CHAPTER 12

Graphical User Interfaces

CHAPTER CONTENTS

Introduction

Many applications we use every day have a graphical user interface, or GUI (pronounced goo-ey), which allows us to control the operation of the program visually. For example, web browsers, such as Microsoft Internet Explorer, provide menus, buttons, drop-down lists, and other visual input and output devices to allow the user to communicate with the program through mouse clicks, mouse movements, and by typing text into boxes.

GUIs enable the user to select the next function to be performed, to enter data, and to set program preferences, such as colors or fonts. GUIs also make a program easier to use because a GUI is a familiar interface to users. If you design your program's GUI well, users can quickly learn to operate your program, in many cases, without consulting documentation or requiring extensive training.

Java is very rich in GUI classes. In this chapter, we will present some of those classes, along with the main concepts associated with developing a GUI application.

12.1 GUI Applications Using *JFrame*

The first step in developing a GUI application is to create a window. We've opened graphical windows in our applets, but so far, we have not displayed a window in our applications. Instead, we've communicated with users through the Java console and dialog boxes. We will use the *JFrame* class in the *javax.swing* package to create a window for our GUI applications. The inheritance hierarchy for the *JFrame* class is shown in Figure 12.1.

REFERENCE POINT

Inheritance, including which class members are inherited, is discussed in Chapter 10.

Inheriting directly from *Object* is the *Component* class, which represents a graphical object that can be displayed. Inheriting from the *Component* class is the *Container* class, which, as its name indicates, can hold other objects. This is good because we will want to add GUI components to our window. The *Window* class, which inherits from the *Container* class, can be used to create a basic window. The *Frame* class, which inherits from the *Window* class, adds a title bar and a border to the window. The title bar contains icons that allow the user to minimize, maximize, resize, and close the window. This is the type of window we are accustomed to seeing.

We need to step down one more level of inheritance, however, to the *JFrame* class before we are ready to create our GUI application. A *JFrame* object is a *swing* component. Thus, the *JFrame* class provides the functionality of *Frame*, as well as support for the *swing* architecture. We'll explain more about the *swing* components later in this chapter. For now, Figure 12.1 shows us that a *JFrame* object is a *Component*, a *Container*, a *Window*, and a *Frame* by inheritance, so we can use a *JFrame* object to display a window that will hold our GUI components, and thus present our GUI to the user. Our typical GUI application class will be a subclass of *JFrame*, inheriting its functionality as a window and a container. Thus, the class header for our GUI applications will follow this pattern:

```
public class ClassName extends JFrame
```

Table 12.1 shows two constructors and some important methods of the *JFrame* class, some of which are inherited from its superclasses. Because our GUI applications extend the *JFrame* class, our applications inherit these methods also.

Example 12.1 shows a shell GUI application class. This class demonstrates the general format for building GUI applications, but has no components.

```
1 /*  A Shell GUI Application
2     Anderson, Franceschi
3 */
4
5 import javax.swing.JFrame;
6 import java.awt.Container;
7 // other import statements here as needed
8
9 public class ShellGUIApplication extends JFrame
10 {
11    private Container contents;
12    // declare other instance variables
13
14    // constructor
15    public ShellGUIApplication( )
16    {
17      // call JFrame constructor with title bar text
18      super( "A Shell GUI Application" );
19
```

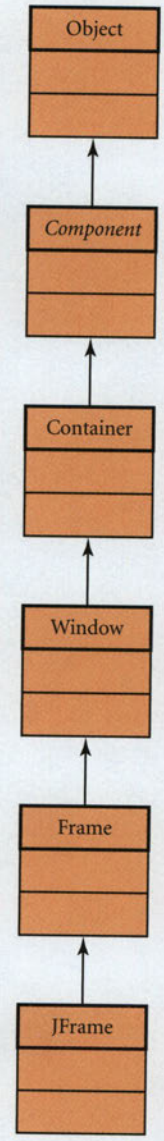


Figure 12.1
Inheritance Hierarchy for *JFrame*

TABLE 12.1 Useful *JFrame* Constructors and Methods

Constructors and Methods of the *JFrame* Class	
Class	**Constructor**
`JFrame`	`JFrame( )` constructs a *JFrame* object, initially invisible, with no text in the title bar
`JFrame`	`JFrame( String titleBarText )` constructs a *JFrame* object, initially invisible, with *titleBarText* displayed in the window's title bar
Methods	
Return value	**Method name and argument list**
`Container`	`getContentPane( )` returns the content pane object for this window
`void`	`setDefaultCloseOperation( int operation )` sets the default operation when the user closes this window, that is, when the user clicks on the X icon in the top-right corner of the window
`void`	`setSize( int width, int height )` sizes the window to the specified *width* and *height* in pixels
`void`	`setVisible( boolean mode )` displays this window if *mode* is *true*; hides the window if *mode* is *false*

```
20      // get container for components
21      contents = getContentPane( );
22
23      // set the layout manager
24
25      // instantiate GUI components and other instance variables
26
27      // add GUI components to the content pane
28
```

```
29      // set original size of window
30      setSize( 300, 200 );
31
32      // make window visible
33      setVisible( true );
34    }
35
36    public static void main( String [ ] args )
37    {
38      ShellGUIApplication basicGui = new ShellGUIApplication( );
39      basicGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
40    }
41  }
```

EXAMPLE 12.1 A Shell GUI Application Using *JFrame*

We have defined one instance variable, *contents*, which is a *Container* object reference (line 11). Our subsequent GUI applications will define instance variables for the components to be placed in the window, as well as instance variables to hold the data of the application as primitive data types or objects.

The *main* method is coded at lines 36–40. We start by instantiating an object of this application class (line 38), which invokes the constructor (lines 14–34).

At line 39, we call the *setDefaultCloseOperation* method. The *EXIT_ON_CLOSE* argument is a useful *static* constant of the *JFrame* class, which specifies that the GUI application should terminate when the user closes the window.

A constructor in a GUI application has several tasks to perform:

- Call the constructor of the *JFrame* superclass.
- Get an object reference to the content pane container. We will add our GUI components to the content pane.
- Set the layout manager. Layout managers arrange the GUI components in the window.
- Instantiate each component.
- Add each component to the content pane.
- Set the size of the window.
- Display the window.

In our constructor, at line 18, we call the *JFrame* constructor, passing as an argument, a *String* representing the text that we want to be displayed in the title bar of the window. As in any subclass, this must be the first statement in the constructor.

At line 21, we call the *getContentPane* method, inherited from the *JFrame* class, and assign its return value to *contents*. The *getContentPane* method returns a *Container* object reference; subsequent applications will add our GUI components to this container.

Although this shell application does not have any GUI components, we have inserted comments to indicate where the component-related operations should be performed. Later in this chapter, we show you how to perform these actions.

We then set the initial size of the window to a width of 300 pixels and a height of 200 pixels by calling the *setSize* method (line 30). Finally, we call the *setVisible* method (line 33) so that the window will be displayed when the constructor finishes executing. Calling the *setSize* and *setVisible* methods is not required, but if you omit either method call, you will get unfavorable results. If you omit the call to *setSize*, the window will consist of just the title bar. If you omit the call to *setVisible*, the window will not be displayed when the application begins.

The window generated by running Example 12.1 is shown in Figure 12.2. Note the text in the title bar, which was set by calling the *JFrame* constructor. When you run the program, try minimizing, maximizing, moving, resizing, and closing the window.

We have opened a window successfully, but an empty window is not very impressive. In the next section, we will show you how to start adding GUI components to the window.

COMMON ERROR TRAP

Be sure to call the *setSize* method to set the initial dimensions of the window and call the *setVisible* method to display the window and its contents. Omitting the call to the *setSize* method will create a default *JFrame* consisting of a title bar only, although it will be resizable by the user. If you omit the call to the *setVisible* method, the window will not open when the application begins.

Figure 12.2
The Window Generated by Example 12.1

12.2 GUI Components

We have already said that a **component** is an object having a graphical representation. Labels, text fields, buttons, radio buttons, checkboxes, and drop-down lists are examples of components. A component performs at least one of these functions:

- displays information
- collects data from the user
- allows the user to initiate program functions

Java provides an extensive set of classes that can be used to add a GUI to your applications. Table 12.2 lists some GUI components and the Java classes that encapsulate them. All classes listed in Table 12.2 belong to the *javax.swing* package.

Java supports two implementations of GUI components: AWT (Abstract Window Toolkit) and *swing*.

The AWT components are the original implementation of Java components and hand off some of the display and behavior of the component to the native windowing system. In other words, on a PC, AWT components,

TABLE 12.2 Selected GUI Components and Java Classes

Component	Purpose	Java Class
Label	Displays an image or read-only text. Labels are often paired with text fields and are used to identify the contents of the text field.	*JLabel*
Text field	A single-line text box for displaying information and accepting user input.	*JTextField*
Text area	Multiple-line text field for data entry or display.	*JTextArea*
Password field	Single-line text field for accepting passwords without displaying the characters typed.	*JPasswordField*
Button	Command button that the user clicks to signal that an operation should be performed.	*JButton*
Radio button	Toggle button that the user clicks to select one option in the group. Clicking on a radio button deselects any previously selected option.	*JRadioButton*
Checkbox	Toggle button that the user clicks to select 0, 1, or more options in a group.	*JCheckBox*
List	List of items that the user clicks to select one or more items.	*JList*
Drop-down list	Drop-down list of items that the user clicks to select one item.	*JComboBox*

such as buttons and frames, are rendered by the Windows windowing system; on a Macintosh computer, AWT components are rendered by the MacOS windowing system; on a Unix computer, AWT components are rendered by the X-Window system. As a result, the AWT GUI components automatically take on the appearance and behavior (commonly called the **look and feel**) of the windowing system on which the application is running. One disadvantage of AWT components, however, is that because of the inconsistencies in the look and feel among the various windowing systems, an application may behave slightly differently from one platform to another. Because AWT components rely on the native windowing system, they are often called **heavyweight** components.

The *swing* components are the second generation of GUI components, developed entirely in Java to provide a consistent look and feel from platform to platform. As such, they are often referred to as **lightweight** components. Some of the benefits of *swing* components are:

- Applications run consistently across platforms, which makes maintenance easier.
- The *swing* architecture has its roots in the model-view-controller paradigm, which facilitates programming:
  - the model represents the data for the application
  - the view is the visual representation of that data
  - the controller takes user input on the view and updates the model accordingly
- The *swing* components can be rendered in multiple windowing system styles, so your application can take on the look and feel of the platform on which it is running, if desired.

Given the advantages of *swing* over AWT, we will build our GUI applications using *swing* components exclusively. You can recognize a *swing* component because its class name begins with *J*. Thus, *JFrame* is a *swing* component. *Swing* components are found in the *javax.swing* package.

A summary of the inheritance hierarchy for selected *swing* components is shown in Figure 12.3. *JComponent* is the base class for all swing components, except for top-level containers such as *JFrame*. Notice that the *JFrame* class does not inherit from the *JComponent* class. However, the *JFrame* class and the *JComponents* shown in Table 12.2 do share some common superclasses: *Component* and *Container*. Thus, the *JComponents* are

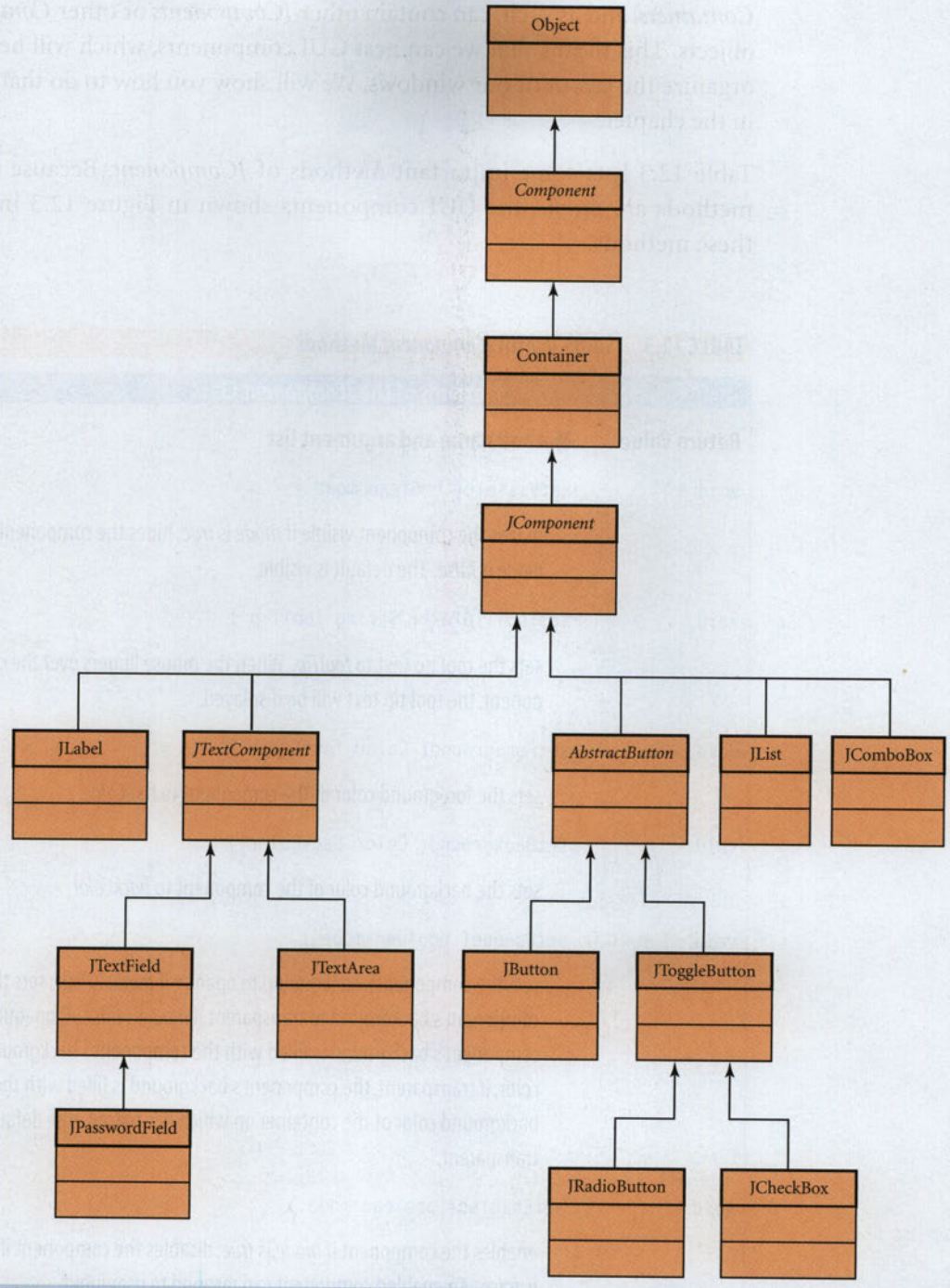


Figure 12.3
The Inheritance Hierarchy for Some GUI Classes

Containers, and as such, can contain other *JComponents* or other *Container* objects. This means that we can nest GUI components, which will help us organize the layout of our windows. We will show you how to do that later in the chapter.

Table 12.3 lists some important methods of *JComponent*. Because these methods are *public*, the GUI components shown in Figure 12.3 inherit these methods.

TABLE 12.3 Some Useful *JComponent* Methods

Methods of the *JComponent* Class	
Return value	**Method name and argument list**
`void`	`setVisible( boolean mode )` makes the component visible if *mode* is *true*; hides the component if *mode* is *false*. The default is visible.
`void`	`setToolTipText( String toolTip )` sets the tool tip text to *toolTip*. When the mouse lingers over the component, the tool tip text will be displayed.
`void`	`setForeground( Color foreColor )` sets the foreground color of the component to *foreColor*.
`void`	`setBackground( Color backColor )` sets the background color of the component to *backColor*.
`void`	`setOpaque( boolean mode )` sets the component's background to opaque if *mode* is *true*; sets the component's background to transparent if *mode* is *false*. If opaque, the component's background is filled with the component's background color; if transparent, the component's background is filled with the background color of the container on which it is placed. The default is transparent.
`void`	`setEnabled( boolean mode )` enables the component if *mode* is *true*; disables the component if *mode* is *false*. An enabled component can respond to user input.

12.3 A Simple Component: *JLabel*

Now it's time to place a component into the window. We will start with a simple component, a label, encapsulated by the *JLabel* class. A user does not interact with a label; the label just displays some information, such as a title, an identifier for another component, or an image.

Example 12.2 creates a window containing two labels, one that displays text and one that displays an image. Figure 12.4 shows the window when the application is run.

```
1 /*  Using JLabels to display text and images
2     Anderson, Franceschi
3 */
4
5 import java.awt.Container;
6 import javax.swing.JFrame;
7 import javax.swing.ImageIcon;
8 import javax.swing.JLabel;
9 import java.awt.FlowLayout;
10 import java.awt.Color;
11
12 public class Dinner extends JFrame
13 {
14   private Container contents;
15   private JLabel labelText;
16   private JLabel labelImage;
17
18   // Constructor
19   public Dinner( )
20   {
21     super( "What's for dinner?" );          // call JFrame constructor
22
23     contents = getContentPane( );           // get content pane
24
25     contents.setLayout( new FlowLayout( ) ); // set layout manager
26
27     // use the JLabel constructor with a String argument
28     labelText = new JLabel( "Sushi tonight?" );
29
30     // set label properties
31     labelText.setForeground( Color.WHITE );
32     labelText.setBackground( Color.BLUE );
33     labelText.setOpaque( true );
```

```
34
35      // use the JLabel constructor with an ImageIcon argument
36      labelImage = new JLabel( new ImageIcon( "sushi.jpg" ) );
37
38      // set tool tip text
39      labelImage.setToolTipText( "photo of sushi" );
40
41      // add the two labels to the content pane
42      contents.add( labelText );
43      contents.add( labelImage );
44
45      setSize( 300, 200 );
46      setVisible( true );
47    }
48
49    public static void main( String [ ] args )
50    {
51      Dinner dinner = new Dinner( );
52      dinner.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
53    }
54  }
```

EXAMPLE 12.2 A Simple GUI Component: *JLabel*

Java provides classes to help us organize window contents. These classes are called layout managers. They implement the *LayoutManager* interface and determine the size and position of the components within a container. We will use the simple *FlowLayout* layout manager for the next few examples; later in the chapter, we will look into two other, more sophisticated layout managers: *BorderLayout* and *GridLayout*.

The *FlowLayout* layout manager arranges components in rows from left to right in the order in which the components are added to the container.

Figure 12.4
Running Example 12.2

Whenever a newly added component does not fit into the current row, the *FlowLayout* layout manager starts a new row. With *FlowLayout*, if the user resizes the window, the components may be rearranged into fewer or more rows. We will use the default constructor of the *FlowLayout* class, which centers the components within each row.

To add components to a window, we use two methods of the *Container* class, shown in Table 12.4, to set the layout manager and add components to the window. Remember that we obtain the content pane of the *JFrame* window as a *Container* object. We call these methods using the content pane object reference.

In Example 12.2, we declare two *JLabel* instance variables at lines 15–16. At line 25, we set *FlowLayout* as the layout manager.

Next, we instantiate our *JLabel* objects. Table 12.5 shows several constructors of the *JLabel* class.

The *SwingConstants* interface, which is implemented by many *JComponents* that display text, such as labels, text fields, and buttons, provides a collection of *static int* constants that can be used for positioning and orientation of the text within a component.

REFERENCE POINT

More information about the *SwingConstants* interface can be found on the Oracle Java website: *www.oracle.com/technetwork/java*.

The first *JLabel* object reference, *labelText*, is instantiated at line 28, using the constructor with a *String* argument.

The second *JLabel* object reference, *labelImage*, is instantiated at line 36, using the constructor that takes an *Icon* argument. The *ImageIcon* class

TABLE 12.4 Useful Methods of the *Container* Class

Methods of the *Container* Class	
Return value	**Method name and argument list**
`void`	`setLayout( LayoutManager mgr )` sets the layout manager of the window to *mgr*
`Component`	`add( Component component )` adds the *component* to the container, using the rules of the layout manager. Returns the argument
`void`	`removeAll( )` removes all components from the window

TABLE 12.5 Useful Constructors of the *JLabel* Class

Constructors of the *JLabel* Class	
Class	**Constructor**
`JLabel`	`JLabel( String text )` creates a *JLabel* object that displays the specified *text*.
`JLabel`	`JLabel( String text, int alignment )` creates a *JLabel* object that displays the specified *text*. The *alignment* argument specifies the alignment of the text within the label component. The *alignment* value can be any of the following *static int* constants from the *SwingConstants* interface: LEFT, CENTER, RIGHT, LEADING, or TRAILING. By default, the label text is left-adjusted.
`JLabel`	`JLabel( Icon image )` creates a *JLabel* object that displays the *image*.

encapsulates an image and implements the *Icon* interface, so an *IconImage* object can be used wherever an *Icon* argument is required. One of the *ImageIcon* constructors takes as a *String* argument the file name where the image is stored. Its API is:

```
public ImageIcon( String filename )
```

COMMON ERROR TRAP

As for any object reference, it is mandatory to instantiate a component before using it. Forgetting to instantiate a component before adding it to the content pane will result in a *NullPointerException* at run time.

Lines 31–33 call the *setForeground*, *setBackground*, and *setOpaque* methods inherited from the *JComponent* class to set some properties of *labelText*. On line 39, we set up a tool tip that will pop up the message "photo of sushi" when the user pauses the mouse pointer over the image. You can see this tool tip displayed in Figure 12.4.

In lines 42–43, we add the two labels to the window. Because we are using the *FlowLayout* layout manager, the labels appear in the order we added them to the component, first *labelText*, then *labelImage*, centered left to right in the window. If the user resizes the window to be about the same width as the *JLabel labelText*, the layout manager will arrange the display so that the image appears under the text (as a second row).

Note that the last statement of the constructor (line 46) is a call to the *setVisible* method to make the window visible.

Skill Practice
with these end-of-chapter questions

12.18.1 Multiple Choice Exercises

Questions 1, 2, 3, 4

12.18.2 Reading and Understanding Code

Question 18

12.18.3 Fill In the Code

Question 27

12.18.4 Identifying Errors in Code

Question 42

12.18.5 Debugging Area

Questions 46, 47

12.4 Event Handling

Now we know how to open a window in an application, and we know how to display labels and images. But the user can't interact with our application yet, except to display a tool tip. We need to add some interactive GUI components, like buttons or text fields or lists. By interacting with these components the user will control which operations of the program will take place, and in what order.

GUI programming uses an **event-driven model** of programming, as opposed to the procedural model of programming we have been using thus far. By that, we mean that by using a GUI, we put the user in interactive control. For example, we might display some text fields, a few buttons, and a selectable list of items. Then our program will "sit back" and wait for the user to do something. When the user enters data into a text field, presses a button, or selects an item from the list, our program will respond, perform the function that the user has requested, then sit back again and wait for the user to do something else. These user actions generate **events**. Thus, the

processing of our program will consist of responding to events caused by the user interacting with GUI components.

When the user interacts with a GUI component, the component **fires an event**. Java provides interfaces, classes, and methods to handle these events. Using these Java tools, we can register our application's interest in being notified when an event occurs for a particular component, and we can specify the code we want to be executed when that event occurs.

To allow a user to interact with our application through a GUI component, we need to perform the following functions:

1. write an event handler class (called a **listener**)
2. instantiate an object of that class
3. register that listener on one or more components

A typical event handler class *implements* a listener interface. The listener interfaces, which inherit from the *EventListener* interface, are supplied in the *java.awt.event* and *javax.swing.event* packages. A listener interface specifies one or more *abstract* methods that an event handler class needs to implement. The listener methods receive as a parameter an event object, which represents the event that was fired.

An application can instantiate multiple event handlers, and a single event handler can be the listener for multiple components.

Event classes are subclasses of the *EventObject* class, as shown in Figure 12.5, and are in the *java.awt.event* and *javax.swing.event* packages. From the *EventObject* class, the event classes inherit the *getSource* method, which returns the object reference of the component that fired the event. Its API is shown in Table 12.6.

Thus, with some simple *if/else* statements, an event handler that is registered as the listener for more than one component can identify which of the components fired the event and decide on the appropriate action.

The types of listeners that we can register on a component depend on the types of events that the component fires. Table 12.7 shows some user activities that generate events, the type of event object created, and the appropriate listener interface for the event handler to implement. Some of these components can fire other events, as well.

To add event handling to our *ShellGUIApplication* class, we will follow the following pattern. The code we need to add is shown in bold.

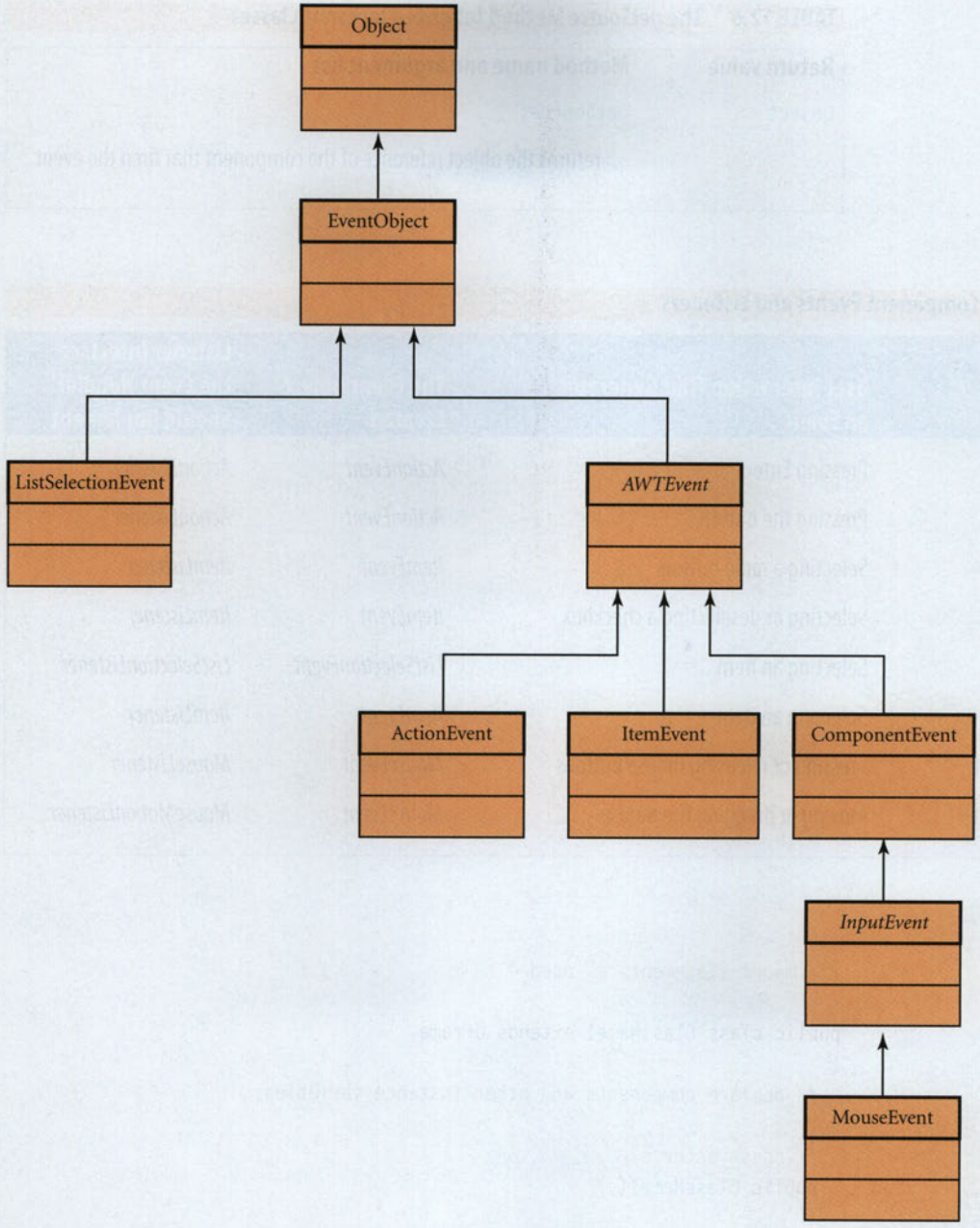


Figure 12.5
Event Class Hierarchy

TABLE 12.6 The *getSource* Method Inherited by Event Classes

Return value	Method name and argument list
`Object`	`getSource( )` returns the object reference of the component that fired the event

TABLE 12.7 Component Events and Listeners

JComponent	User Interaction That Generates an Event	The Event Object Created	Listener Interface the Event Handler Should Implement
JTextField	Pressing Enter	*ActionEvent*	*ActionListener*
JButton	Pressing the button	*ActionEvent*	*ActionListener*
JRadioButton	Selecting a radio button	*ItemEvent*	*ItemListener*
JCheckBox	Selecting or deselecting a checkbox	*ItemEvent*	*ItemListener*
JList	Selecting an item	*ListSelectionEvent*	*ListSelectionListener*
JComboBox	Selecting an item	*ItemEvent*	*ItemListener*
Any component	Pressing or releasing mouse buttons	*MouseEvent*	*MouseListener*
Any component	Moving or dragging the mouse	*MouseEvent*	*MouseMotionListener*

```
// import statements as needed

public class ClassName1 extends JFrame
{
  // declare components and other instance variables

  // constructor
  public ClassName1( )
  {
     // call JFrame constructor
     // get content pane
     // set the layout manager
     // instantiate components and other instance variables
     // add components to the content pane
     // declare and instantiate event handler objects
```

```
        // register event handlers on components

        // set window size
        // make window visible
    }

    // private event handler class
    private class EventHandlerName implements ListenerName
    {
        // implement the methods of the listener interface
        // to process the events
    }

    public static void main( String [ ] args )
    {
        // instantiate application object
    }
}
```

Although we can define an event handler class as *public* in its own source file, we usually declare an event handler as a ***private* inner class** within the GUI application class. A *private* inner class is defined within the *public* class and has access to all the members of the *public* class. Thus, declaring our event handler as a *private* inner class simplifies our code by giving the event handler direct access to our application's GUI components.

In the constructor, we instantiate an object of our event handler class. Then we register that event handler on a component by calling an *add...Listener* method. Table 12.8 shows a few of these methods. We call the *add...Listener* method using the object reference of the component on which we want to register the listener and passing as an argument the event handler object we instantiated.

TABLE 12.8 Some *add...Listener* Methods

add...Listener Method APIs
`void addActionListener( ActionListener handler )`
`void addItemListener( ItemListener handler )`
`void addListSelectionListener( ListSelectionListener handler )`
`void addMouseListener( MouseListener handler )`
`void addMouseMotionListener( MouseMotionListener handler )`

12.5 Text Fields

Now we are ready for our first GUI program with user interaction. For this first example, we will build a GUI that takes an ID and a password from a user. If the ID is "open" and the password is "sesame", we display "Welcome!" Otherwise, we display "Sorry: wrong login".

We will use a *JTextField* for the ID and a *JPasswordField* for the password. For confidentiality of passwords, the *JPasswordField* does not display the characters typed by the user. Instead, as each character is typed, the *JPasswordField* displays an echo character. The default echo character is *, but we can specify a different character, if desired. We will also use a *JTextArea* to display a legal warning to potential hackers. Both *JTextField* and *JTextArea* are direct subclasses of *JTextComponent*, which encapsulates text entry from the user. *JTextField* displays a single-line field, and *JTextArea* displays a multiline field.

Table 12.9 lists some useful constructors and methods of these text classes.

TABLE 12.9 Useful Constructors and Methods of *JTextField*, *JTextArea*, and *JPasswordField*

Constructors and Methods of the *JTextField*, *JTextArea*, and *JPasswordField* Classes	
Constructors	
Class	**Constructor**
`JTextField`	`JTextField( String text, int numColumns )` constructs a text field initially filled with *text*, with the specified number of columns
`JTextField`	`JTextField( int numberColumns )` constructs an empty text field with the specified number of columns
`JTextArea`	`JTextArea( String text )` constructs a text area initially filled with *text*
`JTextArea`	`JTextArea( int numRows, int numColumns )` constructs an empty text area with the number of rows and columns specified by *numRows* and *numColumns*
`JTextArea`	`JTextArea( String text, int numRows, int numColumns )` constructs a text area initially filled with *text*, and with the number of rows and columns specified by *numRows* and *numColumns*
`JPasswordField`	`JPasswordField( int numberColumns )` constructs an empty password field with the specified number of columns

TABLE 12.9 (continued)

Methods Common to *JTextField*, *JTextArea*, and *JPasswordField* Classes	
Return value	**Method name and argument list**
`void`	`setEditable( boolean mode )` sets the properties of the text component as editable or noneditable, depending on whether *mode* is *true* or *false*. The default is editable.
`void`	`setText( String newText )` sets the text of the text component to *newText*.
`String`	`getText( )` returns the text contained in the text component.
Additional Methods of the *JPasswordField* Class	
Return value	**Method name and argument list**
`void`	`setEchoChar( char c )` sets the echo character of the password field to *c*.
`char [ ]`	`getPassword( )` returns the text entered in this password field as an array of *chars*. Note: This method is preferred over the *getText* method for getting the password typed by the user.

Example 12.3 implements our login application, and Figure 12.6 shows the application in action.

```
1 /* Using JTextFields, JTextArea, and JPasswordField
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JFrame;
6 import javax.swing.JLabel;
7 import javax.swing.JTextField;
8 import javax.swing.JPasswordField;
9 import javax.swing.JTextArea;
10 import java.awt.Container;
11 import java.awt.FlowLayout;
12 import java.awt.Color;
13 import java.awt.event.ActionListener;
14 import java.awt.event.ActionEvent;
15
```

```
16 public class Login extends JFrame
17 {
18  private Container contents;
19  private JLabel idLabel, passwordLabel, message;
20  private JTextField id;
21  private JPasswordField password;
22  private JTextArea legal;
23
24  // Constructor
25  public Login( )
26  {
27   super( "Login Screen" );
28   contents = getContentPane( );
29   contents.setLayout( new FlowLayout( ) );
30
31   idLabel = new JLabel( "Enter id" ); // label for ID
32   id = new JTextField( "", 12 );      // instantiate ID text field
33
34   passwordLabel = new JLabel( "Enter password" ); // password label
35   password = new JPasswordField( 8 ); // instantiate password field
36   password.setEchoChar( '?' );        // set echo character to '?'
37
38   message = new JLabel( "Log in above" );  // label to hold messages
39
40   // instantiate JTextArea with legal warning
41   legal = new JTextArea( "Warning: Any attempt to illegally\n"
42            + "log in to this server is punishable by law.\n"
43            + "This corporation will not tolerate hacking,\n"
44            + "virus attacks, or other malicious acts." );
45   legal.setEditable( false );        // disable typing in this field
46
47   // add all components to the window
48   contents.add( idLabel );
49   contents.add( id );
50   contents.add( passwordLabel );
51   contents.add( password );
52   contents.add( message );
53   contents.add( legal );
54
55   // instantiate event handler for the text fields
56   TextFieldHandler tfh = new TextFieldHandler( );
57
58   // add event handler as listener for ID and password fields
59   id.addActionListener( tfh );
```

```
60   password.addActionListener( tfh );
61
62   setSize( 250, 200 );
63   setVisible( true );
64  }
65
66  // private inner class event handler
67  private class TextFieldHandler implements ActionListener
68  {
69   public void actionPerformed( ActionEvent e )
70   {
71    if ( id.getText( ).equals( "open" )
72     && ( new String( password.getPassword( ) ) ).equals( "sesame" ) )
73    {
74     message.setForeground( Color.BLACK );
75     message.setText( "Welcome!" );
76    }
77    else
78    {
79     message.setForeground( Color.RED );
80     message.setText( "Sorry: wrong login" );
81    }
82   }
83  }
84
85  public static void main( String [ ] args )
86  {
87   Login login = new Login( );
88   login.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
89  }
90 }
```

EXAMPLE 12.3 The *Login* Class

Lines 5–14 import the various classes that we use in this program. Line 16 declares the class, which we name *Login*, as a subclass of *JFrame*.

We declare our instance variables at lines 18–22. In addition to our three components (the *JTextField id*, the *JPasswordField password*, and the *JTextArea legal*), we also have three *JLabel* components (*idLabel*, *passwordLabel*, and *message*). The *idLabel* and *passwordLabel* components provide descriptions for the corresponding text fields. We will use the *message* label to display feedback to the user on whether the login was successful.

Figure 12.6
Running Example 12.3

At line 32, we instantiate the *JTextField id* using the constructor that takes an initial value and the size of the text field. The initial value is blank, and we want the text field to accommodate 12 characters. At line 35, we instantiate the *JPasswordField password* and set its length to 8 characters. At line 36, we call the *setEchoChar* method to reset the echo character from its default of * to ?. Thus, when the user types characters into the password field, a ? will be displayed for each character typed.

At lines 40–44, we instantiate the *JTextArea legal*. By default, a user can enter data into a text area. However, since we are using this text area only to

display a legal warning, we turn off the ability of the user to type into the text area by calling the *setEditable* method with the argument *false* (line 45).

At lines 47–53, we add all the components to the content pane. As Figure 12.6 shows, the components will be displayed in the window in the order we added them to the container.

After the window is displayed, our application will wait for the user to interact with the components. When the user enters a value into either the *id* or the *password* field, we want to check whether the values the user entered are correct. This will be the job of the event handler for those fields.

Pressing the *Enter* key in a text field generates an *ActionEvent*; therefore, at lines 66–83, we coded our event handler as the *private* inner class, *TextFieldHandler*, which *implements* the *ActionListener* interface. As shown in Table 12.10, the *ActionListener* interface has only one *abstract* method, *actionPerformed*, which receives an *ActionEvent* parameter.

Be sure that you code the *actionPerformed* method correctly, as shown in Table 12.10. Failure to code the header properly causes a compiler error. For example, misspelling the method name generates the following compiler error:

```
Login.TextFieldHandler is not abstract and does not override abstract
method actionPerformed(ActionEvent) in ActionListener
private class TextFieldHandler implements ActionListener
              ^
1 error
```

COMMON ERROR TRAP

Be sure that the header of the *actionPerformed* method is coded correctly. Otherwise, your method will not override the *abstract actionPerformed* method, as required by the *ActionListener* interface.

In the constructor, we declare and instantiate a *TextFieldHandler* object reference, *tfh* (line 55–56). Then at lines 58–60, we register *tfh* as the listener on the *id* and *password* components. When either *id* or *password* is in focus (that is, if the mouse was last clicked on the component) and the user

TABLE 12.10 *ActionListener* Method API

`public void actionPerformed( ActionEvent event )`
An event handler that implements the *ActionListener* interface writes code in this method to respond to the *ActionEvent* fired by any registered components.

presses the *Enter* key, the component that is in focus will fire an action event and our *actionPerformed* method will be executed.

We implement the *actionPerformed* method at lines 69–82. Because we defined this event handler as a *private* inner class, the *actionPerformed* method can directly access the *id* and *password* fields. At lines 71–72, we test if the ID and password entered by the user are correct.

As shown in Table 12.9, the *getPassword* method of the *JPasswordField* class returns an array of *chars*. At line 72, to convert that array of *chars* to a *String*, we use a constructor of the *String* class with the following API, which takes an array of *chars* as an argument:

```
public String( char [ ] charArray )
```

If the user has typed correct values into both the *id* and *password* fields, our event handler sets the foreground color to black and sets the text of *message* to "Welcome!" (lines 74–75). If either the id or the password is incorrect, we set the foreground color to red and set the text of *message* to "Sorry: wrong login" (lines 79–80).

Note that we did not register our event handler for the *JTextArea*. We have disabled typing into that field, so the user cannot enter new text into that field. Also, if we did not register a listener on the *id* and *password* components, the user could still type characters into these two fields and press the *Enter* key, but no events would fire, and therefore, our application would not respond to the user's actions.

COMMON ERROR TRAP

If you do not register a listener on a component, the component will not fire an event. Thus, the event handler will not execute when the user interacts with the component.

Figure 12.6 shows the application when the window first opens, after the user enters incorrect login values, and after the user enters correct login values. We suggest you become familiar with this application. First run the application as it is written. Then, delete either line 59 or 60, which registers the listeners on the components, and test the application again. Finally, change the echo character and test the application again.

12.6 Command Buttons

Handling an event associated with a button follows the same pattern as Example 12.3. We instantiate an event handler and add that event handler as a listener for the button. Clicking on a button generates an *ActionEvent*, so our listener needs to implement the *ActionListener* interface.

In Example 12.4, we present the user with a text field and two command buttons. The user enters a number into the text field, then presses one of

the two buttons. When the user clicks on the first button, we square the number; when the user clicks on the second button, we cube the number. We then display the result using a label component. To determine whether to square or cube the number, our event handler will need to identify which button was clicked by calling the *getSource* method of the *ActionEvent* class, as described in Table 12.6.

This example will demonstrate the following:

- how to handle an event originating from a button
- how to determine which component fired the event

```
1 /* Simple Math Operations Using JButtons
2    Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 public class SimpleMath extends JFrame
10 {
11  private Container contents;
12  private JLabel operandLabel, resultLabel, result;
13  private JTextField operand;
14  private JButton square, cube;
15
16  public SimpleMath( )
17  {
18   super( "Simple Math" );
19   contents = getContentPane( );
20   contents.setLayout( new FlowLayout( ) );
21
22   operandLabel = new JLabel( "Enter a number" ); // text field label
23   operand = new JTextField( 5 ); // text field is 5 characters wide
24
25   // instantiate buttons
26   square = new JButton( "Square" );
27   cube = new JButton( "Cube" );
28
29   resultLabel = new JLabel( "Result:" ); // label for result
30   result = new JLabel( "???" );          // label to hold result
31
32   // add components to the window
```

```
33   contents.add( operandLabel );
34   contents.add( operand );
35   contents.add( square );
36   contents.add( cube );
37   contents.add( resultLabel );
38   contents.add( result );
39
40   // instantiate our event handler
41   ButtonHandler bh = new ButtonHandler( );
42
43   // add event handler as listener for both buttons
44   square.addActionListener( bh );
45   cube.addActionListener( bh );
46
47   setSize( 175, 150 );
48   setVisible( true );
49  }
50
51  // private inner class event handler
52  private class ButtonHandler implements ActionListener
53  {
54   // implement actionPerformed method
55   public void actionPerformed( ActionEvent ae )
56   {
57    try
58    {
59     double op = Double.parseDouble( operand.getText( ) );
60
61     // identify which button was pressed
62     if ( ae.getSource( ) == square )
63       result.setText( ( new Double( op * op ) ).toString( ) );
64     else if ( ae.getSource( ) == cube )
65       result.setText( ( new Double( op * op * op ) ).toString( ) );
66    }
67    catch ( NumberFormatException e )
68    {
69     operandLabel.setText( "Enter a number" );
70     operand.setText( "" );
71     result.setText( "???" );
72    }
73   }
74  }
75
76  public static void main( String [ ] args )
77  {
```

```
78    SimpleMath sm = new SimpleMath( );
79    sm.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
80  }
81 }
```

EXAMPLE 12.4 Simple Math Using *JButton*

As you have seen in the previous examples, GUI applications use a number of Java classes. For Example 12.4, as well as the remaining examples in this chapter, we will use the bulk *import* statement, which replaces the individual class names with the wildcard character (*). Its syntax is:

```
import packageName.*;
```

In this way, we write only one *import* statement per package (lines 5–7).

Notice that the *java.awt.event* package is different from the *java.awt* package. You might assume that importing *java.awt.** also imports the classes in the *java.awt.event* package. That is not the case, however; we need to import both packages.

COMMON ERROR TRAP

The *java.awt.event* package is not imported with the *java.awt* package. Include *import* statements for both the *java.awt* and the *java.awt.event* packages.

At lines 51–74, we define our event handler, *ButtonHandler*, as a *private* inner class. Because clicking a button fires an *ActionEvent*, our event handler implements the *ActionListener* interface. Thus, we provide our code to handle the event in the *actionPerformed* method. Line 59 retrieves the data the user typed into the text field and attempts to convert the text to a *double*. Because the *parseDouble* method will *throw* a *NumberFormatException* if the user enters a value that cannot be converted to a *double*, we perform the conversion inside a *try* block.

The *if* statement at line 62 calls the *getSource* method using the *ActionEvent* object reference to test if the component that fired the event is the *square* button. If so, we set the text of the *result* label to the square of the number entered.

If the *square* button did not fire the event, we test if the source of the event was the *cube* button (line 64). If so, we set the text of the *result* label to the cube of the number entered. Actually, the test in line 64 is not necessary, because we registered our event handler as the listener for only two buttons. So if the source isn't the *square* button, then the *cube* button must have fired the event. However, in order to improve readability and maintenance, it is good practice to check the source of the event specifically, in particular if additional buttons or components will be added to the application later.

SOFTWARE ENGINEERING TIP

Verify the source component before processing the event.

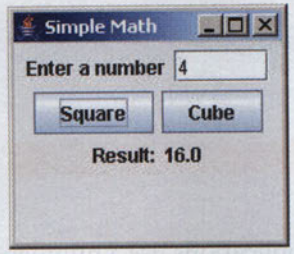


Figure 12.7
Running Example 12.4

In the constructor, at lines 40–41, we declare and instantiate the *ButtonHandler* event handler, *bh*. At lines 43–45, we call the *addActionListener* method to register the handler on our two buttons.

The output of this example is shown in Figure 12.7. When you run this example, try entering a valid number into the text field, and press each button. Then try to enter text that cannot be converted to a number. Also, try deleting either statement that registers the event listener on the buttons (line 44 or 45), and run the application again.

12.7 Radio Buttons and Checkboxes

If you have ever completed a survey on the web, you are probably acquainted with radio buttons and checkboxes.

Radio buttons prompt the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected option. Thus, in a group of radio buttons, a user can select only one option at a time.

Checkboxes are often associated with the instruction "check all that apply"; that is, the user is asked to select 0, 1, or more options. A checkbox is a toggle button in that if the option is not currently selected, clicking on a checkbox selects the option, and if the option is currently selected, clicking on the checkbox deselects the option.

The *JRadioButton* and *JCheckBox* classes, which belong to the *javax.swing* package, encapsulate the concepts of radio buttons and checkboxes, respectively. We will present two similar examples in order to illustrate how to use these classes and how they differ. Both examples allow the user to select the background color for a label component. We display three color options: red, green, and blue. Using radio buttons, only one option can be selected at a time. Thus, by clicking on a radio button, the user will cause the background of the label to be displayed in one of three colors. Using

checkboxes, the user can select any combination of the three color options, so the label color can be set to any of eight possible combinations. Table 12.11 lists all these combinations.

Table 12.12 shows several constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* classes.

TABLE 12.11 Selecting Colors Using Radio Buttons vs. Checkboxes

Using Radio Buttons
(1 selection possible at a time)

Color Selection			Resulting Color
red	green	blue	
◉	○	○	red
○	◉	○	green
○	○	◉	blue

Using Checkboxes
(0 to 3 selections possible at a time)

Color Selections			Resulting Color
red	green	blue	
☐	☐	☐	black
☑	☐	☐	red
☐	☑	☐	green
☐	☐	☑	blue
☑	☑	☐	yellow
☑	☐	☑	purple
☐	☑	☑	blue-green
☑	☑	☑	white

TABLE 12.12 Useful Constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* Classes

Constructors of the *JRadioButton*, *ButtonGroup*, and *JCheckBox* Classes	
Constructors	
Class	**Constructor**
`JRadioButton`	`JRadioButton( String buttonLabel )` constructs a radio button labeled *buttonLabel*. By default, the radio button is initially deselected.
`JRadioButton`	`JRadioButton( String buttonLabel, boolean selected )` constructs a radio button labeled *buttonLabel*. If *selected* is *true*, the button is initially selected; if *selected* is *false*, the button is deselected.
`ButtonGroup`	`ButtonGroup( )` constructs a button group. Adding buttons to this group makes the buttons mutually exclusive.
`JCheckBox`	`JCheckBox( String checkBoxLabel )` constructs a checkbox labeled *checkBoxLabel*. By default, the checkbox is initially deselected.
`JCheckBox`	`JCheckBox( String checkBoxLabel, boolean selected )` constructs a checkbox labeled *checkBoxLabel*. If *selected* is *true*, the checkbox is initially selected; if *selected* is *false*, the checkbox is initially deselected.

Selecting or deselecting radio buttons and checkboxes fires an *ItemEvent*. To receive this event, our event handler needs to implement the *ItemListener* interface, which has only one method, shown in Table 12.13.

Note that selecting a radio button fires both an *ItemEvent* and an *ActionEvent*, so an alternative for radio buttons is to register a handler that implements the *ActionListener* interface. We have chosen to use an *ItemListener* for both radio buttons and checkboxes to demonstrate the similarities and differences between the two components.

TABLE 12.13 *ItemListener* Method API

`public void itemStateChanged( ItemEvent event )` An event handler that implements the *ItemListener* interface writes code in this method to respond to the *ItemEvent* fired by any registered components.

Example 12.5 shows the color selection application using radio buttons.

```
1 /* Select a Color using JRadioButtons
2    Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 public class ChangingColors extends JFrame
10 {
11  private Container contents;
12  private JRadioButton red, green, blue;
13  private ButtonGroup colorGroup;
14  private JLabel label;
15  private Color selectedColor = Color.RED;
16
17  public ChangingColors( )
18  {
19   super( "Selecting a color" );
20   contents = getContentPane( );
21   contents.setLayout( new FlowLayout( ) );
22
23   red = new JRadioButton( "red", true );
24   green = new JRadioButton( "green" );
25   blue = new JRadioButton( "blue" );
26
27   label = new JLabel( "Watch my background" );
28   label.setForeground( Color.GRAY );
29   label.setOpaque( true );
30   label.setBackground( selectedColor );
31
32   contents.add( red );
33   contents.add( green );
34   contents.add( blue );
35   contents.add( label );
36
37   // create button group
38   colorGroup = new ButtonGroup( );
39   colorGroup.add( red );
40   colorGroup.add( green );
41   colorGroup.add( blue );
42
```

```
43   // create RadioButtonHandler event handler
44   // and register it on the radio buttons
45   RadioButtonHandler rbh = new RadioButtonHandler( );
46   red.addItemListener( rbh );
47   green.addItemListener( rbh );
48   blue.addItemListener( rbh );
49
50   setSize( 225, 200 );
51   setVisible( true );
52  }
53
54  private class RadioButtonHandler implements ItemListener
55  {
56   public void itemStateChanged( ItemEvent ie )
57   {
58    if ( ie.getSource( ) == red )
59       selectedColor = Color.RED;
60    else if ( ie.getSource( ) == green )
61       selectedColor = Color.GREEN;
62    else if ( ie.getSource( ) == blue )
63       selectedColor = Color.BLUE;
64
65    label.setBackground( selectedColor );
66   }
67  }
68
69  public static void main( String [ ] args )
70  {
71   ChangingColors cc = new ChangingColors( );
72   cc.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
73  }
74 }
```

EXAMPLE 12.5 Selecting Colors Using *JRadioButtons*

The various instance variables are declared at lines 11–15. The instance variable *selectedColor*, which is initialized to red, will store the choice of the user and will be used to set the background of the *JLabel* component *label*. The three radio buttons and *label* are instantiated at lines 23–27. At line 23, we use a *JRadioButton* constructor with two arguments to instantiate the radio button *red*. The first argument is the title of the radio button; the second argument, *true*, specifies that *red* is selected. Thus, when the window appears, the

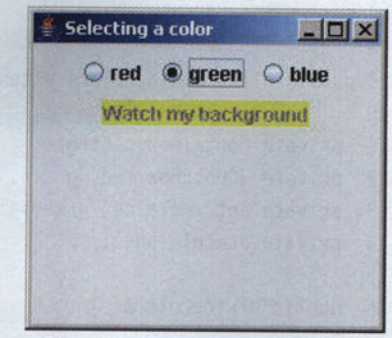


Figure 12.8
Running Example 12.5

red radio button will be selected. In order to reflect the effect of the selected radio button, we set the background color of the label to red at line 30.

At lines 24 and 25, *green* and *blue* are instantiated using the *JRadioButton* constructor with only a *String* argument, specifying the radio button titles. The four components are added to the content pane at lines 32–35.

At lines 37–41, we instantiate a *ButtonGroup* object, then add the *red*, *green*, and *blue* radio buttons to *colorGroup* so that they will be mutually exclusive selections.

At lines 43–48, we instantiate our event handler, *RadioButtonHandler rbh*, and call the *addItemListener* method to register the listener *rbh* on the three radio buttons.

At lines 54–67, the *private* class *RadioButtonHandler implements* the *ItemListener* interface. The only method of *ItemListener*, *itemStateChanged*, is overriden at lines 56–66. Inside *itemStateChanged*, we use an *if/else* statement to determine which radio button was selected in order to set *selectedColor* to the appropriate color.

Figure 12.8 shows a run of the example after the user has clicked on the *green* radio button. Try running the example and clicking on each radio button. Try modifying the code; for example, delete the lines adding the radio buttons to the button group, then compile and run the program again.

COMMON ERROR TRAP

Forgetting to make *JRadioButtons* part of the same *ButtonGroup* makes them independent and not mutually exclusive.

Example 12.6 shows the color selection application using checkboxes.

```
1 /*  Using JCheckBoxes
2     Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
```

```
import java.awt.event.*;

public class MixingColors extends JFrame
{
 private Container contents;
 private JCheckBox red, green, blue;
 private int redValue, greenValue, blueValue;
 private JLabel label;

 public MixingColors( )
 {
  super( "Selecting a color" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );

  red = new JCheckBox( "red" );
  green = new JCheckBox( "green" );
  blue = new JCheckBox( "blue" );

  label = new JLabel( "Watch my background" );
  label.setOpaque( true );
  label.setForeground( Color.GRAY );
  label.setBackground( new Color ( 0, 0, 0 ) );

  contents.add( red );
  contents.add( green );
  contents.add( blue );
  contents.add( label );

  // create CheckBoxHandler event handler
  // and register it on the checkboxes
  CheckBoxHandler cbh = new CheckBoxHandler( );
  red.addItemListener( cbh );
  green.addItemListener( cbh );
  blue.addItemListener( cbh );

  setSize( 225, 200 );
  setVisible( true );
 }

 private class CheckBoxHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
   if ( ie.getSource( ) == red )
```

```
52    {
53     if ( ie.getStateChange( ) == ItemEvent.SELECTED )
54       redValue = 255;
55     else
56       redValue = 0;
57    }
58    else if ( ie.getSource( ) == green )
59    {
60     if ( ie.getStateChange( ) == ItemEvent.SELECTED )
61       greenValue = 255;
62     else
63       greenValue = 0;
64    }
65    else if ( ie.getSource( ) == blue )
66    {
67     if ( ie.getStateChange( ) == ItemEvent.SELECTED )
68       blueValue = 255;
69     else
70       blueValue = 0;
71    }
72
73    label.setBackground(
74         new Color( redValue, greenValue, blueValue ) );
75   }
76  }
77
78  public static void main( String [ ] args )
79  {
80   MixingColors mc = new MixingColors( );
81   mc.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
82  }
83 }
```

EXAMPLE 12.6 Using *JCheckBoxes*

The various instance variables are declared at lines 11–14. The instance variables *redValue*, *greenValue*, and *blueValue* will store the red, green, and blue intensity values, depending on which checkboxes the user selects. These values will be used to set the background color of the *JLabel* component *label*.

The three checkboxes and *label* are instantiated at lines 22–26. We use a *JCheckBox* constructor with a *String* argument that specifies the checkbox titles. The four components are added to the content pane at lines 31–34.

TABLE 12.14 A Useful Method of the *ItemEvent* Class

Return value	Method name and argument list
`int`	`getStateChange( )`
	returns the state of the checkbox. If the checkbox is selected, the value *SELECTED* is returned; if the checkbox is deselected, the value *DESELECTED* is returned, where *SELECTED* and *DESELECTED* are *static int* constants of the *ItemEvent* class.

At lines 36–41, we instantiate the *CheckBoxHandler* event handler and register the *cbh* listener on the three checkboxes by calling the *addItemListener* method for each checkbox.

At lines 47–76, we define the *private* inner class, *CheckBoxHandler*, which *implements* the *ItemListener* interface. The only method of *ItemListener*, *itemStateChanged*, is overriden at lines 49–75. Inside *itemStateChanged*, we first use an *if/else* statement to determine which checkbox fired the event. We then use a nested *if/else* statement to determine if that checkbox has just been selected. The *getStateChange* method in the *ItemEvent* class returns this information. Its API is shown in Table 12.14. The *ItemEvent* class provides the *static int* constants *SELECTED* and *DESELECTED* for convenience in coding this method call. Every click on our checkboxes will fire an event, so calling the *getStateChange* method helps us to distinguish between the event fired when the user selects a checkbox from the event fired when the user deselects a checkbox.

If the source checkbox was selected, we set the corresponding color value to 255 (lines 54, 61, and 68). If the source checkbox was deselected, we set the corresponding color value to 0 (lines 56, 63, and 70). For example, if the *red* checkbox is the source of the event, then we check whether the *red* checkbox was selected. If so, we set *redValue* to 255. Otherwise, the *red* checkbox must have been deselected, so we set *redValue* to 0.

At lines 73–74, we set the background color of *label* to a color we instantiate using the values of *redValue*, *greenValue*, and *blueValue* for the color's respective intensities of red, green, and blue.

Figure 12.9 shows a run of the example after the user has selected the red and green checkboxes.

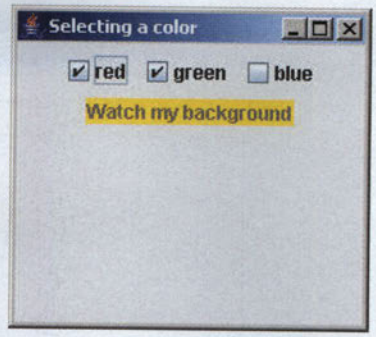


Figure 12.9
Running Example 12.6

12.8 Programming Activity 1: Working with Buttons

In this activity, you will work with two *JButtons* that control one electrical switch. Specifically, you will write the code to perform the following operations:

1. If the user clicks on the "OPEN" button, open the switch.
2. If the user clicks on the "CLOSE" button, close the switch.

The framework for this Programming Activity will animate your code so that you can check its accuracy. Figures 12.10 and 12.11 show the application after the user has clicked on the button labeled "OPEN" and the button labeled "CLOSE", respectively.

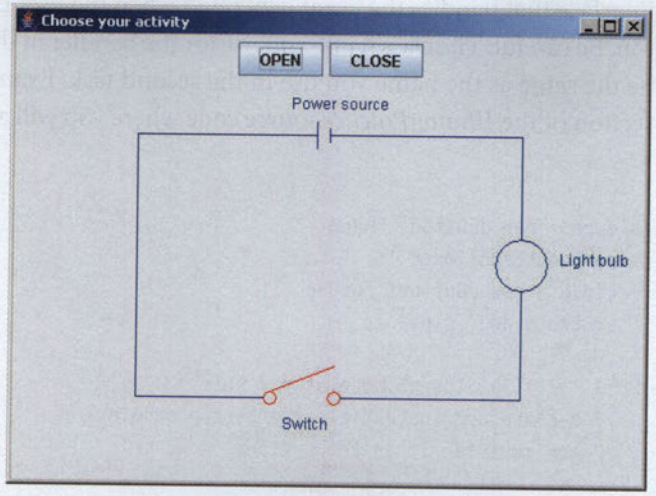


Figure 12.10
User Clicked on "OPEN"

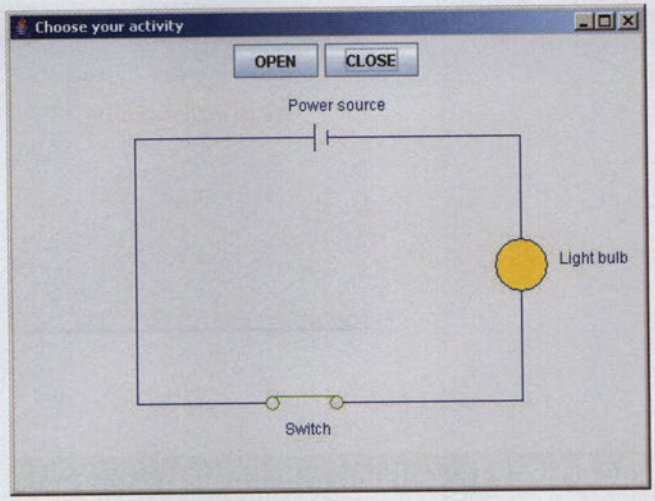


Figure 12.11
User Clicked on "CLOSE"

Instructions

In the Chapter 12 Programming Activity 1 directory on the CD-ROM accompanying this book, you will find the source files needed to complete this activity. Copy all the files to a directory on your computer. Note that all files should be in the same directory.

Open the *JButtonPractice.java* source file. Searching for five asterisks (*****) in the source code will position you to the first code section and then to the second location where you will add your code. In task 1, you will fill in the code to instantiate a button handler and register the handler on the appropriate components. In task 2, you will code the *private* class implementing the listener interface that handles the event generated when the user clicks on either button. Be careful: The class name you use for the handler in the first task needs to be the same as the name you use in the second task. Example 12.7 shows the section of the *JButtonPractice* source code where you will add your code.

```
...
open = new JButton( "OPEN" );
contents.add( open );
close = new JButton( "CLOSE" );
contents.add( close );

// ***** 1. Student code starts here
// declare and instantiate the button handler
// and register it on the buttons
```

```
    // end of task 1

    setSize( 500, 375 );
    setVisible( true );
}

// ***** 2. Student code restarts here
// Code a private class to implement the correct Listener
// and its required method
// To open the switch, call the open method with the statement
//   open( );
// To close the switch, call the close method with the statement
//   close( );
// The last statement of the method should be
//   animate( );


// end of task 2
```

EXAMPLE 12.7 Location of Student Code in *JButtonPractice.java*

The framework will animate your code so that you can watch your code work. For this to happen, be sure that you call the *animate* method.

To test your code, compile and run the *JButtonPractice.java* file, then click on each button.

Troubleshooting

If your method implementation does not animate, check these tips:

- Verify that the last statement in your method inside the *private* class is:

  ```
  animate( );
  ```

- Verify that your listener is registered on the components.
- Verify that you have identified the origin of the event correctly.

? DISCUSSION QUESTIONS

1. Explain why the *getSource* method is useful here.
2. Could you implement the preceding with *JRadioButtons* instead of *JButtons*? What interface would you implement? What would be the method of the interface that you would need to override? What type of event would you handle?

12.9 Lists

In GUIs, often users are asked to select one or more items in a list. Sometimes all items in the list are displayed by default, and sometimes clicking a button displays the items in a scrollable window. Java provides the *JList* class for the former and the *JComboBox* class for the latter.

The *JList* class encapsulates a list in which the user can select one or more items. An item is selected by clicking on it. A range of items can be selected by clicking on the first item in the range, then holding down the *Shift* key while selecting the last item in the range. You can add an item to a selected group of items by holding down the control (*CTRL*) key while clicking on that item.

When the user clicks on one or more items in a list, the *JList* component fires a *ListSelectionEvent*. Our event handler for a *JList* will implement the *ListSelectionListener* interface, which has only one method, *valueChanged*, with the API shown in Table 12.15.

Table 12.16 shows some useful constructors and methods of the *JList* class.

Example 12.8 shows how to use the *JList* class. In this application, we allow the user to choose from a list of countries. We then display a typical food from the country selected.

We construct our *JList* (line 31) by providing an array of *Strings*, *countryList*, which holds the names of the countries to be displayed in the list. When the user selects a country from the list, we display the food image in a *JLabel* component, named *foodImage*, which we instantiate at line 32.

SOFTWARE ENGINEERING TIP

Arrange list items in a logical order so that the user can find the desired item quickly.

Notice that our list of countries is arranged alphabetically to simplify finding a desired country. In this case, we have only five countries; however, with a longer list, the alphabetical ordering would make it easy for the user to find a particular item in the list. Another option is to display the most likely choice first, then display the remaining items in alphabetical order.

TABLE 12.15 *ListSelectionListener* Method API

`public void valueChanged( ListSelectionEvent e )` An event handler that implements the *ListSelectionListener* interface writes code in this method to respond to the *ListSelectionEvent* fired by any registered components.

TABLE 12.16 Useful Methods of the *JList* Class

Constructors and Methods of the *JList* Class	
Constructor	
Class	**Constructor**
`JList`	`JList( Object [ ] arrayName )` constructs a new *JList* component initially filled with the objects in *arrayName*. Often, the objects are *Strings*.
Methods	
Return value	**Method name and argument list**
`void`	`setSelectionMode( int selectionMode )` sets the number of selections that can be made at one time. The following *static int* constants of the *ListSelectionModel* interface can be used to set the selection mode: SINGLE_SELECTION—one selection allowed SINGLE_INTERVAL_SELECTION—multiple contiguous items can be selected MULTIPLE_INTERVAL_SELECTION—multiple contiguous intervals can be selected (This is the default.)
`int`	`getSelectedIndex( )` returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setSelectedIndex( int index )` selects the item at *index*. The index of the first item in the list is 0.

For example, if most users reside in the United States, you could list "USA" first in the list, with the remaining countries alphabetized starting in the second position of the list.

We declare and instantiate the instance variable, *foods*, as an array of *ImageIcons* (lines 17–22). The *foods* array stores images of the food samplings in the same order as the countries in our list. We initialize the *foodImage* label to the first element of *foods* array (line 32). Thus, when we first display the window, the label will show the food sampling from France, which is the first country in the list. In order to match the displayed food sampling with

Figure 12.12
Running Example 12.8

the list selection, we programmatically select the first item of the *countries* list (line 36), using the method *setSelectedIndex* of the *JList* class.

For this application, we will allow the user to select only one country at a time, so we set the selection mode of the list to single selection at line 35 using the *setSelectionMode* method of the *JList* class. The argument passed to the method, *SINGLE_SELECTION*, is a *static int* constant of the *ListSelectionModel* interface.

At lines 50–56, we define our event handler as the *private* inner class, *ListHandler*, which implements the *ListSelectionListener* interface. The only method of *ListSelectionListener*, *valueChanged*, is overriden at lines 52–55. We set up the appropriate food icon to display by calling the *getSelectedIndex* method of the *JList* class. This method returns the index of the item selected by the user. Because we have set up the *foods* array in the same order as the *countries* array, we can use the return value from the *getSelectedIndex* method as the index into the *foods* array to retrieve the corresponding food image for the country selected. In this way, we pass the corresponding array element of *foods* to the *setIcon* method of *JLabel* so that the *foodImage* label will display the appropriate food sampling. Figure 12.12 shows this example running after the user has clicked on "Greece".

```
1 /* Using JList to show a sampling of international foods
2    Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import javax.swing.event.*;
8
9 public class FoodSamplings extends JFrame
10 {
11  private Container contents;
12  private JList countries;
```

```
13 private JLabel foodImage;
14
15 private String [ ] countryList =
16           { "France", "Greece", "Italy", "Japan", "USA" };
17 private ImageIcon [ ] foods =
18           {  new ImageIcon( "cheese.jpg" ),
19              new ImageIcon( "fetaSalad.jpg" ),
20              new ImageIcon( "pizza.jpg" ),
21              new ImageIcon( "sushi.jpg" ),
22              new ImageIcon( "hamburger.jpg" ) };
23
24 public FoodSamplings( )
25 {
26  super( "Food samplings of various countries" );
27  contents = getContentPane( );
28  contents.setLayout( new FlowLayout( ) );
29
30  // instantiate the components
31  countries = new JList( countryList );
32  foodImage = new JLabel( foods[0] );
33
34  // allow single selections only
35  countries.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
36  countries.setSelectedIndex( 0 );
37
38  // add components to the content pane
39  contents.add( countries );
40  contents.add( foodImage );
41
42  // set up event handler
43  ListHandler lslh = new ListHandler( );
44  countries.addListSelectionListener( lslh );
45
46  setSize( 350, 150 );
47  setVisible( true );
48 }
49
50 private class ListHandler implements ListSelectionListener
51 {
52  public void valueChanged( ListSelectionEvent lse )
53  {
54   foodImage.setIcon( foods[countries.getSelectedIndex( )] );
55  }
56 }
57
```

```
58  public static void main( String args[ ] )
59  {
60   FoodSamplings fs = new FoodSamplings( );
61   fs.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
62  }
63 }
```

EXAMPLE 12.8 Using *JList* to Display Food Samplings

12.10 Combo Boxes

A *JComboBox* implements a drop-down list. When the combo box appears, one item is displayed, along with a button showing a down arrow. When the user presses on the button, the combo box "drops" open and displays a list of items, with a scroll bar for viewing more items. The user can select only one item from the list.

When the user selects an item, the list closes and the selected item is the one item displayed. A *JComboBox* fires an *ItemEvent*, so the event handler must implement the *ItemListener* interface, and thus, provide the *itemStateChanged* method.

Table 12.17 shows the APIs for a constructor and some useful methods of the *JComboBox* class. The constructor is similar to the *JList* constructor in that it takes an array of objects. The *JComboBox* class also provides the *getSelectedIndex* method, so that the event handler can determine which item the user has selected, and the *setSelectedIndex* method, which typically is used to initialize the default item displayed when the list appears. The *setMaximumRowCount* method allows us to specify how many items to display when the combo box opens. The user can use the scroll bar to move through all the items in the list.

To illustrate a combo box, we will display five possible destinations for a spring break vacation: Cancun, Colorado, Jamaica, Orlando, and Pinehurst. When the user selects a destination, information about that destination will be displayed in a text area.

The information about our vacation specials is stored in the file *specials.txt*. Each line in the file represents a vacation destination, the sponsoring organization, a brief description, and a price.

The contents of the *specials.txt* file is shown in Figure 12.13.

TABLE 12.17 Useful Constructors and Methods of the *JComboBox* Class

Constructors and Methods of the *JComboBox* Class	
Constructor	
Class	**Constructor**
`JComboBox`	`JComboBox( Object [ ] arrayName )`
	constructs a new *JComboBox* component initially filled with the objects in *arrayName*. Often, the objects are *Strings*.
Methods	
Return value	**Method name and argument list**
`void`	`setSelectedIndex( int index )`
	sets the item at *index* as selected. The index of the first item in the list is 0.
`int`	`getSelectedIndex( )`
	returns the index of the selected item. The index of the first item in the list is 0.
`void`	`setMaximumRowCount( int size )`
	sets the number of rows that will be visible at one time to *size*.

```
Cancun,Club Med,all inclusive,1230
Colorado,Club Med,all inclusive,780
Jamaica,Extreme Vacations,all inclusive,1150
Orlando,Disney Vacations,unlimited pass to DisneyWorld,800
Pinehurst,Golf Concepts,unlimited golf,900
```

Figure 12.13
Contents of the *specials.txt* File

We will first create a *Vacation* class to encapsulate the information about each vacation destination. Then we will create a *VacationList* class that reads vacation information from a text file and creates an *ArrayList* of *Vacation* objects. Finally, our application will retrieve data from a *VacationList* object to create the items for our *JComboBox* dynamically.

For simplicity, in our *Vacation* class (Example 12.9), we have coded only an overloaded constructor (lines 17–32), the accessor method for the *location* instance variable (lines 34–40), and the *toString* method (lines 42–51).

```
1 /* Vacation class
2    Anderson, Franceschi
3 */
4
5 import java.text.DecimalFormat;
6
7 public class Vacation
8 {
9  public final DecimalFormat MONEY
10                    = new DecimalFormat( "$#,##0.00" );
11
12  private String location;
13  private String organization;
14  private String description;
15  private double price;
16
17  /** Constructor
18  *  @param  startLocation       location
19  *  @param  startOrganization   organization
20  *  @param  startDescription    description
21  *  @param  startPrice          price
22  */
23  public Vacation( String startLocation,
24                   String startOrganization,
25                   String startDescription,
26                   double startPrice )
27  {
28   location = startLocation;
29   organization = startOrganization;
30   description = startDescription;
31   price = startPrice;
32  }
33
34  /** getLocation
35  *  @return location
36  */
37  public String getLocation( )
38  {
```

```
39    return location;
40  }
41
42  /** toString
43  *  @return location, organization, description, and price
44  */
45  public String toString( )
46  {
47    return "Location: " + location + "\n"
48           + "Organization: " + organization + "\n"
49           + "Description: " + description + "\n"
50           + "Price: " + MONEY.format( price );
51  }
52 }
```

EXAMPLE 12.9 The *Vacation* Class

Our next step is to create the *VacationList* class, shown in Example 12.10, which we will use to read vacation information from a file and build a list of *Vacation* objects. We assume that we do not know how many records are in the file; thus, we will choose an *ArrayList* rather than an array to store the *Vacation* objects.

Again, for simplicity, we have coded just an overloaded constructor (lines 12–56), the *getLocationList* method (lines 58–67), and the *getDescription* method (lines 69–76). The only instance variable, *vacationList*, is an *ArrayList* of *Vacation* objects. The overloaded constructor takes one argument, the name of the file that contains the data.

At lines 23–45, the *while* loop reads each line of the file, creates a *Vacation* object, and adds the object to the *ArrayList*. The *getLocationList* method returns a *String* array of the *location* for each *Vacation* object. We will call this method in our GUI application to create the items for our *JComboBox*. The *getDescription* method returns the *String* representation of the *Vacation* object stored at a given index of *vacationList*. Our GUI application will call this method to display information about the vacation destination that the user has selected from the *JComboBox*.

REFERENCE POINT

The *ArrayList* class is explained in Chapter 9. *I/O* classes and the *Scanner* class are discussed in Chapter 11.

```
1 /* VacationList class
2    Anderson, Franceschi
3 */
4
```

```
5 import java.util.*;
6 import java.io.*;
7
8 public class VacationList
9 {
10  private ArrayList<Vacation> vacationList;
11
12  /** Constructor
13  *  @param  fileName  the name of the file containing the data
14  */
15  public VacationList( String fileName )
16  {
17   vacationList = new ArrayList<Vacation>( );
18   try
19   {
20    File file = new File( fileName );
21    Scanner br = new Scanner( file );
22
23    while ( br.hasNext( ) )
24    {
25     String record = br.nextLine( );
26     // extract the fields from the records
27     Scanner scanLine = new Scanner( record );
28     scanLine.useDelimiter( "," );
29     String loc = scanLine.next( );
30     String org =  scanLine.next( );
31     String desc =  scanLine.next( );
32
33     try
34     {
35      double pr = scanLine.nextDouble( );
36
37      Vacation vacationTemp = new Vacation( loc, org, desc, pr );
38      vacationList.add( vacationTemp );
39     }
40     catch ( InputMismatchException ime )
41     {
42      System.out.println( "Error in vacation record: "
43              + record + "; record ignored" );
44     }
45    }
46    br.close( )
47   }
```

```
48   catch ( FileNotFoundException fnfe )
49   {
50    System.out.println( "Unable to find " + fileName );
51   }
52   catch ( IOException ioe )
53   {
54    ioe.printStackTrace( );
55   }
56  }
57
58  /** getLocationList
59  *  @return array of locations
60  */
61  public String [ ] getLocationList( )
62  {
63   String [ ] temp = new String[vacationList.size( )];
64   for ( int i = 0; i < vacationList.size( ); i++ )
65          temp[i] = ( vacationList.get( i ) ).getLocation( );
66   return temp;
67  }
68
69  /** getDescription
70  *  @param  index     index of vacation in vacationList
71  *  @return description of vacation at index
72  */
73  public String getDescription( int index )
74  {
75   return ( ( vacationList.get( index ) ).toString( ) );
76  }
77 }
```

EXAMPLE 12.10 The *VacationList* Class

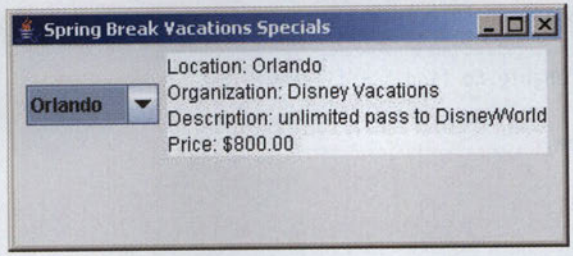
We are now ready to code a GUI application, shown in Example 12.11.

The instance variable, *vacations*, a *VacationList* object reference, will store an *ArrayList* of vacation information. We instantiate *vacations* (line 20) by passing the file name that the *main* method passed to the constructor (line 53). For simplicity in execution of the application, we have hard coded the file name. However, we could have accepted the file name as a command line argument to the application or we could have prompted the user for the file name.

REFERENCE POINT

Retrieving command line arguments is discussed in Chapter 8.

Figure 12.14
Running Example 12.11



We then instantiate our *JComboBox* (line 26) by passing to the constructor the array of locations returned by the *getLocationList* method of the *VacationList* class.

At line 27, we call the *setMaximumRowCount* method of the *JComboBox* class with an argument of *4* to set the maximum number of items that the user will see when the combo box opens.

We use a *JTextArea* component, named *summary*, to display the vacation descriptions, because the descriptions are of variable length. At line 28, we instantiate the *JTextArea*, with the initial value being the description of the first *Vacation* object in *vacations*. We retrieve the first description by calling the *getDescription* method of the *VacationList* class with an index of 0. By default, a *JComboBox* displays the first item in the list, so in this way, we will match the description with the first item displayed.

At lines 42–49, we define the *ItemListenerHandler* event handler. At line 46, we call the *getSelectedIndex* method of the *JComboBox* class to determine which item the user has selected. Using that returned index, we call the *getDescription* method of the *VacationList* class to set the text of the *JTextArea* to the description matching the selected item.

Figure 12.14 shows a run of the application, after the user has clicked on "*Orlando*" inside the combo box. Try running the example, then modify the number of visible rows by changing the argument of the *setMaximumRowCount* method, and run the example again.

```
1 /* Using a JComboBox to display dynamic data
2    Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
```

```
8
9  public class VacationSpecials extends JFrame
10 {
11  private VacationList vacations;
12
13  private Container contents;
14  private JComboBox places;
15  private JTextArea summary;
16
17  public VacationSpecials( String fileName )
18  {
19   super( "Spring Break Vacations Specials" );
20   vacations = new VacationList( fileName );
21
22   contents = getContentPane( );
23   contents.setLayout( new FlowLayout( ) );
24
25   // instantiate components
26   places = new JComboBox( vacations.getLocationList( ) );
27   places.setMaximumRowCount( 4 );
28   summary = new JTextArea( vacations.getDescription( 0 ) );
29
30   // add components to content pane
31   contents.add( places );
32   contents.add( summary );
33
34   // set up event handler
35   ItemListenerHandler ilh = new ItemListenerHandler( );
36   places.addItemListener( ilh );
37
38   setSize( 350, 150 );
39   setVisible( true );
40  }
41
42  private class ItemListenerHandler implements ItemListener
43  {
44   public void itemStateChanged( ItemEvent ie )
45   {
46    int index = places.getSelectedIndex( );
47    summary.setText( vacations.getDescription( index ) );
48   }
49  }
50
51  public static void main( String [ ] args )
52  {
```

```
53   VacationSpecials vs = new VacationSpecials( "specials.txt" );
54   vs.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
55  }
56 }
```

EXAMPLE 12.11 The *VacationSpecials* class

Skill Practice
with these end-of-chapter questions

12.18.1 Multiple Choice Exercises

Questions 5, 6, 7, 8

12.18.2 Reading and Understanding Code

Questions 16, 19, 20, 21, 22, 23, 24, 25, 26

12.18.3 Fill In the Code

Questions 28, 29, 31, 32

12.18.4 Identifying Errors in Code

Questions 43, 44, 45

12.18.5 Debugging Area

Questions 49, 50

12.18.6 Write a Short Program

Questions 51, 52, 53, 54, 55, 56, 57

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of working with GUI components. Click on the link for Chapter 12 to start the movie.

12.11 Adapter Classes

We have learned that events are associated with event listener interfaces, and that we implement at least one interface in order to process an event. So far, we have used the *ActionListener*, *ItemListener*, and *ListSelection-*

Listener interfaces, each of which has only one method. For mouse events, there are two listeners: the *MouseListener* and the *MouseMotionListener*. The *MouseListener* interface specifies five methods to implement and the *MouseMotionListener* interface specifies two methods to implement. If we want to use *MouseListener* but need only one of its five methods to process a *MouseEvent*, we still have to implement the other four methods as "do-nothing" methods with empty method bodies. For convenience, Java provides **adapter classes**, each of which *implements* an interface and provides an empty body for each of the interface's methods. For mouse events, the adapter classes are *MouseAdapter* and the *MouseMotionAdapter*. Thus, instead of implementing an interface, we can extend the appropriate adapter class and override only the method or methods we need. For example, if we want to process a *MouseEvent* using only one method of the five in the *MouseListener* interface, we can simply extend the *MouseAdapter* class and override that one method. We will do that in our next example.

SOFTWARE ENGINEERING TIP

When you need to implement only a few methods of a listener interface having more than one method, consider extending the corresponding adapter class instead.

12.12 Mouse Movements

A truly interactive application allows the user to point and click using the mouse. Any mouse activity (clicking, moving, or dragging) by the user generates a *MouseEvent*. When any mouse activity occurs, we will be interested in determining where on the window the user clicked, moved, or dragged the mouse. To determine the (x, y) coordinate of the mouse event, we can call two methods of the *MouseEvent* class, *getX* and *getY*, which are described in Table 12.18.

Depending on the type of mouse activity, our application will implement either a *MouseListener* interface or a *MouseMotionListener* interface. As

TABLE 12.18 Useful Methods of the *MouseEvent* Class

Return value	Method name and argument list
`int`	`getX( )`
	returns the *x* value of the (*x*, *y*) coordinate of the mouse activity
`int`	`getY( )`
	returns the *y* value of the (*x*, *y*) coordinate of the mouse activity

TABLE 12.19 Methods of the *MouseListener* Interface

MouseListener Interface Method APIs
`public void mousePressed( MouseEvent e )` called when the mouse button is pressed
`public void mouseReleased( MouseEvent e )` called when the mouse button is released after being pressed
`public void mouseClicked( MouseEvent e )` called when the mouse button is pressed and released
`public void mouseEntered( MouseEvent e )` called when the mouse enters the registered component
`public void mouseExited( MouseEvent e )` called when the mouse exits the registered component

mentioned, both of the listener interfaces have been implemented as adapter classes that we can extend. The *MouseListener* interface includes the five *abstract* methods described in Table 12.19.

To illustrate how to use the *MouseAdapter* class, we will build a simple submarine hunt game. A submarine is hidden somewhere in the window, and the user will try to sink the submarine by clicking the mouse at various locations in the window, simulating the dropping of a depth charge. Each time the user clicks the mouse, we will indicate how close that click is to the submarine. If the user clicks too far from the submarine, we will display "In the water" in the title bar and draw a blue circle where the mouse was clicked. If the user clicks close to the submarine, we will display "Close ..." in the title bar. Finally, if the submarine is hit, we will change the title bar to "Sunk!", display the submarine, and remove the listener so that the game ends.

Before designing the GUI class, we first code a class to encapsulate the functionality of the submarine hunt game; this class (Example 12.12) enables us to create a game of a given size, enable play, and enforce the rules of the game.

The instance variable, *status* (line 17), a *String*, stores the state of the last shot fired to the submarine. We also define a *boolean* flag variable, *hit*, which is initially *false* (lines 18 and 33). Thus, we will change *hit* to *true* when the user successfully hits the submarine.

In the constructor, we randomly generate the (x, y) coordinate for the center of the submarine and store the generated values in the *xCtr* and *yCtr* instance variables (lines 29–32).

The *play* method (lines 60–80) processes a play from the user, setting the value of *status* to either "Sunk!", "Close...", or "In the water" based on the x and y coordinate of the shot fired by the user. "Sunk!" means the submarine has been hit, "Close..." means that our shot was within two lengths of the center of the submarine, and "In the water" means that our shot was more than two lengths away. If the submarine was hit, we set the value of *hit* to *true* (line 71).

In the *draw* method (lines 82–110), we test the value of *status*. If *status* is equal to "Sunk!", the submarine has been hit so we draw a red depth charge and the sunken submarine (lines 91–102), revealing its location. If *status* is equal to "In the water", we draw a filled blue circle (lines 106–107) at the location of the shot fired. If *status* is equal to "Close...", we do not draw anything so that we do not give too big of a hint to the player.

```
 1 /* SubHunt class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.awt.Graphics;
 6 import java.awt.Color;
 7 import java.util.Random;
 8
 9 public class SubHunt
10 {
11  public static int DEFAULT_GAME_SIZE = 300;
12  public static int SIDE = 35; // size of submarine
13
14  private int gameSize;
15  private int xCtr;    // x coordinate of center of submarine
16  private int yCtr;    // y coordinate of center of submarine
17  private String status = "";
18  private boolean hit;
```

```
19
20  /** Constructor
21  *  @param  newGameSize          gameSize
22  */
23  public SubHunt( int newGameSize )
24  {
25   if ( newGameSize > SIDE )
26    gameSize = newGameSize;
27   else
28    gameSize = DEFAULT_GAME_SIZE;
29   // generate submarine center
30   Random random = new Random( );
31   xCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
32   yCtr = SIDE / 2 + random.nextInt( gameSize - SIDE );
33   hit = false;
34  }
35
36  /** getStatus
37  *  @return status
38  */
39  public String getStatus( )
40  {
41   return status;
42  }
43
44  /** getGameSize
45  *  @return gameSize
46  */
47  public int getGameSize( )
48  {
49   return gameSize;
50  }
51
52  /** isHit
53  *  @return hit
54  */
55  public boolean isHit( )
56  {
57   return hit;
58  }
59
60  /** play
61  *  @param  x the x coordinate of the play
62  *  @param  y the y coordinate of the play
```

```
63  */
64  public void play( int x, int y )
65  {
66    // is click within the submarine?
67    if ( Math.abs( x - xCtr ) < SIDE / 2
68        && Math.abs( y - yCtr ) < SIDE / 2 )
69    {
70      status = "Sunk!";
71      hit = true;
72    }
73    // is click close?
74    else if ( Math.abs( x - xCtr ) < 2 * SIDE
75              && Math.abs( y - yCtr ) < 2 * SIDE )
76      status = "Close ...";
77    // click is too far from submarine
78    else
79      status = "In the water";
80  }
81
82  /** draw
83  *  @param  g a Graphics object
84  *  @param  x the x coordinate of the play
85  *  @param  y the y coordinate of the play
86  */
87  public void draw( Graphics g, int x, int y )
88  {
89   if ( status.equals( "Sunk!" ) )
90   {
91    // draw sunken submarine
92    g.setColor( Color.BLACK );
93    g.fillRoundRect( xCtr - SIDE/2, yCtr - SIDE/2, SIDE/2, SIDE,
94                     15, 15 );
95    g.fillRoundRect( xCtr - SIDE/4, yCtr - SIDE/3, SIDE/2, SIDE/2,
96                     7 , 7 );
97    g.drawLine( xCtr + SIDE/4, yCtr - SIDE/12,
98                     xCtr + SIDE/2, yCtr - SIDE/12 );
99
100    // draw red depth charge
101    g.setColor( Color.RED );
102    g.fillOval( x - SIDE/4, y - SIDE/4, SIDE/2, SIDE/2 );
103   }
104   else if ( status.equals( "In the water" ) ) // draw blue circle
105   {
106    g.setColor( Color.BLUE );
107    g.fillOval( x - SIDE/2, y - SIDE/2, SIDE, SIDE );
```

```
108    }
109    // else Close ... , do not draw
110  }
111 }
```

EXAMPLE 12.12 The *SubHunt* Class

Example 12.13 shows the GUI client class that enables a user to play the submarine hunting game.

In this game, the only mouse action we care about is a click; therefore, we are interested in only one method of the *MouseListener* interface: *mouseClicked*. To simplify our code, we can extend the *MouseAdapter* class, which provides implementations for the five *MouseListener* methods, so our event handler needs to override only the *mouseClicked* method.

At lines 11–16, we declare our instance variables. The *ints x* and *y* represent the *x* and *y* coordinate where the player clicked the mouse. The *SubHunt* instance variable *sub* represents the submarine hunting game that we will display inside the window. When the user plays, we will use the *sub* reference to call the various methods of the *SubHunt* class to enable play and enforce the rules of the game. We will use *gameStarted* in the *paint* method. Before the game starts *gameStarted* is *false*, and becomes *true* (line 48) when the window opens.

In this example, we have set up the listener reference, *mh*, as an instance variable (line 16) rather than defining the reference as a local variable to the constructor, as we have done in previous examples. We define *mh* as an instance variable because when the submarine is hit, the event handler needs to access that listener to turn it off. The event handler is instantiated at line 24.

In this application, we want the listener to handle mouse clicks anywhere in the window, so we register the *mouseListener* on our window (*JFrame*) component, which is the *SubHuntClient* object. Thus, we do not use an object reference to call the *addMouseListener* method, and line 25 is equivalent to:

```
this.addMouseListener( mh );
```

The constructor also instantiates *sub*.

At lines 31–41, we define our mouse event handler, *MouseHandler*, as extending *MouseAdapter*. Inside the method *mouseClicked*, overridden at lines 33–40, we call the *MouseEvent* methods *getX* and *getY* to get the (*x*, *y*)

coordinate where the mouse was clicked. Then we call the *play* method of *SubHunt* with *sub* (line 37) to process the play. Next, we update the title of the window by getting the updated status and display that status in the window title bar by calling the *setTitle* method (line 38). Finally, we call the *repaint* method (line 39). The *repaint* method is inherited from the *Component* class and has the API shown in Table 12.20. A call to *repaint* forces a call to the *paint* method, which our application cannot call directly. The *paint* method (lines 43–53) updates the drawing in the window. Because we want to preserve any previously drawn blue circles, rather than clear the window and redraw every time, we set *gameStarted* to *true* the first time *paint* is called (lines 45–49). In this way, we call *super.paint(g)* only when the window opens, before the first shot is fired. A subsequent call to *super.paint()* would erase the contents of the window, and we would lose all the previously drawn blue circles.

Then we draw the result of the last shot fired (line 50) by calling the *draw* method from *SubHunt* with *sub*, passing the *Graphics* object *g* and the mouse position. Finally, if the submarine has been hit, we remove *mh* as a listener to this object (line 52) by calling the *removeMouseListener* method, inherited from the *Component* class.

Figure 12.15 shows a run of this game. At this point, the user has sunk the submarine.

```
1 /* Using MouseListener
2    Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 public class SubHuntClient extends JFrame
10 {
11  private int x;      // current x mouse position
12  private int y;      // current y mouse position
13  private SubHunt sub;   // submarine
14  private boolean gameStarted = false;
15
16  private MouseHandler mh; // mouse event handler
17
```

```
18  public SubHuntClient( )
19  {
20   super( "Click in the window to sink the sub" );
21
22   sub = new SubHunt( SubHunt.DEFAULT_GAME_SIZE );
23   // instantiate event handler and register listener on window
24   mh = new MouseHandler( );
25   addMouseListener( mh );
26
27   setSize( sub.getGameSize( ), sub.getGameSize( ) );
28   setVisible( true );
29  }
30
31  private class MouseHandler extends MouseAdapter
32  {
33   public void mouseClicked( MouseEvent me )
34   {
35    x = me.getX( );
36    y = me.getY( );
37    sub.play( x, y );
38    setTitle( sub.getStatus( ) );
39    repaint( );
40   }
41  }
42
43  public void paint( Graphics g )
44  {
45   if ( !gameStarted )
46   {
47    super.paint( g );
48    gameStarted = true;
49   }
50   sub.draw( g, x, y );
51   if ( sub.isHit( ) )
52    removeMouseListener( mh );
53  }
54
55  public static void main( String [ ] args )
56  {
57   SubHuntClient subH = new SubHuntClient( );
58   subH.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
59  }
60 }
```

EXAMPLE 12.13 A Simple Game Using a *MouseListener*

The second interface related to mouse events is *MouseMotionListener*, which has two methods, *mouseMoved* and *mouseDragged*, shown in Table 12.21. Dragging the mouse is defined as the user moving the mouse with the mouse button pressed, but not released.

TABLE 12.20 The *repaint* Method API in the *Component* Class

Return value	Method name and argument list
`void`	`repaint( )`
	automatically forces a call to the *paint* method

Figure 12.15
A Run of Example 12.13

TABLE 12.21 *MouseMotionListener* Interface Method APIs

MouseMotionListener Interface Method APIs
`public void mouseMoved( MouseEvent e )`
called when the mouse is moved onto a component
`public void mouseDragged( MouseEvent e )`
called when the mouse button is pressed on a component and the mouse is dragged

In order to illustrate how to use the *MouseMotionListener* interface with a *MouseEvent* class, we will build a simple treasure hunt game that is similar to the submarine hunt game. A treasure is hidden somewhere in the window, and the user will try to find it, not by clicking the mouse button, but by moving the mouse inside the window. Depending on how close the mouse is to the treasure, we will display a message at the mouse location so that the user can find the treasure and win the game.

Again, we first code a class to encapsulate the functionality of the treasure hunt game (Example 12.14).

The *status* (line 17) instance variable, a *String*, stores the state of the player's position compared to the location of the treasure. We also define a *boolean* flag variable, *gameOver*, which is initially *false* (line 18); when the user finds the treasure, we will change it to *true*.

In the constructor, we randomly generate the (x, y) coordinate for the center of the treasure and store the generated values in the *xCtr* and *yCtr* instance variables (lines 24–26).

The *play* method (lines 37–64) sets the value of *status*. If the player is far from the treasure, we set the value of *status* to "Cold." As the user moves closer and closer to the treasure, the value becomes "Lukewarm," then "Warm," and then "Hot," and finally "Found" when the player finds the treasure. If the treasure is found, we set the value of *gameOver* to *true* (line 47).

In the *draw* method (lines 66–84), we test if the value of *status* is "Found" (line 75). If it is, the treasure has been found and we reveal its location by drawing it (lines 77–80). If the treasure has not been found, we draw the *String status* at the current location (line 83), determined by the parameters *x* and *y* sent to the *draw* method.

```
1 /* TreasureHunt class
2    Anderson, Franceschi
3 */
4
5 import java.awt.*;
6 import java.util.Random;
7
8 public class TreasureHunt
9 {
10  public static int GAME_SIZE = 300; // side of window
11  public static int SIDE = 40; // side of treasure
```

```
12
13  private int xCtr;  // x coordinate of center of treasure
14  private int yCtr;  // y coordinate of center of treasure
15  private int x;     // current x mouse position
16  private int y;     // current y mouse position
17  private String status = ""; // message
18  private boolean gameOver = false;
19
20  /** Constructor
21  */
22  public TreasureHunt( )
23  {
24   Random random = new Random( );
25   xCtr = SIDE / 2 + random.nextInt( GAME_SIZE - SIDE );
26   yCtr = SIDE / 2 + random.nextInt( GAME_SIZE - SIDE );
27  }
28
29  /** isGameOver
30  *  @return gameOver
31  */
32  public boolean isGameOver( )
33  {
34   return gameOver;
35  }
36
37  /** play
38  *  @param  x the x coordinate of the play
39  *  @param  y the y coordinate of the play
40  */
41  public void play( int x, int y )
42  {
43   // is mouse within treasure?
44   if ( Math.abs( x - xCtr ) < SIDE / 2
45        && Math.abs( y - yCtr ) < SIDE / 2 )
46   {
47       gameOver = true;
48       status = "Found";
49   }
50   // is mouse within half-length of the treasure?
51   else if ( Math.abs( x - xCtr ) < ( 1.5 * SIDE )
52             && Math.abs( y - yCtr ) < ( 1.5 * SIDE ) )
53        status = "Hot";
54   // is mouse within 1 length of the treasure?
55   else if ( Math.abs( x - xCtr ) < ( 2 * SIDE )
56             && Math.abs( y - yCtr ) < ( 2 * SIDE ) )
57        status = "Warm";
```

```
58   // is mouse within 2 lengths of the treasure?
59   else if ( Math.abs( x - xCtr ) < ( 3 * SIDE )
60              && Math.abs( y - yCtr ) < ( 3 * SIDE ) )
61        status = "Lukewarm";
62   else // mouse is not near treasure
63        status = "Cold";
64  }
65
66  /** draw
67  *  @param  g a Graphics object
68  *  @param  x the x coordinate of the play
69  *  @param  y the y coordinate of the play
70  */
71  public void draw( Graphics g, int x, int y )
72  {
73   g.setColor( Color.BLUE );
74
75   if ( status.equals( "Found" ) ) // if found, draw treasure
76   {
77    g.setColor( Color.RED );
78    g.fillRect( xCtr - SIDE / 2, yCtr - SIDE / 2, SIDE, SIDE );
79    g.setColor( Color.GREEN );
80    g.drawString( "$$$", xCtr - SIDE / 4, yCtr );
81   }
82   else
83    g.drawString( status, x, y );   // display current status
84  }
85 }
```

EXAMPLE 12.14 The *TreasureHunt* Class

Example 12.15 shows the GUI client class, which enables a user to play the treasure hunt game.

In this application, instead of coding the event handler as a *private* inner class, we define our application class as implementing the *MouseMotionListener* interface. As a result, our application is a listener, and we register the listener on itself.

Thus, in our class definition, we include the clause *implements MouseMotionListener* (lines 9–10).

Again, the instance variables *x* and *y* (lines 12–13) will store the current location of the mouse. The *TreasureHunt* instance variable *hunt* represents the treasure hunting game that we will display inside the window.

```
1 /* A Treasure Hunt using MouseMotionListener
2     Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 public class TreasureHuntClient extends JFrame
10                          implements MouseMotionListener
11 {
12  private int x;      // current x mouse position
13  private int y;      // current y mouse position
14  private TreasureHunt hunt;
15
16  public TreasureHuntClient( )
17  {
18   super( "Move the mouse to find the treasure" );
19
20   hunt = new TreasureHunt( );
21   // application registers on itself
22   // since it is a MouseMotionListener itself
23   addMouseMotionListener( this );
24
25   setSize( hunt.GAME_SIZE, hunt.GAME_SIZE );
26   setVisible( true );
27  }
28
29  public void mouseDragged( MouseEvent me )
30  { } // we do not want to process mouse drag events
31
32  public void mouseMoved( MouseEvent me )
33  {
34   // get location of mouse
35   x = me.getX( );
36   y = me.getY( );
37   hunt.play( x, y );
38   repaint( );
39  }
40
41  public void paint( Graphics g )
42  {
43   super.paint( g );
```

```
44    hunt.draw( g, x, y );
45    if ( hunt.isGameOver( ) )
46     removeMouseMotionListener( this );
47   }
48
49   public static void main( String [ ] args )
50   {
51    TreasureHuntClient th_gui = new TreasureHuntClient( );
52    th_gui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
53   }
54 }
```

EXAMPLE 12.15 A Simple Game Using *MouseMotionListener*

Figure 12.16
The User Is Getting Close to the Treasure

Figure 12.17
The User Has Found the Treasure

At lines 21–23, we register this *TreasureHunt* object on itself as a *MouseMotionListener*. In this game, the only event we want to handle is the user moving the mouse. Because *TreasureHuntClient* implements the *MouseMotionListener* interface, however, we must implement both the *mouseMoved* and the *mouseDragged* methods. We are interested in the *mouseMoved* method only; thus, we implement *mouseDragged* with an empty body (lines 29–30). Note that we could not use the *MouseMotionAdapter* class because our application (which is the listener) already extends the *JFrame* class. Remember that a class can extend only one class, but can implement multiple interfaces.

The *mouseMoved* method is implemented at lines 32–39. At lines 34–36, we assign the (x, y) coordinate of the mouse position to the instance variables x and y, using the *getX* and *getY* methods of the *MouseEvent* class. Then we call the *play* method of *TreasureHunt* with *hunt* (line 37) to process the play, and call the *repaint* method (line 38) to update the window.

We begin the *paint* method (lines 41–47) with a call to the *paint* method of the *JFrame* superclass to clear the contents of the window. Then we draw the result of the current move (line 44) by calling the *draw* method from *TreasureHunt* with *hunt*, passing the *Graphics* object *g* and the mouse location. Finally, if the game is over (i.e., the treasure has been found), we remove this object, the current *JFrame*, as a listener (line 46) by calling the *removeMouseMotionListener* method.

Figures 12.16 and 12.17 show the program running. In Figure 12.16, the user is getting close to the treasure, and in Figure 12.17, the user has found the treasure.

12.13 Layout Managers: *GridLayout*

We have been using the *FlowLayout* layout manager because it is the easiest to use. The *FlowLayout* layout manager adds new components from left to right, creating as many rows as needed. A problem, however, is that if the user resizes the window, the components are rearranged to fit the new window size. Typically, GUIs can be quite complex and use many components,

all of which need to be organized so that the user interface is effective and easy to use even if the user resizes the window. Two useful layout managers that give us more control over the organization of components within a window are the *GridLayout* and the *BorderLayout*.

Regardless of the layout manager used, the constructor of our application will still perform the following setup operations:

- declare and instantiate the layout manager
- use the *setLayout* method of the *Container* class to set the layout manager of the content pane of the *JFrame*
- instantiate components
- add components to the content pane using one of the *add* methods of the *Container* class

The *GridLayout* organizes the container as a grid. We can visualize the layout as a table made up of equally sized cells in rows and columns. Each cell can contain one component. The first component added to the container is placed in the first column of the first row; the second component is placed in the second column of the first row, and so on. When all the cells in a row are filled, the next component added is placed in the first cell of the next row.

Table 12.22 shows two constructors of the *GridLayout* class.

Example 12.16 shows how to use a *GridLayout* to display a chessboard. A two-dimensional array of *JButtons*, named *squares* (line 13), will make up the chessboard. A two-dimensional array of *Strings*, named *names*, will hold the position of each square (lines 14–22). The position of a square is composed of a letter representing the row (a–h) and a number representing

TABLE 12.22 *GridLayout* Constructors

Constructors	
Class	**Constructor**
`GridLayout`	`GridLayout( int numberOfRows, int numberOfColumns )` creates a grid layout with the number of rows and columns specified by the arguments.
`GridLayout`	`GridLayout( int numberOfRows, int numberOfColumns, int hGap, int vGap )` creates a grid layout with the specified number of rows and columns and with a horizontal gap of *hGap* pixels between columns and a vertical gap of *vGap* pixels between rows. Horizontal gaps are also placed at the left and right edges, and vertical gaps are placed at the top and bottom edges.

the column (1–8). When the user clicks on a button of the chessboard, we will set the text of that button to display its position on the board, retrieving the name from the array *names*.

```
/* Using GridLayout to organize our window
   Anderson, Franceschi
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ChessBoard extends JFrame
{
 public static final int SIDE = 8;
 private Container contents;
 private JButton [ ][ ] squares;
 private String [ ][ ] names =
  { { "a1","a2","a3","a4","a5","a6", "a7", "a8" },
    { "b1","b2","b3","b4","b5","b6", "b7", "b8" },
    { "c1","c2","c3","c4","c5","c6", "c7", "c8" },
    { "d1","d2","d3","d4","d5","d6", "d7", "d8" },
    { "e1","e2","e3","e4","e5","e6", "e7", "e8" },
    { "f1","f2","f3","f4","f5","f6", "f7", "f8" },
    { "g1","g2","g3","g4","g5","g6", "g7", "g8" },
    { "h1","h2","h3","h4","h5","h6", "h7", "h8" } };

 public ChessBoard( )
 {
  super( "Click a square to reveal its position" );
  contents = getContentPane( );

  // set layout to an 8-by-8 Grid
  contents.setLayout( new GridLayout( SIDE, SIDE ) );

  squares = new JButton[SIDE][SIDE];

  ButtonHandler bh = new ButtonHandler( );

  for ( int i = 0; i < names.length; i++ )
  {
   for ( int j = 0; j < SIDE; j++ )
   {
    // instantiate JButton array
    squares[i][j] = new JButton( );

    // make every other square red
    if ( ( i + j ) % 2 == 0 )
      squares[i][j].setBackground( Color.RED );
```

```
46
47      // add the JButton
48      contents.add( squares[i][j] );
49
50      // register listener on button
51      squares[i][j].addActionListener( bh );
52    }
53  }
54
55  setSize( 400, 400 );
56  setVisible( true );
57 }
58
59 private class ButtonHandler implements ActionListener
60 {
61  public void actionPerformed( ActionEvent ae )
62  {
63    for ( int i = 0; i < SIDE; i++ )
64    {
65      for ( int j = 0; j < SIDE; j++ )
66      {
67        if ( ae.getSource( ) == squares[i][j] )
68        {
69          squares[i][j].setText( names[i][j] );
70          return;
71        }
72      }
73    }
74  }
75 }
76
77 public static void main( String [ ] args )
78 {
79   ChessBoard myGame = new ChessBoard( );
80   myGame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
81 }
82 }
```

EXAMPLE 12.16 Using *GridLayout* to Display a Chessboard

At line 30, we instantiate a *GridLayout* anonymous object using the constructor with two arguments representing the number of rows and columns in the grid, here *SIDE* and *SIDE*. The constant *SIDE* is declared at line 11 and has the value 8. Still at line 30, we pass the *GridLayout* anonymous object as the argument of the *setLayout* method. Thus, components

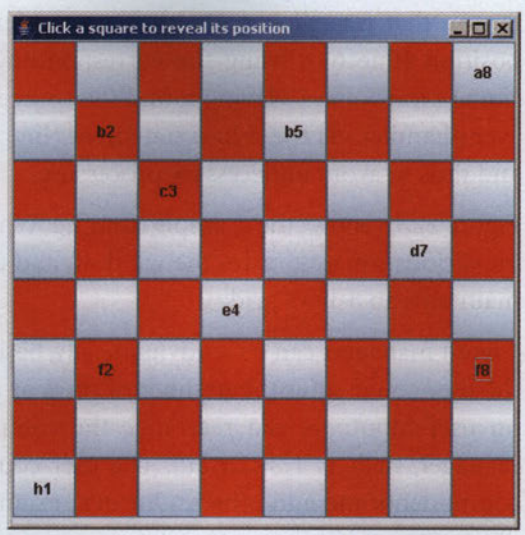


Figure 12.18
Running Example 12.16

will be added cell by cell to an 8-by-8 grid, with each row being filled before starting the next row.

At line 34, we declare and instantiate the listener *bh*, a *ButtonHandler* object reference. Since we are interested in events related to buttons, our *ButtonHandler private* inner class *implements* the *ActionListener* interface and overrides the *actionPerformed* method.

At line 32, we instantiate the two-dimensional array *squares*. Because we have a two-dimensional array of *JButtons*, we use nested *for* loops at lines 36–53 to instantiate the *JButtons*, add them to the content pane, and register the listener on all of the buttons. Inside the inner loop, the value of the expression *(i + j)* will alternate between an even and odd number. Accordingly, at lines 43–45, we set the background of every other button to red.

In the *actionPerformed* method, we use nested *for* loops at lines 63–73 to identify the source of the event; that is, which button the user clicked. When a button is clicked, the condition of the *if* statement at line 67 will evaluate to *true* when the *i* and *j* indices are such that *squares[i][j]* is the

button that was clicked. We then set the text of that button to its board position (line 69), using the corresponding element in the two-dimensional array *names*. Having found the source of the event, we then exit the event handler via the *return* statement (line 70) to interrupt the *for* loops, and thus to avoid unnecessary processing.

Figure 12.18 shows the example running. When you run this program, click on various squares; resize the window, and check that the layout manager maintains an 8-by-8 grid.

Layout managers can be set dynamically, based on user input. For example, the user could enter the number of rows or columns of the grid. Or based on user input, we can rearrange the components using another layout manager, such as *FlowLayout*. The user could also instruct us to remove components and add others. Our next example, the Tile Puzzle game, will illustrate some of these capabilities.

In the Tile Puzzle game, eight tiles displaying the digits 1 through 8 are scrambled on a 3-by-3 grid, leaving one cell empty. Any tile adjacent to the empty cell can be moved to the empty cell by clicking on the numbered tile. The goal is to rearrange the tiles so that the numbers are in the correct order, as shown in Figure 12.19.

The Tile Puzzle game can also be played on a 4-by-4, a 5-by-5, and more generally, an *n*-by-*n* grid. Example 12.18 will set up a 3-by-3 grid for the

Figure 12.19
The Winning Position of a 3-by-3 Tile Puzzle Game

first game, then will randomly select a 3-by-3, a 4-by-4, a 5-by-5, or a 6-by-6 grid for subsequent games. Once you understand this example, you can modify it to allow the user to specify the size of the grid.

Before designing the GUI class, we first code a class to encapsulate the functionality of the tile puzzle game; this class (Example 12.17) handles the creation of a game of a given size, enables play, and enforces the rules of the game.

The instance variable *tiles* (line 7), a two-dimensional array of *Strings*, stores the state of the puzzle. Each element of *tiles* will be a cell in the puzzle grid. The instance variable *side*, declared at line 8, represents the size of the grid. The instance variables *emptyRow* and *emptyCol*, declared at lines 9–10, identify the empty cell in the puzzle grid.

The constructor (lines 12–18) calls the *setUpGame* method (lines 20–44), passing the size of the grid (*newSide*) as an argument. We also call the *setUpGame* method before starting each new game in the GUI class.

Inside the *setUpGame* method, we assign *newSide* to *side*. Rather than randomly generating each tile label, which would complicate this example, we assign the labels to the tiles in descending order using nested *for* loops (lines 33–41). We set the empty cell to the last cell in the grid at lines 42–43.

The *tryToPlay* method (lines 71–89) first checks if the play is legal by calling the *possibleToPlay* method (line 77). If the *possibleToPlay* method returns *true*, we proceed with the play and return *true*; otherwise, we return *false*. Playing means swapping the values of the empty cell (*emptyRow*, *emptyCol*) and the cell that was just played, represented by the two parameters of *tryToPlay*, *row*, and *col*.

The *possibleToPlay* method is coded at lines 91–101. If the play is legal—that is, if the tile played is within one cell of the empty cell—the method returns *true*; otherwise, the method returns *false*.

The *won* method (lines 103–119) checks if the tiles are in order. If the user has won the game, the *won* method returns *true*; otherwise, the method returns *false*.

Figure 12.20 shows the game in progress.

```
1 /* TilePuzzle class
2 *  Anderson, Franceschi
3 */
4
5 public class TilePuzzle
6 {
7  private String [ ][ ] tiles;
8  private int side; // grid size
9  private int emptyRow;
10 private int emptyCol;
11
12 /** constructor
13 *  @param  newSide  grid size
14 */
15 public TilePuzzle( int newSide )
16 {
17  setUpGame( newSide );
18 }
19
20 /** setUpGame
21 *  @param  newSide   grid size
22 */
23 public void setUpGame( int newSide )
24 {
25    if ( newSide < 1 )
26      side = 3;
27    else
28      side = newSide;
29    emptyRow = side - 1;
30    emptyCol = side - 1;
31    tiles = new String[side][side];
32
33  // initialize tiles
34  for ( int i = 0; i < side; i++ )
35  {
36   for ( int j = 0; j < side; j++ )
37   {
38     tiles[i][j] = String.valueOf( ( side * side )
39                                - ( side * i + j + 1 ) );
40   }
41  }
42  // set empty cell label to blank
43  tiles[side - 1][side - 1] = "";
44 }
45
46 /** getSide
47 *  @return side
```

```
48 */
49 public int getSide( )
50 {
51   return side;
52 }
53
54 /** getTiles
55 *  @return a copy of tiles
56 */
57 public String [ ][ ]  getTiles( )
58 {
59  String[ ][ ] copyOfTiles = new String[side][side] ;
60
61  for ( int i = 0; i < side; i++ )
62  {
63   for ( int j = 0; j < side; j++ )
64   {
65     copyOfTiles[i][j] = tiles[i][j];
66   }
67  }
68  return copyOfTiles;
69 }
70
71 /** tryToPlay
72 *  enable play if play is legal
73 *  @return true if the play is legal, false otherwise
74 */
75 public boolean tryToPlay( int row, int col )
76 {
77  if ( possibleToPlay( row, col ) )
78  {
79   // play: switch empty String and tile label at row, col
80   tiles[emptyRow][emptyCol] = tiles[row][col];
81   tiles[row][col] =  "";
82   // update emptyRow and emptyCol
83   emptyRow = row;
84   emptyCol = col;
85   return true;
86  }
87  else
88   return false;
89 }
90
91 /** possibleToPlay
92 *  @return true if the play is legal, false otherwise
93 */
```

```
94  public boolean possibleToPlay( int row, int col )
95  {
96   if ( ( col == emptyCol && Math.abs( row - emptyRow ) == 1 )
97          || ( row == emptyRow && Math.abs( col - emptyCol ) == 1 ) )
98     return true;
99   else
100    return false;
101 }
102
103 /** won
104 *  @return true if the tiles are all at the right place, false otherwise
105 */
106 public boolean won( )
107 {
108  for ( int i = 0; i < side ; i++ )
109  {
110   for ( int j = 0; j < side; j++ )
111   {
112     if ( !( tiles[i][j].equals(
113                        String.valueOf( i * side + j + 1 ) ) )
114                        && ( i != side - 1 || j != side - 1 )  )
115        return false;
116   }
117  }
118  return true;
119 }
120 }
```

EXAMPLE 12.17 ***TilePuzzle*** **Class**

Figure 12.20
Running Example 12.18

Example 12.18 shows the GUI client class that enables a user to play the tile puzzle game.

Each cell of the grid is a button. At line 14, we declare a two-dimensional array of *JButtons* named *squares*. Each element of *squares* will be a cell in the game grid. The *TilePuzzle* instance variable *game*, declared at line 15, represents the tile puzzle game that we will display inside the window. When the user plays, we will call the various methods of the *TilePuzzle* class to enforce the rules of the game.

The constructor (lines 17–22) instantiates *game* and calls the *setUpGameGUI* method. We also call the *setUpGameGUI* method (lines 24–44) before starting each new game.

Inside the *setUpGameGUI* method, we remove all the components from the content pane by calling the *removeAll* method of the *Container* class (line 27) and set the layout manager (line 28) as a *GridLayout* with a grid size equal to the *side* instance variable of *game*. We instantiate the *squares* array and our event handler at lines 30 and 32. After that, we use nested *for* loops (lines 34–44) to instantiate each button, add it to the container, and register the event handler. The *squares* array parallels the *tiles* array of the *TilePuzzle* class; the *squares* buttons are labeled with the values of *tiles*.

Inside the *private* class *ButtonHandler* (lines 71–88), the *actionPerformed* method identifies the button that was clicked and calls the *tryToPlay* method at line 81 with the row and column arguments corresponding to the button that originated the event. If the play is legal, the *tryToPlay* method returns *true* and we update the puzzle grid by calling the *update* method at line 82.

Inside the *update* method (lines 50–69), we first update the *squares* button array (lines 52–58). We then test if this move solved the puzzle by calling the *won* method (line 60) with our instance variable *game*. If the *won* method returns *true*, we congratulate the user by popping up a dialog box at lines 62–63.

We then randomly generate a grid size between 3 and 6 (lines 64–65) for the next game and call the *setUpGame* method with *game* at line 66 and the *setUpGameGUI* method to begin the game with the new grid size at line 67.

```
1 /** Using GridLayout dynamically
2 *  Anderson, Franceschi
3 */
4
```

```
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8 import java.util.Random;
9
10 public class TilePuzzleClient extends JFrame
11 {
12  private Container contents;
13
14  private JButton [ ][ ] squares;
15  private TilePuzzle game; // the tile puzzle game
16
17  public TilePuzzleClient( int newSide )
18  {
19   super( "The Tile Puzzle Game" );
20   game = new TilePuzzle( newSide );
21   setUpGameGUI( );
22  }
23
24  private void setUpGameGUI( )
25  {
26   contents = getContentPane( );
27   contents.removeAll( ); // remove all components
28   contents.setLayout( new GridLayout( game.getSide( ),
29       game.getSide( ) ) );
30   squares = new JButton[game.getSide( )][game.getSide( )];
31
32   ButtonHandler bh = new ButtonHandler( );
33
34   // for each button: instantiate button with appropriate button label,
35   // add to container, and register listener
36   for ( int i = 0; i < game.getSide( ); i++ )
37   {
38    for ( int j = 0; j < game.getSide( ); j++ )
39    {
40     squares[i][j] = new JButton( game.getTiles( ) [i][j] );
41     contents.add( squares[i][j] );
42     squares[i][j].addActionListener( bh );
43    }
44   }
45
46   setSize( 300, 300 );
47   setVisible( true );
48  }
49
50  private void update( int row, int col )
51  {
```

```
52   for ( int i = 0; i < game.getSide( ); i++ )
53   {
54    for ( int j = 0; j < game.getSide( ); j++ )
55    {
56      squares[i][j].setText( game.getTiles( ) [i][j] );
57    }
58   }
59
60   if ( game.won( ) )
61   {
62     JOptionPane.showMessageDialog( TilePuzzleClient.this,
63              "Congratulations! You won!\nSetting up new game" );
64     Random random = new Random( );
65     int sideOfPuzzle = 3 + random.nextInt( 4 );
66     game.setUpGame( sideOfPuzzle );
67     setUpGameGUI( );
68   }
69 }
70
71 private class ButtonHandler implements ActionListener
72 {
73  public void actionPerformed( ActionEvent ae )
74  {
75   for ( int i = 0; i < game.getSide( );  i++ )
76   {
77    for ( int j = 0; j < game.getSide( ); j++ )
78    {
79      if ( ae.getSource( ) == squares[i][j] )
80      {
81        if ( game.tryToPlay( i, j ) )
82           update( i, j );
83        return;
84      }
85    }
86   }
87  }
88 }
89
90 public static void main( String [ ] args )
91 {
92  TilePuzzleClient myGame = new TilePuzzleClient(  3  );
93  myGame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
94 }
95 }
```

EXAMPLE 12.18 Setting *GridLayout* Dynamically in the Tile Puzzle Game

12.14 Layout Managers: *BorderLayout*

A *BorderLayout* organizes a container into five areas: north, south, west, east, and center, with each area holding at most one component. Thus, for each area, you can add one component or no component. The size of each area expands or contracts depending on the size of the component in that area, the sizes of the components in the other areas, and whether the other areas contain a component. Figure 12.21 shows the areas in a sample border layout.

In contrast to the *FlowLayout* and *GridLayout* layout managers, the order in which we add components to a *BorderLayout* is not important. We use a second argument in the *add* method to specify the area in which to place the component. This *add* method of the *Container* class has the API shown in Table 12.23.

Figure 12.21
***BorderLayout* Areas**

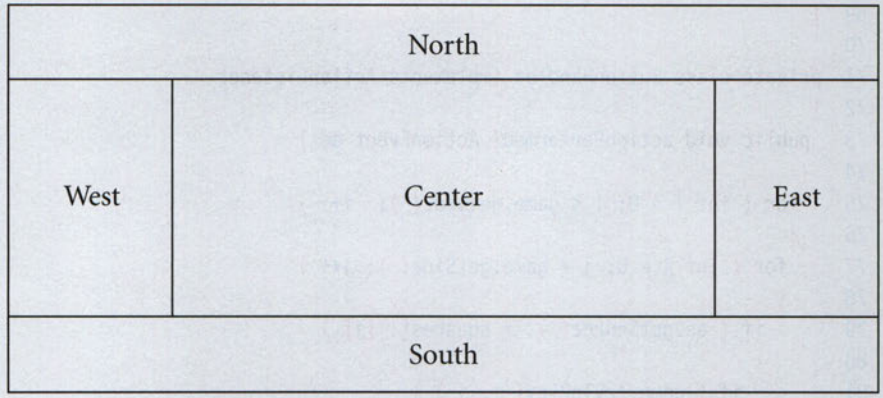


TABLE 12.23 The *Container add* Method for Using a *BorderLayout*

Return value	Method name and argument list
`void`	`add( Component c, Object borderlayoutArea )` adds the component *c* to the container. The area defined by *borderlayoutArea* can be specified using any of the following *static String* constants in the *BorderLayout* class: *NORTH*, *SOUTH*, *EAST*, *WEST*, *CENTER*.

TABLE 12.24 *BorderLayout* Constructors

Constructors	
Class	**Constructor**
`BorderLayout`	`BorderLayout( )` creates a new border layout with no gaps between components.
`BorderLayout`	`BorderLayout( int hGap, int vGap )` creates a border layout with a horizontal gap of *hGap* pixels between components and a vertical gap of *vGap* pixels between components. Unlike *GridLayout*, horizontal gaps are not placed at the left and right edges, and vertical gaps are not placed at the top and bottom edges.

Two *BorderLayout* constructors are shown in Table 12.24. The *BorderLayout* is the default layout for a *JFrame* class, so if we want to use a border layout for our GUI application, we do not need to instantiate a new layout manager.

Example 12.20 uses a *BorderLayout* to illustrate bidding order in the card game of Bridge. There are four players: North, East, South, and West. Players take turns bidding in clockwise order, starting with the dealer. In this example, we assume the player sitting in the North position is the dealer, so North will bid first. A bid consists of a level and a suit, with the lowest bid being 1 Club and the highest bid being 7 No trump. Each new bid must be higher than the previous bid; that is, a new bid must name a higher-ranked suit at the same level, or any suit at a higher level. Suits are ranked in the following ascending order: Clubs, Diamonds, Hearts, Spades, and No trump. Thus, if the current bid is 3 Hearts, the next possible bids are, in order, 3 Spades, 3 No trump, 4 Clubs, 4 Diamonds, 4 Hearts, and so on. We will use a window managed by a *BorderLayout* manager to simulate sequential bids by our four Bridge players. At any point, a player can "pass" or "double" the previous bid, but for simplicity and to illustrate bidding order only, we do not take these bids into account.

Example 12.19 shows the class encapsulating the functionality of Bridge bidding.

The *String* arrays *suitNames* and *players*, the *int* instance variables *dealer*, *bidder*, *level*, and *suit*, declared and initialized at lines 7–17, will help us keep track of the bidding and enforce the bidding order. The *bidder* instance variable represents the next bidder: its value is used as an index into the *players* array to represent North, East, South, and West, respectively. Similarly, *suit* will be used as an index into the *suitNames* array. The *level* instance variable will store a value between 1 and 7, which are the valid levels of bidding in Bridge. The first legal bid in Bridge is 1 Club so we have initialized *level* to 1 and *suit* to 0.

The method *getPlayer* (lines 32–39), given a parameter representing an index, returns the corresponding player, *getBidder* (lines 41–47) returns the current bidder, and *isBiddingAllowed* (lines 49–55) returns *true* if the bidding level is between 1 and 7 and *false* if it is not.

The method *nextBid* (lines 57–81) returns the current bid and sets up the next bid. After checking that the bidding can continue (line 63), we define the current bid at lines 65–69, and we set up the values for the next bid (lines 71–75). We increment *bidder*, using the modulus operator to cycle through the valid indexes of the *players* array. We also increment *suit*, cycling through the possible indexes for the *suitNames* array. Whenever the next suit index is 0, we increment *level*.

```
1 /** BridgeBidding class
2 *   Anderson, Franceschi
3 */
4
5 public class BridgeBidding
6 {
7  private String [ ] suitNames =
8   { "Club", "Diamond", "Heart", "Spade", "No trump" };
9  private String [ ] players =
10  { "North", "East", "South", "West" };
11  // next bidder
12  private int bidder;          // the next bidder
13  private int dealer;          // the dealer
14
15  // bidding will open at 1 Club
16  private int level = 1;              // current level
17  private int suit = 0;               // index of current suit
18
19  /** Constructor
```

```
20  *  @param  newDealer   dealer
21  */
22  public BridgeBidding( int newDealer )
23  {
24   if ( newDealer >= 0 && newDealer < players.length )
25    dealer = newDealer;
26   else
27    dealer = 0;
28   players[dealer] += " - Dealer";
29   bidder = dealer;
30  }
31
32  /** getPlayer
33  *  @param  player   the index of the player
34  *  @return a String, the name of the player
35  */
36  public String getPlayer( int player )
37  {
38   return players[player];
39  }
40
41  /** getBidder
42  *  @return bidder
43  */
44  public int getBidder( )
45  {
46   return bidder;
47  }
48
49  /** isBiddingAllowed
50  *  @return true if level is strictly less than 8, false otherwise
51  */
52  public boolean isBiddingAllowed( )
53  {
54   return ( level >= 1 && level <= 7 );
55  }
56
57  /** nextBid
58  * Returns current bid and sets up next bid
59  *  @return a String, the current bid
60  */
61  public String nextBid( )
62  {
63   if ( isBiddingAllowed( ) )
64   {
```

```
65    String currentBid = level + " " + suitNames[suit];
66
67    // add an "s" if level > 1 and suit is not No trump
68    if ( level > 1 && suit != suitNames.length - 1 )
69     currentBid += "s";
70
71    // set up next bid
72    bidder = ( bidder + 1 ) % players.length;
73    suit = ( suit + 1 ) % suitNames.length;
74    if ( suit == 0 )
75     level++;
76
77    return currentBid;
78   }
79   else
80    return "Pass";
81  }
82 }
```

EXAMPLE 12.19 *BridgeBidding* Class

In Example 12.20, we declare four buttons representing the players (line 12); we will place one button each in the north, east, south, and west areas of the border layout. We use the label *bid*, declared at line 13, to display the current bid in the center area. The *BridgeBidding* instance variable *bidding*, declared at line 15, represents the bidding sequence that we will display inside the window.

At line 23, we set the layout of the content pane to be a *BorderLayout*. Note that this statement is optional, because *BorderLayout* is the default layout manager for a subclass of the *JFrame* class.

When instantiating *bid* at lines 31–32, we use an overloaded *JLabel* constructor. We use the *static int* constant *CENTER* of the *SwingConstants* interface to center the text of the label within the component. By default, the text for a *JLabel* is left-aligned. Because the text on a *JButton* is centered by default, we can simply use the constructor with one argument when we instantiate the *JButtons* for the players.

We add the four buttons and the label to the content pane in the appropriate areas at lines 34–39. The order in which we add the components is not important because we specify a border layout area as the second argument of the *add* method.

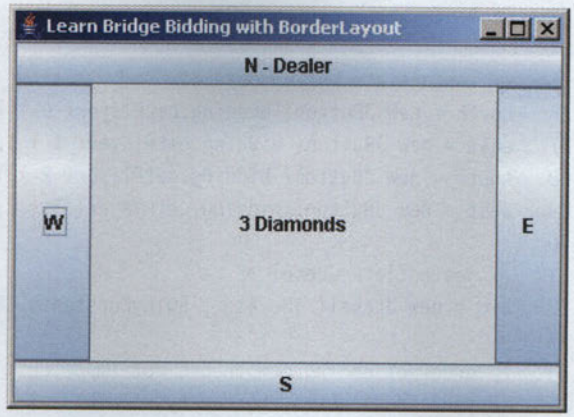


Figure 12.22
Running Example 12.20

In our *ButtonHandler* event handler (lines 51–69), the *actionPerformed* method first determines the button that fired the event, translating the source button to an integer value between 0 and 3. That value is assigned to the local variable *source*. We want to allow only bids that are legal and only from the player whose turn it is. If *source* is the bidder and bidding is allowed (lines 65–66), we display a new bid (line 67); otherwise, we do not.

Figure 12.22 shows the Bridge bidding example running.

```
1 /** Using BorderLayout to display Bridge bidding
2 *   Anderson, Franceschi
3 */
4
5 import javax.swing.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 public class BridgeBiddingClient extends JFrame
10 {
11  private Container contents;
12  private JButton north, east, south, west;
13  private JLabel bid;
14
15  private BridgeBidding bidding;
16
17  public BridgeBiddingClient( )
18  {
19   super( "Learn Bridge Bidding with BorderLayout" );
20   bidding = new BridgeBidding( 0 );
21
22   contents = getContentPane( );
```

```
23   contents.setLayout( new BorderLayout( ) );  // optional
24
25   // instantiate button objects
26   north = new JButton( bidding.getPlayer( 0 ) );
27   east = new JButton( bidding.getPlayer( 1 ) );
28   south = new JButton( bidding.getPlayer( 2 ) );
29   west = new JButton( bidding.getPlayer( 3 ) );
30
31   // instantiate JLabel
32   bid = new JLabel( "No bid", SwingConstants.CENTER );
33
34   // order of adding components not important
35   contents.add( north, BorderLayout.NORTH );
36   contents.add( east, BorderLayout.EAST );
37   contents.add( south, BorderLayout.SOUTH );
38   contents.add( west, BorderLayout.WEST );
39   contents.add( bid, BorderLayout.CENTER );
40
41   ButtonHandler bh = new ButtonHandler( );
42   north.addActionListener( bh );
43   east.addActionListener( bh );
44   south.addActionListener( bh );
45   west.addActionListener( bh );
46
47   setSize( 350, 250 );
48   setVisible( true );
49  }
50
51  private class ButtonHandler implements ActionListener
52  {
53   public void actionPerformed( ActionEvent ae )
54   {
55    int source = 0;
56    if ( ae.getSource( ) == north )
57      source = 0;
58    else if ( ae.getSource( ) == east )
59      source = 1;
60    else if ( ae.getSource( ) == south )
61      source = 2;
62    else if ( ae.getSource( ) == west )
63      source = 3;
64
65    if ( source == bidding.getBidder( )
66                && bidding.isBiddingAllowed( ) )
67     bid.setText( bidding.nextBid( ) ); // set label to current bid
```

```
68   }
69  }
70
71  public static void main( String [ ] args )
72  {
73   BridgeBiddingClient bbGui = new BridgeBiddingClient( );
74   bbGui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
75  }
76 }
```

EXAMPLE 12.20 Using *BorderLayout* to Illustrate Bridge Bidding Order

12.15 Using Panels to Nest Components

Components can be nested. Indeed, since the *JComponent* class is a subclass of the *Container* class, a *JComponent* object is a *Container* object as well. As such, it can contain other components. We can use this feature to achieve more precise layouts.

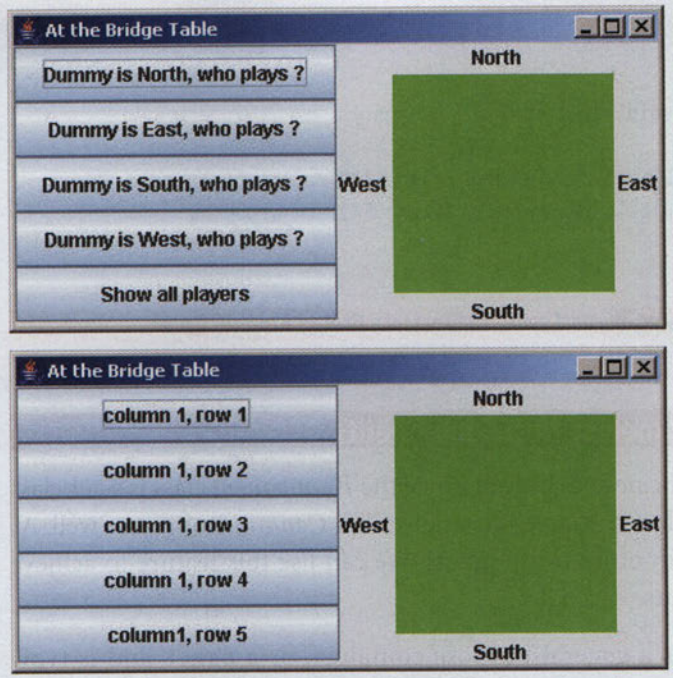
The *JPanel* class is a general-purpose container, or a **panel**, and is typically used to hold other components. When nesting components, we usually place several components into a panel, and place the panel into the content pane of the current *JFrame*. Each panel has its own layout manager, which can vary from panel to panel, and the content pane for the *JFrame* application has its own layout manager, which can be different from any of the panels. We can even have multiple levels of nesting, as needed.

Figure 12.23 shows a window that uses multiple panels, along with the underlying layout of the window.

The content pane of the window is using a *GridLayout* with one row and two columns. In the first column, we have defined a panel managed by a *GridLayout* having five rows and one column. We placed five buttons into the panel, then placed the panel into the first column of the content pane grid. In the second column, we have defined a panel managed by a *BorderLayout*. We put five components into the panel and added the panel as the second column of the content pane grid. Thus, we have two panels, each of which has a different layout manager, and these two panels are added to the content pane, which is itself managed by a *GridLayout*.

In a Bridge game, depending on the bidding, one of the four players will be the "dummy" and will not participate in the play for the current hand. We

Figure 12.23
Sample Window and Underlying Layout



illustrate this concept in Example 12.21. The panel in column 2 of the content pane grid uses a *BorderLayout* layout manager to show four players around the Bridge table. The panel in column 1 of the content pane grid uses a *GridLayout* to provide five buttons.

If the user clicks on one of the first four buttons, we make the corresponding player disappear from the Bridge table (in column 2). If the user clicks the fifth button, we make all players visible. Thus, the actions we want to perform are different depending on whether the user clicks on one of the first four buttons or the "Show all players" button. So it makes sense to write two event handlers: one that listens to the first four buttons, and a second event handler that listens to the fifth button only.

In Example 12.21, we first define the layout manager for our content pane to be a *GridLayout* with one row and two columns (line 34). Then we set up the panel for the first column at lines 37–38, by instantiating the *JPanel* named *questionPanel* and setting the layout manager as a *GridLayout* with five rows and one column. At lines 40–48, we instantiate the event handler for the question buttons, instantiate the buttons, register the listener on the buttons, and add the buttons to the *questionPanel* in rows 1–4.

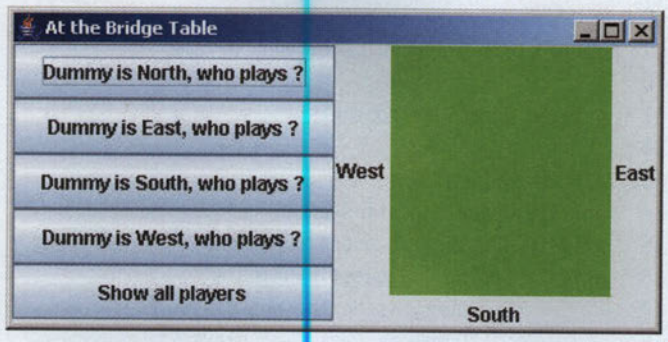


Figure 12.24
User Clicked Top Button

In lines 50–54, we set up the fifth row in this grid by instantiating the *reset* button, instantiating our separate listener for this button, and adding the button in the fifth row.

In lines 56–73, we set up column 2, by instantiating the *JPanel gamePanel* and setting its layout to a *BorderLayout* with horizontal and vertical gaps of 3 pixels between elements. At lines 69–73, we add one component to each area of the *BorderLayout*, representing the four players and the Bridge table.

Finally, we add the two panels to the content pane of the application window at lines 75–77.

The *QuestionButtonHandler* event handler (lines 83–96) checks which question button was clicked and sets the visibility of the appropriate label in the *gamePanel* to *false*. The other three labels are made visible, in case they were hidden as a result of a previous button click.

The second event handler, *ResetButtonHandler* (lines 98–106), does not need to check the source of the event because the listener is registered only on the *reset* button component. Its job is to make every player's button visible.

When the application begins, we display the window shown in Figure 12.23. Figure 12.24 shows the window after the user has clicked on the top button ("Dummy is North, who plays?").

```
1  /** Nesting components using layout managers
2  *   Anderson, Franceschi
3  */
4
5  import javax.swing.*;
6  import java.awt.*;
7  import java.awt.event.*;
```

```
8
9  public class BridgeRules extends JFrame
10 {
11  private Container contents;
12
13  // 1st row, column 1
14  private JPanel questionPanel;
15  private JButton [ ] questionButtons;
16  private String [ ] questionNames = {
17              "Dummy is North, who plays ?",
18              "Dummy is East, who plays ?",
19              "Dummy is South, who plays ?",
20              "Dummy is West, who plays ?" };
21
22  private JButton reset;
23
24  // 1st row, column 2
25  private JPanel gamePanel;
26  private JLabel gameTable;
27  private JLabel [ ] gameLabels;
28
29  public BridgeRules( )
30  {
31   super( "At the Bridge Table" );
32
33   contents = getContentPane( );
34   contents.setLayout( new GridLayout( 1, 2 ) );
35
36   // 1st row, col 1: question buttons and reset button
37   questionPanel = new JPanel( );
38   questionPanel.setLayout( new GridLayout( 5, 1 ) );
39
40   QuestionButtonHandler qbh = new QuestionButtonHandler( );
41   questionButtons = new JButton[questionNames.length];
42
43   for ( int i = 0; i < questionNames.length; i ++ )
44   {
45      questionButtons[i] = new JButton( questionNames[i] );
46      questionButtons[i].addActionListener( qbh );
47      questionPanel.add( questionButtons[i] );
48   }
49
50   reset = new JButton( "Show all players" );
51   ResetButtonHandler rbh = new ResetButtonHandler( );
```

```
52    reset.addActionListener( rbh );
53
54    questionPanel.add( reset );
55
56    // 1st row, column 2: gamePanel contains the players and table
57    gamePanel = new JPanel( );
58    gamePanel.setLayout( new BorderLayout( 3, 3 ) );
59    gameLabels = new JLabel[4];
60    gameLabels[0] = new JLabel( "North", SwingConstants.CENTER );
61    gameLabels[1] = new JLabel( "East", SwingConstants.CENTER );
62    gameLabels[2] = new JLabel( "South", SwingConstants.CENTER );
63    gameLabels[3] = new JLabel( "West", SwingConstants.CENTER );
64
65    gameTable = new JLabel( );
66    gameTable.setBackground( Color.GREEN );
67    gameTable.setOpaque( true );
68
69    gamePanel.add( gameLabels[0], BorderLayout.NORTH );
70    gamePanel.add( gameLabels[1], BorderLayout.EAST );
71    gamePanel.add( gameLabels[2], BorderLayout.SOUTH );
72    gamePanel.add( gameLabels[3], BorderLayout.WEST );
73    gamePanel.add( gameTable, BorderLayout.CENTER );
74
75    // add panels to content pane
76    contents.add( questionPanel );
77    contents.add( gamePanel );
78
79    setSize( 410, 200 );
80    setVisible( true );
81  }
82
83  private class QuestionButtonHandler
84                        implements ActionListener
85  {
86    public void actionPerformed( ActionEvent ae )
87    {
88      for ( int i = 0; i < questionButtons.length; i++ )
89      {
90        if ( ae.getSource( ) == questionButtons[i] )
91          gameLabels[i].setVisible( false );
92        else
93          gameLabels[i].setVisible( true );
94      }
95    }
96  }
```

```
97
98  private class ResetButtonHandler
99                              implements ActionListener
100 {
101   public void actionPerformed( ActionEvent ae )
102   {
103      for ( int i = 0; i < gameLabels.length; i++ )
104        gameLabels[i].setVisible( true );
105   }
106 }
107
108 public static void main( String [ ] args )
109 {
110   BridgeRules myNestedLayout = new BridgeRules( );
111   myNestedLayout.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
112 }
113 }
```

EXAMPLE 12.21 More Bridge Rules with Nested Components

Skill Practice
with these end-of-chapter questions

12.18.1 Multiple Choice Exercises

Questions 9, 10, 11, 12, 13, 14, 15

12.18.2 Reading and Understanding Code

Question 17

12.18.3 Fill In the Code

Questions 30, 33, 34, 35, 36, 37, 38, 39, 40, 41

12.18.5 Debugging Area

Question 48

12.18.6 Write a Short Program

Questions 58, 59, 60, 61, 62

12.18.8 Technical Writing

Question 73

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of working with layout managers. Click on the link for Chapter 12 to start the movie.

12.16 Programming Activity 2: Working with Layout Managers

In this Programming Activity, you will complete the implementation of a version of the Tile Puzzle game (Examples 12.17 and 12.18) using a more complex GUI. As it stands, the application compiles and runs, but is missing a lot of code. Figure 12.25 shows the window that will open when you run the application without adding your code.

Once you have completed the five tasks of this Programming Activity, you should see the window in Figure 12.26 when you run your program and click on the "3-by-3" button.

When you click on one of the buttons labeled "3-by-3", "4-by-4", or "5-by-5", the tile puzzle will reset to a grid of that size.

In addition to the *TilePuzzle* class, we provide you with a prewritten *Game* class, which encapsulates a Tile Puzzle game in a panel. We have implemented the *Game* class as a *JPanel* component, so you can add it to your window as you would any other panel. It has two important methods,

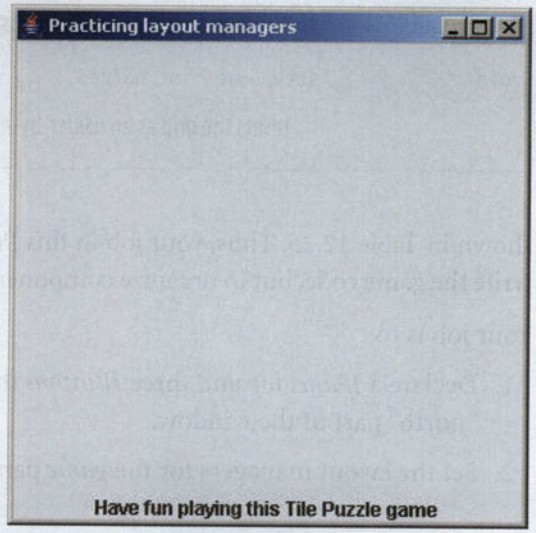


Figure 12.25
The Starting Window When Running the Prewritten Code

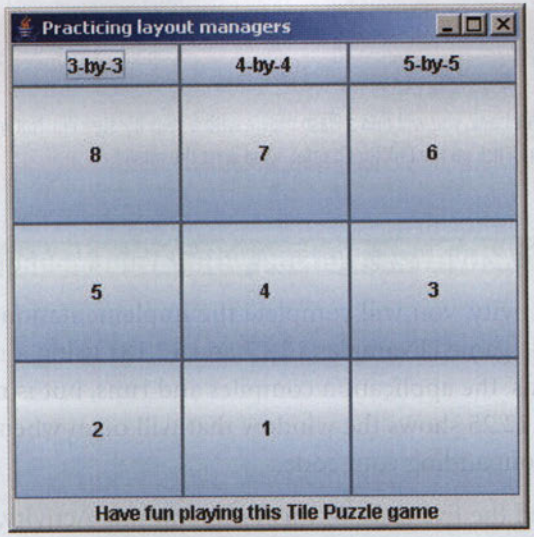


Figure 12.26
The Starting Window When the Activity Is Completed

TABLE 12.25 The *Game* Class API

Game Class API	
Constructor	
Class	**Constructor**
`Game`	`Game( int nSides )` instantiates a tile puzzle as an *nSides*-by-*nSides* grid
Method	
Return value	**Method name and argument list**
`void`	`setUpGame( int nSides )` resets the grid as an *nSides*-by-*nSides* grid

shown in Table 12.25. Thus, your job in this Programming Activity is not to write the game code, but to organize components in a window.

Your job is to:

1. Declare a *JPanel top* and three *JButtons* that will be added to the "north" part of the window.
2. Set the layout managers for the *game* panel and the *top* panel.

3. Add the *top* and the *game* panels.
4. Code an appropriate *private* listener class.
5. Instantiate the listener and register it on the appropriate components.

Instructions

Copy the source files in the Programming Activity 2 directory for this chapter to a directory on your computer.

1. Write the code to declare the needed instance variables. Load the *NestedLayoutPractice.java* source file and search for five asterisks in a row (*****). This will position you to the instance variables declaration.

```
// ***** Task 1: declare a JPanel named top
// also declare three JButton instance variables
// that will be added to the JPanel top
// these buttons will determine the grid size of the game:
//  3-by-3, 4-by-4, or 5-by-5

// task 1 ends here
```

2. Next, write the code to set the layout manager of the window and add the component *game* in the center of the window. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 2: student code starts here
// instantiate the BorderLayout manager bl

// set the layout manager of the content pane contents to bl

game = new Game( 3 );  // instantiating the Game object
// add game to the center of the content pane

// task 2 ends here
```

3. Next, write the code to instantiate the *JPanel top* component, set its layout, instantiate the buttons from task 1, add them to *top*, and finally add *top* as the north component for our overall window. In the

NestedLayoutPractice.java source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 3: Student code restarts here

// instantiate the JPanel component named top

// set the layout of top to a 1-by-3 grid

// instantiate the JButtons that determine the grid size

// add the buttons to JPanel top

// add JPanel top to the content pane as its north component

// task 3 ends here
```

4. Next, write the code for the *private* inner class that implements the appropriate listener. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you between the constructor and the *main* method.

```
// ***** Task 4: Student code restarts here
// create a private inner class that implements ActionListener
// your method should identify which of the 3 buttons was the
//   source of the event.
// depending on which button was pressed,
//   call the setUpGame method of the Game class
//   with arguments 3, 4, or 5
// the API of that method is:
//   public void setUpGame( int nSides )

// task 4 ends here
```

5. Next, write the code to declare and instantiate a listener, and register it on the appropriate components. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 5: Student code restarts here
// Note: search for and complete Task 4 before performing this task
// declare and instantiate an ActionListener
```

```
// register the listener on the 3 buttons
//   that you declared in Task 1

// task 5 ends here
```

After completing each task, compile your code.

When you have finished writing all the code, compile the source code and run the *NestedLayoutPractice* application. Try clicking on the three buttons that you added.

? DISCUSSION QUESTIONS

1. Identify the various layout managers you used and the panels they manage.
2. Explain why the East and West areas do not show up on the window.

12.17 Chapter Summary

CHAPTER SUMMARY

- A graphical user interface allows the user to interact with an application through mouse clicks, mouse movements, the keyboard, and visual input components.
- The *JFrame* class provides the capabilities for creating a window that will hold GUI components.
- A constructor in a GUI application should call the constructor of the *JFrame* superclass, get an object reference to the content pane, set the layout manager, instantiate each component and add it to the window, set the size of the window, and make the window visible.
- A component is an object that has a graphical representation and that displays information, collects data from the user, or allows the user to initiate program functions.
- Java provides a set of GUI components in the *javax.swing* package.
- A *JLabel* component can display text or an image.
- The *FlowLayout* layout manager arranges components in rows from left to right, starting a new row whenever a newly added component does not fit into the current row. If the window is resized, the components are rearranged.

CHAPTER SUMMARY

- Event-driven programming consists of setting up interactive components and event handlers and responding to events generated by user interaction with the components.
- To allow a user to interact with a component, we need to instantiate an object of that class, write an event handler class (called a listener), instantiate an object of the event handler class, and register that listener on the component.
- Event handlers implement a listener interface in the *java.awt.event* or the *javax.swing.event* package. The listener methods receive as a parameter an event object, which encapsulates information about the user interaction. The *getSource* method can be called to determine which component fired the event.
- Event handlers usually are instantiated as *private* inner classes so the handler will have access to the components of the application.
- A *JTextField* component displays a text field for user input. A *JPasswordField* component accepts user input without echoing the characters typed. A *JTextArea* component displays a multiple-line text input area. Events fired by the *JTextField* and *JPasswordField* components generate an *ActionEvent* object and require an *ActionListener* to handle the event.
- A *JButton* component implements a command button used for initiating operations. Clicking on a *JButton* component generates an *ActionEvent* object and requires an *ActionListener* to handle the event.
- *JRadioButton* components allow the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected option. *JRadioButton* components need to be added to a *ButtonGroup*, which manages the mutual exclusivity of the buttons. *JCheckBox* components are toggle buttons; successive clicks on a *JCheckBox* component alternately select and deselect that option. Events fired by these components generate an *ItemEvent* object and use an *ItemListener* to handle the event.
- The *JList* class encapsulates a list from which the user can select one or multiple items. Selecting an item from a *JList* component generates a *ListSelectionEvent* object and requires a *ListSelectionListener* to handle the event.

- A *JComboBox* implements a drop-down list. The user can select only one item from the list. A *JComboBox* fires an *ItemEvent*, so the event handler must implement the *ItemListener* interface.
- Adapter classes, which *implement* an interface and provide empty bodies for each method, are useful when only one of multiple component actions is processed.
- *MouseAdapter* and *MouseMotionAdapter* are adapter classes for the *MouseListener* and *MouseMotionListener* interfaces, respectively.
- The *GridLayout* layout manager arranges components into equally sized cells in rows and columns.
- A *BorderLayout* layout manager, which is the default layout manager for the *JFrame* class, organizes a container into five areas: north, south, west, east, and center, with each area holding at most one component. The size of each area expands or contracts depending on the size of the component in that area, the sizes of the components in the other areas, and whether the other areas contain a component.
- A *JPanel* component can be used as a general-purpose container. To create a complex arrangement of GUI components, we place several components into a panel and place the panel into the content pane of the current *JFrame*. Each panel and the content pane has its own layout manager.

12.18 Exercises, Problems, and Projects

12.18.1 Multiple Choice Exercises

1. An example of a GUI component class is
   - ❑ *ActionEvent.*
   - ❑ *actionPerformed.*
   - ❑ *JTextField.*
   - ❑ *ActionListener.*
2. What are the primary uses of GUI components? (Check all that apply.)
   - ❑ to display information
   - ❑ to facilitate the coding of methods

EXERCISES, PROBLEMS, AND PROJECTS

- ❑ to let the user control the program
- ❑ to collect information from the user

3. In what package do you find the *JButton*, *JTextField*, and *JComboBox* classes?
   - ❑ *javax.swing*
   - ❑ *java.swing*
   - ❑ *javax.awt*
   - ❑ *java.awt*
   - ❑ *java.io*
4. Components can be hidden.
   - ❑ true
   - ❑ false
5. In order to process an event when the user clicks on a button, what should the programmer do? (Check all that apply.)
   - ❑ Code a class that implements the *ActionListener* interface.
   - ❑ Declare and instantiate an object reference (a listener) of the class above.
   - ❑ Call the *actionPerformed* method.
   - ❑ Register the listener on the button.
6. Assuming everything has been coded correctly, what happens when the user clicks a button?
   - ❑ The *actionPerformed* method executes.
   - ❑ The *JButton* constructor executes.
   - ❑ The *main* method executes.
7. If you visit Oracle's Java website (*www.oracle.com/technetwork/java*) and look at the *KeyListener* interface, you will find that it has three methods: *keyPressed*, *keyTyped*, and *keyReleased*. We want to build a class that implements *KeyListener*. Which one of the three methods should we implement?
   - ❑ *keyPressed* only
   - ❑ *keyReleased* only

- ❑ *keyTyped* only
- ❑ All three methods

8. You are designing a GUI with three buttons; a different action will be taken depending on which button the user clicks. You want to code only one *private* class implementing the *ActionListener* interface. Inside the *actionPerformed* method, which method do you call to determine which button was clicked?
   - ❑ *getButton*
   - ❑ *getSource*
   - ❑ *getOrigin*
9. A class extending the *JFrame* class can also implement a listener interface.
   - ❑ true
   - ❑ false
10. *NewLayout* is a layout manager.
   - ❑ true
   - ❑ false
11. In the following code:

```
GridLayout gl = new GridLayout( 6, 4 );
```

what do the arguments 6 and 4 specify?
   - ❑ 6 refers to the number of columns and 4 to the number of rows.
   - ❑ 4 refers to the number of columns and 6 to the number of rows.
   - ❑ There will be 6 components organized in 4 different areas.
   - ❑ There will be 4 components organized in 6 different areas.
12. What is the maximum number of components that a *BorderLayout* can manage?
   - ❑ 2
   - ❑ 3
   - ❑ 4
   - ❑ 5
   - ❑ 6

13. In the following code:

```
BorderLayout bl = new BorderLayout( 7, 3 );
```

what do the arguments 7 and 3 represent?

- ❑ the horizontal and vertical gaps between the five areas of the component
- ❑ the vertical and horizontal gaps between the five areas of the component
- ❑ the number of rows and columns in the component
- ❑ the number of columns and rows in the component

14. In the following code:

```
contents.add( button, BorderLayout.NORTH );
```

what is the data type of the second argument?

- ❑ *int*
- ❑ *String*
- ❑ *BorderLayout*
- ❑ *NORTH*

15. Components can be nested.

- ❑ true
- ❑ false

12.18.2 Reading and Understanding Code

For Questions 16 to 21, consider the following code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Game extends JFrame
{
 private JButton b1, b2, b3, b4;
 private Container contents;
 public Game( )
 {
```

```
 super( "Play this game" );
 contents = getContentPane( );
 contents.setLayout( new GridLayout( 2, 2 ) );
 b1 = new JButton( "Button 1" );
 b2 = new JButton( "Button 2" );
 b3 = new JButton( "Button 3" );
 b4 = new JButton( "Button 4" );
 contents.add( b1 );
 contents.add( b2 );
 contents.add( b3 );

 MyHandler mh = new MyHandler( );
 b1.addActionListener( mh );
 b2.addActionListener( mh );

 setSize( 400, 400 );
 setVisible( true );
}

private class MyHandler implements ActionListener
{
 public void actionPerformed( ActionEvent ae )
 {
  System.out.println( "Hello" );
  if ( ae.getSource( ) == b2 )
     System.out.println( "Hello again" );
 }
}
public static void main( String [ ] args )
{
 Game g = new Game( );
 g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
}
```

16. How many buttons will be displayed in the window?
17. How are the buttons organized in the window?
18. What is the text in the title bar of the window?
19. What happens when the user clicks on the button that says "Button 1"?
20. What happens when the user clicks on the button that says "Button 2"?
21. What happens when the user clicks on the button that says "Button 3"?

For Questions 22 to 26, consider the following code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Game extends JFrame
{
 private JCheckBox c1, c2, c3;
 private int value1, value2, value3;
 private Container contents;
 public Game( )
 {
  super( "Play this game" );
  contents = getContentPane( );
  contents.setLayout( new FlowLayout( ) );
  c1 = new JCheckBox( "Choice 1" );
  c2 = new JCheckBox( "Choice 2" );
  c3 = new JCheckBox( "Choice 3" );
  contents.add( c1 );
  contents.add( c2 );
  contents.add( c3 );
  MyHandler mh = new MyHandler( );
  c1.addItemListener( mh );
  c2.addItemListener( mh );
  c3.addItemListener( mh );

  setSize( 400, 400 );
  setVisible( true );
 }

 private class MyHandler implements ItemListener
 {
  public void itemStateChanged( ItemEvent ie )
  {
    if ( ie.getSource( ) == c1 )
    {
      if ( ie.getStateChange( ) == ItemEvent.SELECTED )
        value1 = 1;
      else
        value1 = 0;
    }
    else if ( ie.getSource( ) == c2 )
    {
```

```
        if ( ie.getStateChange( ) == ItemEvent.SELECTED )
          value2 = 2;
        else
          value2 = 0;
      }
      else if ( ie.getSource( ) == c3 )
      {
        if ( ie.getStateChange( ) == ItemEvent.SELECTED )
          value3 = 4;
        else
          value3 = 0;
      }
      System.out.println( ( value1 + value2 + value3 ) );
    }
  }
  public static void main( String [ ] args )
  {
    Game g = new Game( );
    g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  }
}
```

22. How many checkboxes will be displayed in the window?
23. How are the checkboxes organized in the window?
24. What happens when the user checks "Choice 3" only?
25. What happens when the user checks "Choice 1" and "Choice 3"?
26. What happens when the user checks all the checkboxes?

12.18.3 Fill In the Code

For Questions 27 to 31, consider the following class:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class A extends JFrame
{
 private Container c;
 private JButton b;
 private JTextField tf;

}
```

27. Inside the constructor, this code assigns the content pane of the frame to the *Container c*:

```
// your code goes here
```

28. Inside the constructor, this code instantiates the button *b* with the text "Button":

```
// your code goes here
```

29. Inside the constructor, this code instantiates the text field *tf*; after instantiation, the text field should be emtpy, but have space for 10 characters:

```
// your code goes here
```

30. Inside the constructor, and assuming that *c* has been assigned the content pane, this code sets the layout manager of the content pane to a 2-by-1 grid layout manager:

```
// your code goes here
```

31. Inside the *actionPerformed* method of a *private* inner class implementing the *ActionListener* interface, this code changes the text of *tf* to "Button clicked" if the button *b* was clicked; otherwise, nothing happens:

```
public void actionPerformed( ActionEvent ae )
{
  // your code goes here

}
```

32. Inside the constructor, this code registers the listener *mh* on the button *b*:

```
// the MyHandler class is a private class implementing
//  ActionListener
MyHandler mh = new MyHandler( );
// your code goes here
```

For Questions 33 to 41, consider the following class:

```
import javax.swing.*;
import java.awt.*;

public class B extends JFrame
{
 private Container c;
 private JPanel p1;
```

```
    private JPanel p2;
    private JButton [ ] buttons;          // length 12
    private JTextField [ ] textfields;    // length 10
    private JLabel label1;
    private JLabel label2;

}
```

Also, assume that all the instance variables have been instantiated and you are coding inside the constructor.

33. This code sets the layout manager of the content pane to be a border layout manager.

```
// your code goes here
```

34. This code adds *label1* to the north area of the window.

```
// your code goes here
```

35. This code adds *label2* to the east area of the window.

```
// your code goes here
```

36. This code sets the layout manager of the panel *p1* to be a 3-by-4 grid layout manager.

```
// your code goes here
```

37. This code places all the buttons of the array buttons inside the panel *p1*.

```
// your code goes here
```

38. This code adds the panel *p1* in the center area of the window.

```
// your code goes here
```

39. This code sets the layout manager of the panel *p2* to a 5-by-2 grid with four pixels between cells, both horizontally and vertically.

```
// your code goes here
```

40. This code places all the text fields of the array *textfields* inside the panel *p2*.

```
// your code goes here
```

41. This code adds the panel *p2* in the west area of the window.

```
// your code goes here
```

12.18.4 Identifying Errors in Code

42. Where is the error in this code sequence?

```
import java.swing.*;

public class MyGame extends JFrame
{
  // some code here
}
```

43. Where is the error in this code sequence?

```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
  // some code here
 private class MyHandler extends ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  { }
 }
}
```

44. Where is the error in this code sequence?

```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 // some code here
 private class MyHandler implements ItemListener
 {
  public void actionPerformed( ActionEvent ae )
  { }
 }
}
```

45. Where is the error in this code sequence?

```
import javax.swing.*;
import java.awt.event.*;

public class MyGame extends JFrame
```

```
{
 // some code here
 private class MyHandler implements ItemListener
 {
  public void itemStateChanged( ActionEvent e )
  { }
 }
}
```

12.18.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

46. You coded the following in the file *MyGame.java:*

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyGame extends JFrame
{
 private Container c;
 private JLabel l;

 public MyGame( )
 {
  super( "My Game" );
  c = getContentPane( );
  c.setLayout( new FlowLayout( ) );
  c.add( l );  // Line 15
 }

 public static void main( String [ ] args )
 {
  MyGame mg = new MyGame( ); // Line 20
 }
}
```

The code compiles, but at run time, you get the following output:

```
Exception in thread "main" java.lang.NullPointerException
        at java.awt.Container.addImpl(Container.java:1074)
        at java.awt.Container.add(Container.java:398)
        at MyGame.<init>(MyGame.java:15)
        at MyGame.main(MyGame.java:20)
```

Explain what the problem is and how to fix it.

47. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;

public class MyGame extends JFrame
{
    public MyGame( )
    {
     super( "My Game" );
     setSize( 400, 400 );
    }

    public static void main( String [ ] args )
    {
     MyGame mg = new MyGame( );
    }
}
```

The code compiles, but at run time, you cannot see the window and the code terminates.

Explain what the problem is and how to fix it.

48. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;
import java.awt.*;

public class MyGame extends JFrame
{
 private Container c;
 private JTextField tf;

 public MyGame( )
 {
  super( "My Game" );
  c = getContentPane( );
  tf = new JTextField( "Hello" );
  c.add( tf, NORTH );
  setSize( 400, 400 );
  setVisible( true );
 }

 public static void main( String [ ] args )
 {
  MyGame mg = new MyGame( );
 }
}
```

When you compile your code, you get the following error:

```
MyGame.java:14: cannot find symbol
     c.add( tf, NORTH );
            ^
   symbol  : variable NORTH
   location: class MyGame
1 error
```

Explain what the problem is and how to fix it.

49. You coded the following in the file *MyGame.java*:

```
import javax.swing.*;
import java.awt.*;

public class MyGame extends JFrame
{
    // Some code here
    private class MyHandler implements ActionListener
    {
     public void actionPerformed( ActionEvent ae )
     { }
    }
}
```

When you compile your code, you get the following error:

```
MyGame.java:7: cannot find symbol
     private class MyHandler implements ActionListener
                                        ^
   symbol: class ActionListener
   location: class MyGame
MyGame.java:9: cannot find symbol
     public void actionPerformed( ActionEvent ae )
                                  ^
   symbol: class ActionEvent
   location: class MyGame.MyHandler
2 errors
```

Explain what the problem is and how to fix it.

50. You coded the following in the *MyGame.java* file:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class MyGame extends JFrame
{
 private JButton b;
 private JTextField tf;
 private Container contents;
 public MyGame( )
 {
  super( "Play this game" );
  contents = getContentPane( );
  contents.setLayout( new GridLayout( 1, 2 ) );
  b = new JButton( "Click here" );
  tf = new JTextField( "Hello" );
  contents.add( b );
  contents.add( tf );
  setSize( 400, 400 );
  setVisible( true );
 }

 private class MyHandler implements ActionListener
 {
  public void actionPerformed( ActionEvent ae )
  {
   tf.setText( "Hi" );
  }
 }
 public static void main( String [ ] args )
 {
   MyGame g = new MyGame( );
   g.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 }
}
```

The code compiles and runs. However, when you click the button, the text in the text field does not change.

Explain what the problem is and how to fix it.

12.18.6 Write a Short Program

51. Write a program that displays a text field and two buttons labeled "uppercase" and "lowercase". When the user clicks on the uppercase button, the text changes to uppercase; when the user clicks on the lowercase button, the text changes to lowercase.

52. Write a program with two radio buttons and a text field. When the user clicks on one radio button, the text changes to lowercase; when the user clicks on the other radio button, the text changes to uppercase.

53. Write a program with two checkboxes and a text field. When no checkbox is selected, no text shows in the text field. When only the first checkbox is selected, the word "hello" shows in lowercase. When only the second checkbox is selected, the word "HELLO" shows in uppercase. When both checkboxes are selected, the word "HeLlO" shows (lower and uppercase letters alternate).

54. Write a program with three radio buttons and a circle. (You can choose whether you draw the circle or if the circle is a label image.) When the user clicks on the first radio button, the circle becomes red. When the user clicks on the second radio button, the circle turns orange. When the user clicks on the third radio button, the circle becomes blue.

55. Write a program with three checkboxes and a circle drawn in black. Like the checkbox example in the chapter, each checkbox represents a color (red, blue, or green). Depending on the checkboxes selected, compute the resulting color as follows. If a checkbox is selected, assign the value 255 to the amount of the corresponding color. If a checkbox is not selected, assign 0 to the amount of the corresponding color. For example, if the checkboxes representing the colors red and blue are selected, the resulting color should be *Color(255, 0, 255)*. Color the circle appropriately.

56. Write a program that simulates a guessing game in a GUI program. Ask the user for a number between 1 and 6 in a text field, then roll a die randomly and tell the user if he or she won. Write the program in such a way that any invalid user input (i.e., not an integer between 1 and 6) is rejected and the user is asked again for input.

57. Write a program that simulates a guessing game in a GUI program. Generate a random number between 1 and 100; that number is hidden from the user. Ask the user for a number between 1 and 100 in a text field, then tell the user whether the number is too high, too low,

or the correct number. Let the user continue guessing until he or she guesses the correct number.

58. Write a program that displays a 5-by-5 grid of buttons, each with a different button label. When the user clicks on a button, its text is changed to "Visible".

59. Write a program that displays a 4-by-6 grid of buttons, each with some unique text. One button is the "winning" button, which your program determines randomly, inside the constructor. When the user clicks on a button, its text is changed to "No" if the button clicked is not the winning button, or to "Won" if the button clicked is the winning button.

60. Same as Exercise 59 with the following additions: Keep track of how many times the user clicks on buttons. If the user has not won after five clicks, the text on the last button clicked should be changed to "Lost". Once the user has lost or won, you should disable the game—that is, the buttons no longer respond to clicks from the user.

61. Write a program that displays in the title of the window the position of the mouse as the user moves the mouse around the window.

62. Write a program that draws a small circle that follows the mouse as the user moves the mouse around the window.

12.18.7 Programming Projects

63. Write a GUI-based tic-tac-toe game for two players.

64. Write a GUI-based program that analyzes a word. The user will type the word in a text field. Provide buttons for the following:

    - ❑ One button, when clicked, displays the length of the word.
    - ❑ Another button, when clicked, displays the number of vowels in the word.
    - ❑ Another button, when clicked, displays the number of uppercase letters in the word.

    For this, you should design and code a separate (non-GUI) class encapsulating a word and its analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

65. Write a GUI-based program that analyzes a soccer game. The user will type the names of two teams and the score of the game in four text fields. You should add appropriate labels and create buttons for the following:

    - ❑ One button, when clicked, displays which team won the game.
    - ❑ Another button, when clicked, displays the game score.
    - ❑ Another button, when clicked, displays by how many goals the winning team won.

    For this, you should design and code a separate (non-GUI) class encapsulating a soccer game, then instantiate an object of that class inside your GUI class and call the various methods as needed.

66. Write a GUI-based program that analyzes a round of golf. You will retrieve the data for 18 holes from a text file. On each line in the file will be the par for that hole (3, 4, or 5) and your score for that hole. Your program should read the file and display a combo box listing the 18 holes. When the user selects a hole, the score for that hole should be displayed in a label. Provide buttons for the following:

    - ❑ One button, when clicked, displays whether your overall score was over par, under par, or par.
    - ❑ Another button, when clicked, displays the number of holes for which you made par.
    - ❑ Another button, when clicked, displays how many birdies you scored (a birdie on a hole is 1 under par).

    For this, you should design and code a separate (non-GUI) class encapsulating the analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

67. Write a GUI-based program that analyzes statistics for tennis players. You will retrieve the data from a text file. On each line in the file will be the name of a player, the player's number of wins for the year, and the player's number of losses for the year. Your program should read the file and display the list of players. When the user selects a player, the winning percentage of the player should be displayed in a label. Provide buttons for the scenarios that follow.

- One button, when clicked, displays which player had the most wins for the year.
- Another button, when clicked, displays which player had the highest winning percentage for the year.
- Another button, when clicked, displays how many players had a winning record for the year.

For this, you should design and code a separate (non-GUI) class encapsulating the tennis statistics analysis, then instantiate an object of that class inside your GUI class and call the various methods as needed.

68. Write a GUI-based program that simulates the selection of a basketball team. You will retrieve the data from a text file containing 10 lines. On each line will be the name of a player. Your program needs to read the file and display 10 checkboxes representing the 10 players. A text area will display the team, made up of the players being selected. A basketball team has five players. Your program should not allow the user to change his or her selection after the team has five players. Every time the user checks or unchecks a checkbox, the team in the text area should be updated accordingly. Provide buttons for the following:

    - One button, when clicked, displays how many players are currently on the team.
    - Another button, when clicked, displays how many players remain unselected.

    For this, you should design and code a separate (non-GUI) class encapsulating the player selection process, then instantiate an object of that class inside your GUI class and call the various methods as needed.

69. Write a GUI-based program that analyzes a simplified pick of the NBA (National Basketball Association) draft. You will retrieve the data from a text file containing 10 lines. On each line will be the name of a player, the player's height, and the player's position on the court, each field separated by a space. Your program should read the file and display 10 radio buttons representing the 10 players. A text area will

display the information on the player corresponding to the radio button just selected. Every time the user clicks on a radio button, the information in the text area should be updated accordingly. Provide buttons for the following:

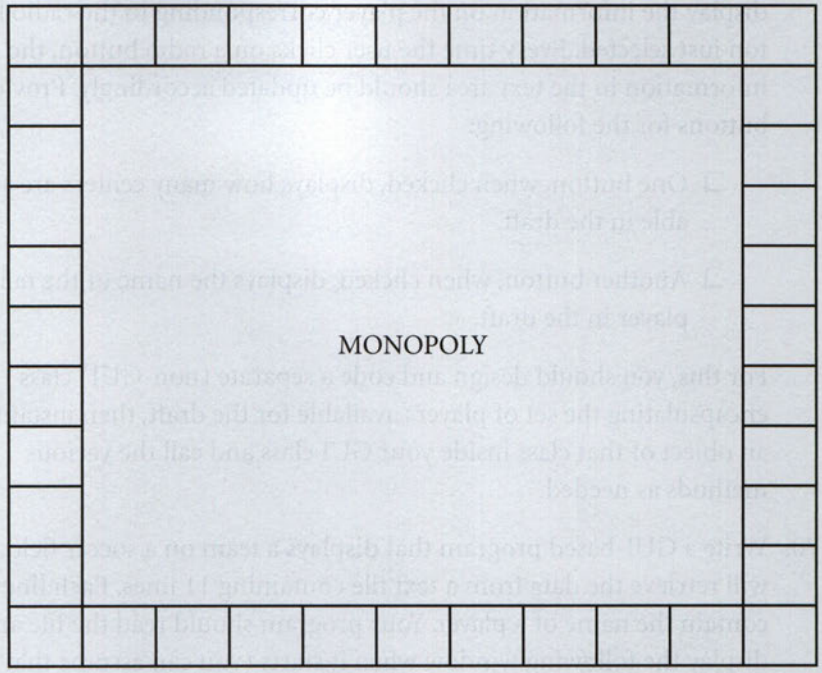
- ❑ One button, when clicked, displays how many centers are available in the draft.
- ❑ Another button, when clicked, displays the name of the tallest player in the draft.

For this, you should design and code a separate (non-GUI) class encapsulating the set of players available for the draft, then instantiate an object of that class inside your GUI class and call the various methods as needed.

70. Write a GUI-based program that displays a team on a soccer field. You will retrieve the data from a text file containing 11 lines. Each line will contain the name of a player. Your program should read the file and display the following window when it starts (you can assume that the players in the file are not in any particular order). Each cell is a button; when the user clicks on a button, the button replaces its text with the name of the player.

Left wing (11)	Striker (9)		Right wing (7)
Left midfielder (6)	Midfielder (10)		Right midfielder (8)
Left defender (3)	Stopper (4)	Sweeper (5)	Right defender (2)
	Goalie (1)		

71. Write a GUI-based program that displays a Monopoly® game. Add labels for the four train stations. Add buttons for all the Chance cells and set the text of these to a question mark. When the user clicks on one of the buttons, set its text to a message of your choice, chosen randomly from four messages.



72. Write a GUI-based, simple drawing program. This program should have two buttons: one allowing the user to draw a rectangle, the other allowing the user to draw an oval. The user draws either a rectangle or an oval by pressing the mouse, dragging it, and releasing it. The top-left (x, y) coordinate of the rectangle (or enclosing rectangle for the oval) is where the user pressed the mouse button; the bottom-right point of the rectangle (or enclosing rectangle for the oval) drawn is where the user released the mouse button.You will need to organize the window in two areas: one for the buttons, one for drawing.

12.18.8 Technical Writing

73. You are writing a program that you expect to be used by many users, all with a different computer system. Would you use layout managers or would you hard code the position of components inside your GUI? Discuss.

12.18.9 Group Project (for a group of 1, 2, or 3 students)

74. Design and code a program that simulates an auction. You should consider the following:

    A file contains a list of items to be auctioned. You can decide on the format of this file and its contents. For example, the file could look like this:

    Oldsmobile, oldsmobile.gif, 100
    World Cup soccer ticket, soccerTickets.gif, 50
    Trip for 2 to Rome, trip.gif, 100

    In the preceding file sample, each line represents an item as follows: the first field is the item's description, the second field is the name of a file containing an image of the item, and the third field is the minimum bid. You can assume that each item's description is unique.

    Items are offered via an online-like auction. (You do not need to include any network programming; your program is a single-computer program.) Users of the program can choose which item to bid on from a list or a combo box. Along with displaying the description of the item, your program should show a picture of the item and the current highest bid (at the beginning, the current highest bid is the minimum bid). Users bid on an item by selecting the item, typing their name (you can assume that all users have a different name), and entering a price for the item. Your program should remember the highest bidder and the highest bid for each item by writing the information to a file. Furthermore, each time a bid is made, the item's highest bid, displayed on the screen, should be updated if necessary.