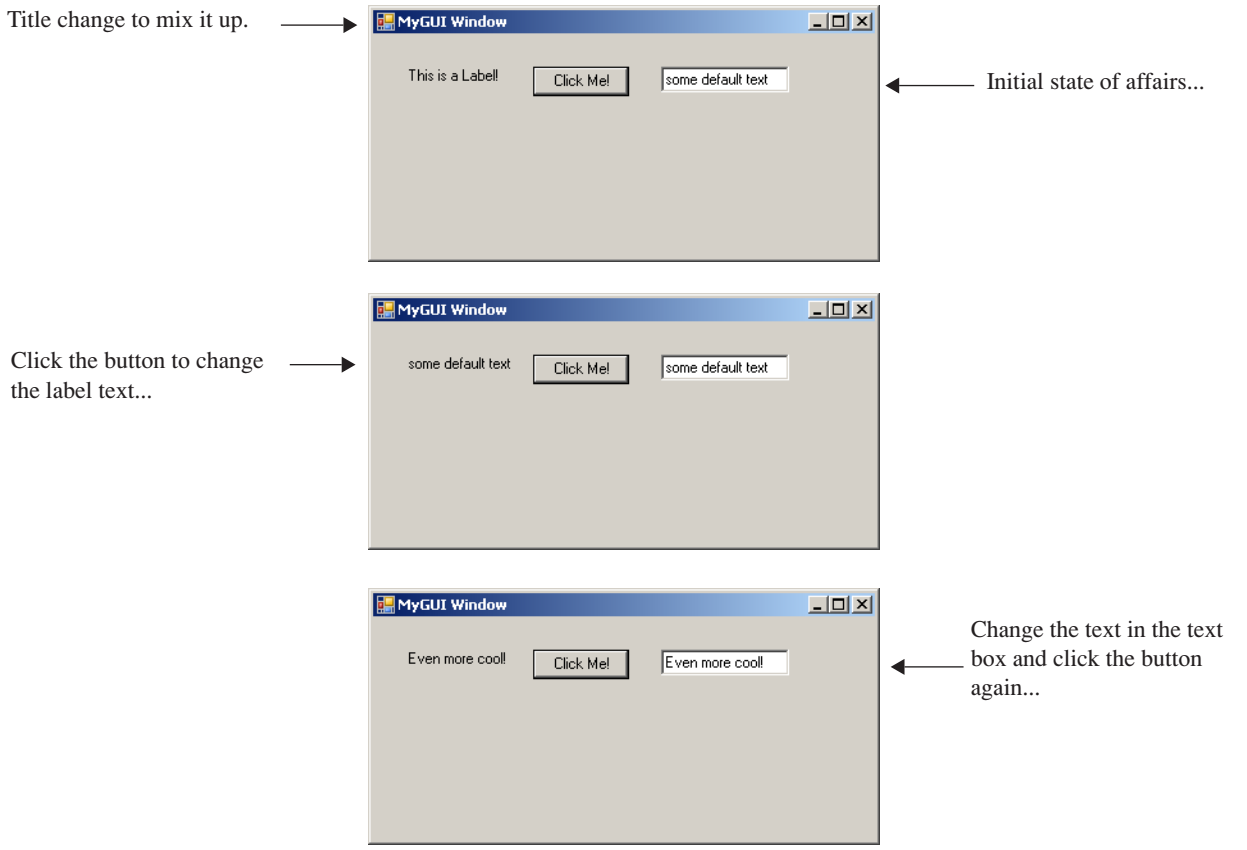

Figure 12-14: Results of Running Example 12.8 — GUI Events Handled in Separate Object

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 += new EventHandler(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.

The button array is created on like 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         Application.Run(_gui);
13      }
14
15      public void ButtonClickHandler(Object sender, EventArgs e){
16           ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
17      }
18
19      public static void Main(){
20           new MainApp();
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 FlowLayout-

GUI 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.

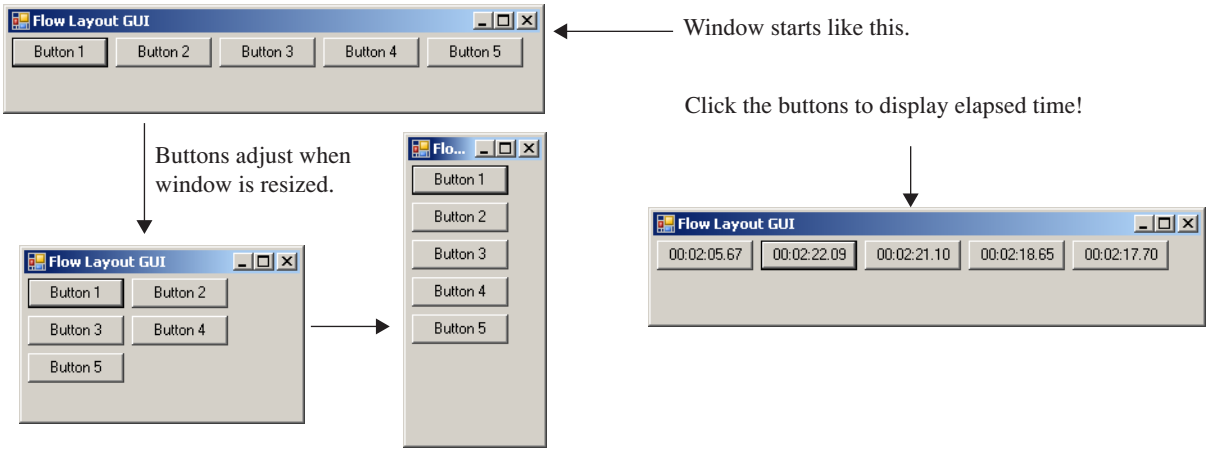

Figure 12-15: Results of Running Example 12.10 — Buttons Adjust when Window is Resized

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 example, 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

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class TableLayoutGUI : Form {

   Button[ ,] _floor = new Button[ 10,10] ;
   TableLayoutPanel _panel;

   public TableLayoutGUI(MainApp ma){
     InitializeComponents(ma);
    }

public void InitializeComponents(MainApp ma){
     _panel = new TableLayoutPanel();
     _panel.SuspendLayout();
      _panel.ColumnCount = 10;
     _panel.RowCount = 10;
     _panel.Dock = DockStyle.Top;
      _panel.AutoSize = true;
     _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;

     for(int i = 0; i<_floor.GetLength(0); i++){
       for(int j = 0; j<_floor.GetLength(1); j++){
         _floor[ i,j] = new Button();
         _floor[ i,j] .Click += new EventHandler(ma.MarkSpace);
         _panel.Controls.Add(_floor[ i,j] );
       }
     }

    this.SuspendLayout();
    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
40    } // 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          Application.Run(_gui);
13       }
14
15       public void MarkSpace(Object sender, EventArgs e){
16            ((Button)sender).BackColor = Color.Blue;
17            ((Button)sender).Image = _bitmap;
18       }
19
20       public static void Main(){
21          new MainApp();
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.

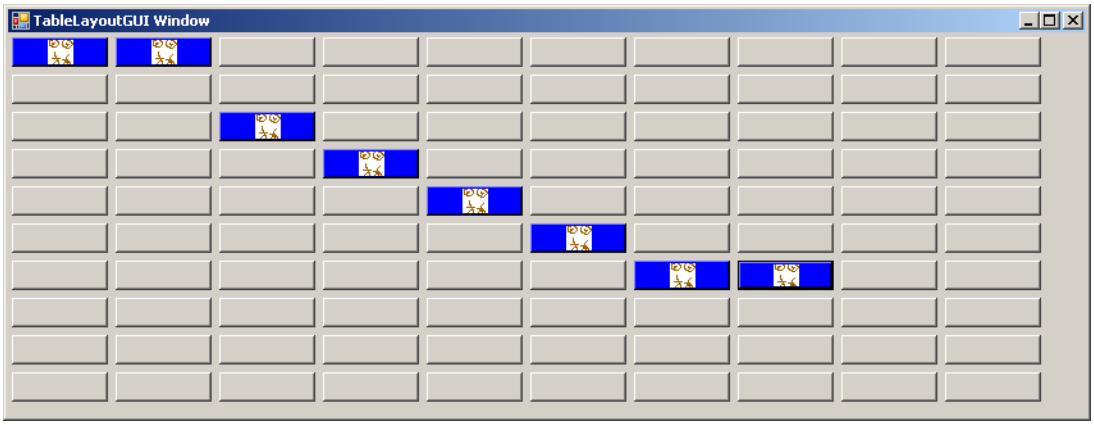

Figure 12-16: Results of Running Example 12.12 after several Buttons have been Clicked

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.

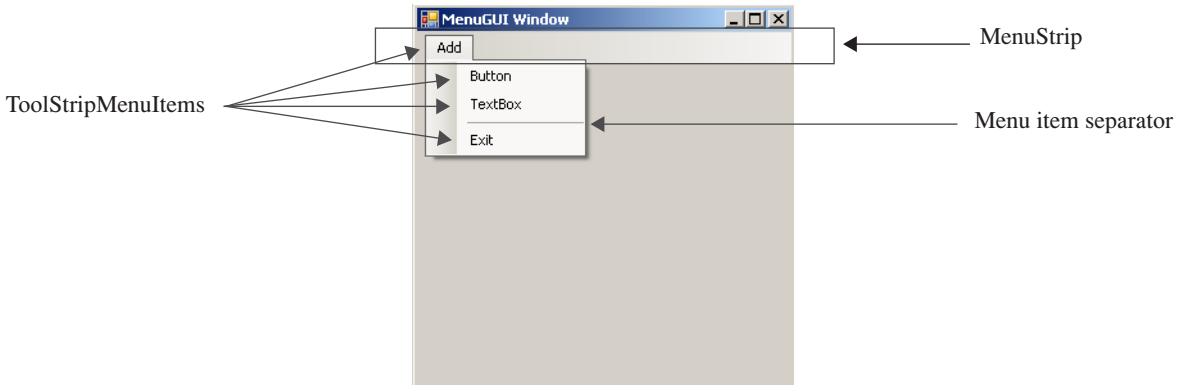

Figure 12-17: Window and Menu Structure of Menu Demo Program

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

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class MenuGUI : Form {

   private FlowLayoutPanel _panel;


   public MenuGUI(MainApp ma, int x, int y, int width, int height){
      this.Bounds = new Rectangle(x, y, width, height);
      this.Text = "MenuGUI Window";
      InitializeComponents(ma);
   }

   public MenuGUI(MainApp ma):this(ma, 125, 125, 300, 300){ }


   private void InitializeComponents(MainApp ma){
      MenuStrip ms = new MenuStrip();

      ToolStripMenuItem addMenu = new ToolStripMenuItem("Add");
      ToolStripMenuItem addButtonItem = new ToolStripMenuItem("Button", null,
                                                              new EventHandler(ma.AddButtonItemHandler));
      ToolStripMenuItem addTextBoxItem = new ToolStripMenuItem("TextBox", null,
                                                              new EventHandler(ma.AddTextBoxItemHandler));
```

```
27         ToolStripMenuItem addExitItem = new ToolStripMenuItem("Exit", null,
28                                                                new EventHandler(ma.AddExitItemHandler));
29
30         addMenu.DropDownItems.Add(addButtonItem);
31         addMenu.DropDownItems.Add(addTextBoxItem);
32         addMenu.DropDownItems.Add("-"); // <---- use a dash to add menu item separators
33         addMenu.DropDownItems.Add(addExitItem);
34
35         ms.Items.Add(addMenu);
36         ms.Dock = DockStyle.Top;
37         this.MainMenuStrip = ms;
38
39         _panel = new FlowLayoutPanel();
40        _panel.SuspendLayout();
41         _panel.AutoSize = true;
42         _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
43         _panel.WrapContents = true;
44         _panel.Dock = DockStyle.Fill;
45
46
47         this.SuspendLayout();
48         this.Controls.Add(_panel);
49         this.Controls.Add(ms); // IMPORTANT: Add the MenuStrip last !!
50         _panel.ResumeLayout();
51         this.ResumeLayout();
52
53      }
54
55      public void AddButton(MainApp ma){
56         Button b = new Button();
57         b.Text = "C# Rocks";
58         b.Click += new EventHandler(ma.ButtonClickHandler);
59         _panel.SuspendLayout();
60         _panel.Controls.Add(b);
61         _panel.ResumeLayout();
62      }
63
64
65      public void AddTextBox(MainApp ma){
66         TextBox t = new TextBox();
67         t.Text = "Default Text";
68         t.Click += new EventHandler(ma.TextBoxClickHandler);
69         _panel.SuspendLayout();
70         _panel.Controls.Add(t);
71         _panel.ResumeLayout();
72      }
73   } // 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 all of the GUI components are declared 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 20 is to declare and create the MenuStrip. Next, on lines 22 through 28, 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 22 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 30 through 33, 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 39 through 44 prepare the FlowLayoutPanel by setting several of its properties. Then, on line 47 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         Application.Run(_gui);
13      }
14
15      public void AddButtonItemHandler(object sender, EventArgs e){
16         _gui.AddButton(this);
17      }
18
19      public void AddTextBoxItemHandler(object sender, EventArgs e){
20         _gui.AddTextBox(this);
21      }
22
23      public void AddExitItemHandler(object sender, EventArgs e){
24         Application.Exit();
25      }
26
27      public void ButtonClickHandler(Object sender, EventArgs e){
28         ((Button)sender).Text = (DateTime.Now - _appStart).ToString();
29      }
30
31      public void TextBoxClickHandler(Object sender, EventArgs e){
32         ((TextBox)sender).Text = (DateTime.Now - _appStart).ToString();
33      }
34
35      public static void Main(){
36         new MainApp();
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 ButtonClick-Handler() 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.

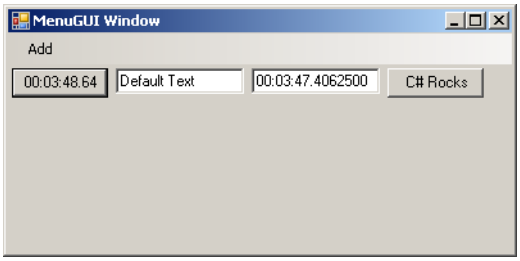

Figure 12-18: Results of Running Example 12.14 and Adding Several Buttons and Text Boxes

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
22        _flowLayoutPanel = new FlowLayoutPanel();
23        _flowLayoutPanel.SuspendLayout();
24        _flowLayoutPanel.Height = 56;
25        _flowLayoutPanel.Width = 20;
26        _flowLayoutPanel.AutoSize = true;
27        _flowLayoutPanel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
28        _flowLayoutPanel.WrapContents = true;
29        _flowLayoutPanel.Dock = DockStyle.Top;
30
31        _tb1 = new TextBox();
32        _tb1.Multiline = true;
33        _tb1.Height = 150;
34        _tb1.Width = 200;
35        _tb1.DoubleClick += new EventHandler(ma.DoubleClickHandler);
36
37        _tb2 = new TextBox();
38         _flowLayoutPanel.Controls.Add(_tb1);
39         _flowLayoutPanel.Controls.Add(_tb2);
40
41        this.SuspendLayout();
42        this.Controls.Add(_flowLayoutPanel);
43
44        _flowLayoutPanel.ResumeLayout();
45        this.ResumeLayout();
46      }
47
48      public void ShowLineNumber(){
49          int index = _tb1.SelectionStart;
50          int line_number = _tb1.GetLineFromCharIndex(index);
51         _tb2.Text = ("Line Number is: " + line_number);
52      }
53
54   } // end LineSelectGUI class definition
55
```

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 29, the MainApp.DoubleClickHandler() method is registered with _tb1's Click event.

The second TextBox is created on line 37, and on lines 38 and 39 both TextBoxes are added to the FlowLayoutPanel.

The LineSelectGUI class defines a method named ShowLineNumber() starting on line 48. 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         Application.Run(_gui);
11      }
12
13      public void DoubleClickHandler(Object sender, EventArgs args){
14         _gui.ShowLineNumber();
15      }
16
17      public static void Main(){
18         new MainApp();
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 and passes it as an argument to the Application.Run() method. 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.

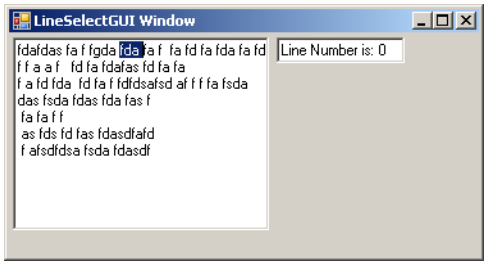

Figure 12-19: Results of Running Example 12.16 — Double-clicking the First Line

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.

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 upon the window.
- Register event handlers.
- Set other component properties as required.
- Add components to panels.
- Add panels to a form.

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.

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) 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 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.

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 the MSDN website 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 the MSDN website 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 the MSDN website 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 FlowLayout-

Panels 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.

Suggested Projects

1. **Programming:** 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 TableLayoutGrid. 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:** Write a program that mimics the operation of a handheld 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.

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 differ-

ent 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?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes