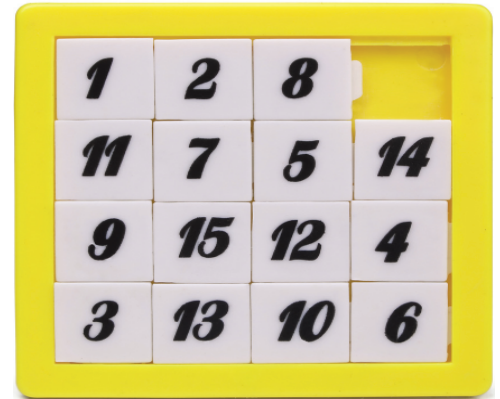


GRAPHICAL USER INTERFACES

11.1	<i>Overview</i>	480
11.2	<i>Enhancing Dialog Boxes</i>	482
11.3	<i>Creating a Graphical User Interface for an Application</i>	487
11.4	<i>Event Processing</i>	500
11.5	<i>Layout Managers</i>	522
11.6	<i>Applets</i>	531
11.7	<i>Chapter Summary</i>	544



In this chapter

In this chapter, we will learn how to create more user-friendly and informative dialog boxes and how to build and incorporate graphical user interfaces (GUIs) into our programs. The use of these point-and-click interfaces makes interacting with a program more user friendly. Java provides two packages, the Abstract Window Toolkit (AWT) and Swing, to facilitate the development of dialog boxes and GUIs.

Principles for designing a GUI interface will be explained and illustrated as will the use of a GUI-builder worker class to create a window, add the GUI components to the window, and perform the processing associated with the user's interaction with its components. These components include panels, buttons, text fields, labels, and tool tips. Various layout managers, used to organize the components, will be compared.

Methods called event handlers will be discussed. These methods are invoked by the Java Runtime Environment (JRE™) when an event, such as a mouse click or a mouse drag, is performed on the GUI. We will learn how to write these event handler methods and how to register the methods with the Runtime Environment. The use of paint event handler methods to draw two-dimensional graphical objects on a GUI component will also be discussed.

Finally, these graphical concepts will be applied to Web-based programs called *applets*. We will discuss how to write applets, the basics of HTML code used to download and launch an applet in a Web browser, and some security issues associated with applets.

After successfully completing this chapter you should:

- Be able to create and use dialog boxes that are more informative and user friendly

- Know how to design and implement GUIs that contain panels, text fields, buttons, labels, and tool tips using Java's AWT and Swing packages
- Be familiar with top-level containers, containers, and atomic GUI components and their role in graphical interfaces
- Understand the advantages of using a GUI-builder worker class to construct a graphical interface and how to implement these classes
- Know the three (or sometimes four) step process for adding components to a container
- Understand how to write event handler methods to process mouse, keyboard, timer, and paint events that occur on a GUI
- Know how to register event handler methods with the Java Runtime Environment so they are invoked when the specific events occur
- Be able to distinguish between applications and applets and be able to implement an applet and launch it in a Web browser

Understand security issues associated with applets and Java's role in these issues

11.1 OVERVIEW

A graphical user interface is a means of interacting with a program. Most often referred to using the acronym GUI (pronounced "goo-ee"), its design goal is to make the use of a program self-evident. GUIs are a much more user friendly than the original command-based interaction scheme in which a program would issue a text-based prompt that generically amounted to "what would you like to do?" and the user responded by typing a command such as "tax program."

Developed during the late 1970s, graphical interfaces were initially used to communicate with the operating system, but their power and ease of use was quickly adopted into all of the applications run on a system. Wherever possible, text-based prompts are replaced with icons, and keyboard input is replaced with mouse clicks, audio commands, and touch screen/pad input.

Just as graphical road signs succinctly communicate information to motorists, GUI objects, called *components*, permit us to quickly navigate our way through a program. The features of each of these components, which include clickable buttons, check boxes, radio buttons, scroll bars, sliders, and menu bars, to name a few, facilitate particular I/O functions that are common to most programs. Figure 11.1 shows some of the more commonly used components.

While the use of GUIs has reduced the time and effort required to interact with a program, incorporating a GUI into a program can significantly increase the time and effort required to develop it. In reaction to this, many integrated development environments provide a GUI-builder feature that allows the programmer to rapidly develop the interface. It is built by selecting commonly used GUI components from a graphical display, dragging them to a position on a panel that will become the user interface, and then setting features associated with them such as their color, size, text type, and visibility. As the programmer builds the interface, the IDE adds the code to the program that creates and displays the components and adds empty methods to the program that will execute when the user interacts with the components. The programmer then adds the code to perform the processing associated with the components to these methods.

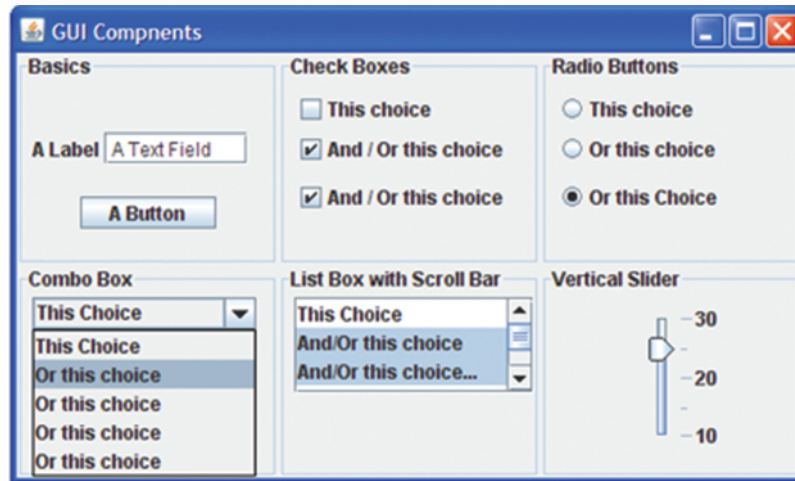


Figure 11.1
Commonly used GUI components.

The AWT and Swing Packages

The Java code generated by these GUI builders relies heavily on the classes that are part of the API AWT and Swing packages. Both of these packages are part of the Java Foundation Classes (JFC), a set of classes that support the development of graphical interfaces. The AWT package was part of the API before the Swing package was added to it. The Swing package both duplicated and extended the range of the types of GUI components available in the AWT package to include file and color-chooser dialog boxes and provided additional features such as tool tips and the ability to interact with the GUI in a drag-and-drop mode. All Swing components are designed to be 100% cross-platform compatible.

Java applications that use components in the Swing package are less dependent on the graphical features of the platform's operating system on which the application is run. For example, minor differences in the components' appearance (or look) and the change in their appearance when the user interacts with them (their feel) that are platform dependent can be eliminated, or the look and feel of the components can be made to emulate the platform on which the application is running.

In addition, Swing component classes are written in Java and do not contain any platform-specific code. For that reason, they are referred to as *lightweight* components, to distinguish them from the subset of *heavyweight* components that are “weighed down” by (i.e., contain) platform-specific code, and always emulate the look and feel of the platform on which they are running. Most applications use GUI components that are instances of classes in the API Swing (`javax.swing`) package.

11.2 ENHANCING DIALOG BOXES

Input and message dialog boxes are graphical user interfaces used to perform I/O with the program user. In addition to the versions of the `showInputDialog` and `showMessageDialog` methods discussed in Chapter 2, the `JOptionPane` class provides several overloaded versions of these methods and other methods that can be used to provide more informative and user-friendly dialog boxes. The default icon and title displayed in the dialog boxes can be changed, the dialog boxes can be displayed in the middle of a specified window such as the game board's window, and a default input or a set of input selections can be displayed in an input dialog box.

Table 11.1 presents a summary of the overloaded versions of the `showInputDialog` and `showMessageDialog` methods, with their signatures given in its left column. The check marks

Table 11.1

Options for Displaying Input and Message Dialog Boxes

Method	Feature(s) Incorporated into the Method				
	Default Input	Specify Window	Title	Icon	Input Choices
Input Dialog Boxes					
<code>showInputDialog(Object prompt)</code>					
<code>showInputDialog(Object prompt, Object defaultInput)</code>	√				
<code>showInputDialog(Component window, Object prompt)</code>		√			
<code>showInputDialog(Component window, Object prompt, Object defaultInput)</code>	√	√			
<code>showInputDialog(Component window, Object prompt, String title, int messageIcon)</code>		√	√	√	
<code>showInputDialog(Component window, Object prompt, String title, int messageIcon, Icon icon, Object[] selectionValues, Object initialValue)</code>	√	√	√	√	√
Message Dialog Boxes					
<code>showMessageDialog(Component window, Object message)</code>		√			
<code>showMessageDialog(Component window, Object message, String title, int messageIcon)</code>		√	√	√	

in the columns to the right identify the features of each version of the methods. The top section of the table presents the input dialog methods, and the message dialog methods are presented in the bottom section. The signatures of the methods that have been used up to this point in the textbook are shown at the beginning of the two sections of the table. All of the input dialog box methods return a reference to a `String` object except for the last one shown in the top portion of the table, which returns an `Object` reference.

The parameter `window` in the signatures of the message-box methods and last four input-box methods could be passed a reference to a GUI component such as a window. When it is, the dialog box is displayed in the center of the component. If the parameter is passed a `null` value, the dialog box is displayed in the center of the program window that invoked the method. To display it in the center of the game board window, the method would be passed the `GameBoard` object `gb` as shown on line 12 of the application `CenterMsgBox`, shown in Figure 11.2. The output it produces is shown in Figure 11.3.

Normally, the `prompt` and `defaultInput` parameter used in the input dialog method signatures shown in the Table 11.1 are passed a `String` object. The `defaultInput` is displayed in the text area of the input box when it appears on the monitor. It can be changed (overtyped) by the program user. The argument passed to the parameter `title` (used in fifth and last rows of the table) is displayed in the title bar at the top of the dialog box. The parameter `message` in the message dialog method signatures is normally passed a `String` object or any object that contains a `toString` method. The parameter `icon` in the sixth row of the table is used to pass a programmer-defined instance of the `Icon` class to the method.

```

1  import edu.sjcny.gpv1.*;
2  import javax.swing.JOptionPane;
3
4  public class CenteredMsgBox extends DrawableAdapter
5  {
6      static CenteredMsgBox ge = new CenteredMsgBox();
7      static GameBoard gb = new GameBoard(ge, "My Game");
8
9      public static void main(String args[])
10     {
11         showGameBoard(gb);
12         JOptionPane.showMessageDialog(gb, "A Messages Box Centered " +
13                                         "in the Game Board Window");
14         showGameBoard(gb);
15     }
16 }

```

Figure 11.2

The application `CenterMsgBox`.

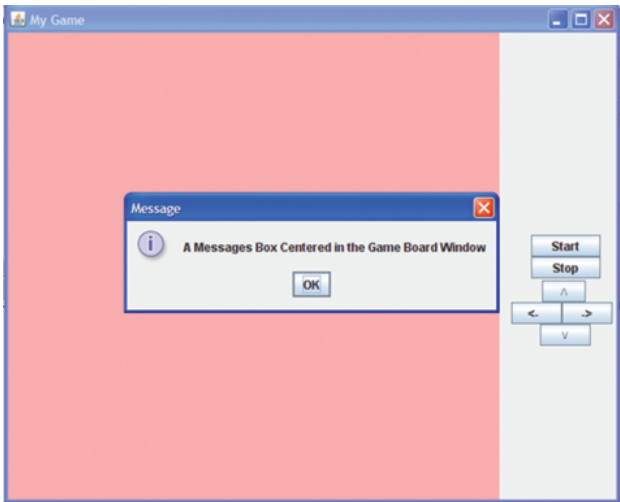






Figure 11.3
The output produced by the application `CenterMsgBox`.

The methods in Table 11.1 whose signatures contain the parameter `messageIcon` can be passed any of five static constants defined in the class `JOptionPane`. This parameter is used to specify which one of five predefined icons will be displayed on the left side of a dialog box. Table 11.2 gives the names of the constants, their integer value, and the icons that are associated with each of them. The methods in Table 11.1 whose signatures do not contain the parameter `messageIcon` always display the default icons identified parenthetically in the rightmost column of the table. An integer literal between -1 and 3 inclusive (one of the five constants' values) can alternately be passed to this parameter.

Table 11.2
The `JOptionPane` Class's Predefined Dialog Box Icon Constants and Icons

Constant Name	Value	Icon	Common Icon Use
PLAIN_MESSAGE	-1	none	Other defined icons are inappropriate: no icon is displayed
ERROR_MESSAGE	0		An error or problem has occurred
INFORMATION_MESSAGE	1		For your information (message dialog box default)
WARNING_MESSAGE	2		Consider possible ramifications
QUESTION_MESSAGE	3		A reply to the prompt is requested (input dialog box default)

The last method shown in the input portion of the Table 11.3 implements all the features presented in that table. In this version of the method the default input is actually designated to be one of a valid set of inputs contained in an array passed to the method's `selectionValues` parameter. The designation of the default value is performed by passing one of the elements of the array to the parameter `initialSelectionValue`. The elements of the array can be `String` objects, instances of a class that contains a `toString` method, or several other options that will be discussed later in this chapter. If a `null` value is passed to the parameter `selectionValues`, the user can overstrike the displayed default value; otherwise, the user can only select one of the objects in the array which are displayed in a drop-down box.

The application `EnhancedDialogBoxes` presented in Figure 11.4 demonstrates the use of all of the features implemented by the overloaded dialog box methods presented in Table 11.1, except centering the dialog box in a GUI component (which was demonstrated in the application presented in Figure 11.2). The dialog box outputs produced by the program are shown in Figure 11.5.

The word *ERROR* passed to the second parameter of the method invoked on line 12 of Figure 11.4 appears in the title area of the message box it outputs (Figure 11.5a). This message box also contains the non-default Error icon, whose number (0) is passed to the method's third parameter using the static constant `JOptionPane.ERROR_MESSAGE`.

Line 15 displays an input dialog box containing the default input, *Sophomore*, as shown in Figure 11.5b. The default value is passed to its second parameter on line 16.

The method invoked on line 19 displays the input dialog box that is shown in Figure 11.5c. The box contains the title *Standing* passed to the method's third parameter on line 21 and the Question icon because the numeric literal 3 is passed to its fourth parameter. The default input *Junior* is also displayed in the text area of the input box because the third element of the array, defined on lines 7 and 8, is passed to the method's last parameter on line 24. The `null` value passed to the method on line 22 indicates that a user programmer-defined icon is not passed to the method.

Figure 11.5d shows the input box displayed to its left after the user clicks the box's down arrow to display the valid input choices passed to the method on line 23. The coercion on line 19 is necessary because the method invoked on that line returns an `Object` reference variable that contains the address of the user-selected object contained in the array passed to it on line 23.

```

1  import javax.swing.JOptionPane;
2
3  public class EnhancedDialogBoxes
4  {
5      public static void main(String[] args)
6      {
7          String[] inputOptions = {"Freshman", "Sophomore",
8                                  "Junior", "Senior"};
9          String s1, s2;
10
11         // Titled message box with an error icon

```



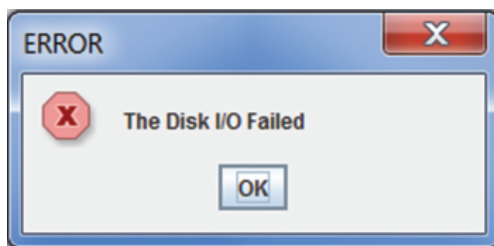
```

12     JOptionPane.showMessageDialog(null, "The Disk I/O Failed", "ERROR",
13                                   JOptionPane.ERROR_MESSAGE);
14     // Input box with a default input
15     s1 = JOptionPane.showInputDialog("Enter your Class Standing",
16                                     "Sophomore");
17
18     // A Non-default icon titled Input box, a valid set of inputs
19     s2 = (String) JOptionPane.showInputDialog(null, "Select your " +
20                                               "class standing",
21                                               "Standing", 3,
22                                               null,
23                                               inputOptions,
24                                               inputOptions[2]);
25
26     System.out.println(s1 + " " + s2);
27 }
28 }

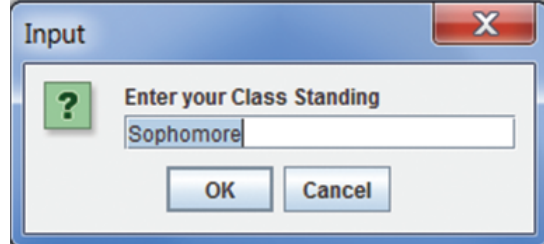
```

Figure 11.4

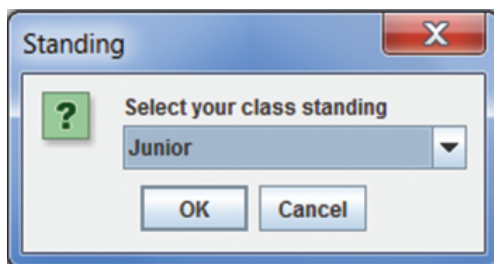
The application **EnhancedDialogBoxes**.



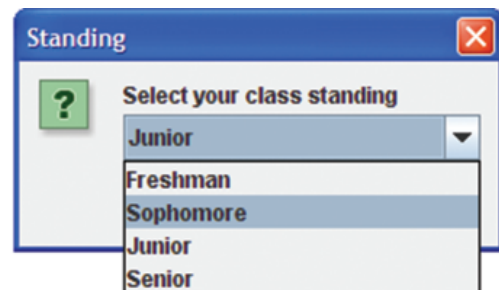
(a)



(b)



(c)



(d)

Figure 11.5

The output produced by the application **EnhancedDialogBoxes**.

11.3 CREATING A GRAPHICAL USER INTERFACE FOR AN APPLICATION

Graphical user interfaces are created by declaring an instance of a top-level container class and then adding GUI components to it. Top-level container classes in the Swing package include `JWindow`, `JFrame`, `JApplet`, and `JDialog`. The class `JFrame` extends the capabilities of its parent class `JWindow` by adding a title bar that contains the window management icons (minimize, resize, and close), an optional title, and the ability to drag the window. Figure 11.6 shows the inheritance chain of the Swing top-level containers.

Non-Web based GUI applications use `JFrame` as their top-level GUI container because it has the look and feel of a program window. Web based applications, referred to as applets, use `JApplet` as their top-level container. The class `JDialog` is used as the top-level container for GUI components that are to be part of a sub-window. Input and message dialog boxes are instances of this class. All top-level containers have a *content pane*, which is the area of the window that will contain the GUI components particular to an application.

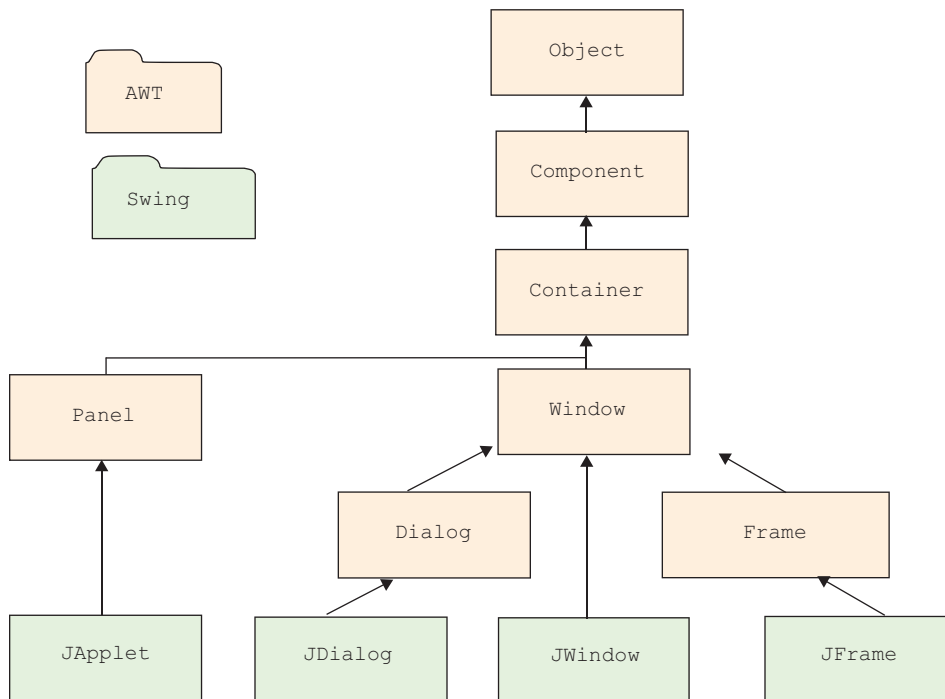


Figure 11.6
The inheritance chain of the top-level swing containers.

11.3.1 The Content Pane

Figure 11.7 shows two `JFrame` instances, one with a menu bar (left window) and one without a menu bar (right window). The content panes are the yellow portions of the windows. When a `JFrame` window is created, its outer width and height (in pixels) is specified. Within these outer dimensions, there is a rectangular border that is, by default, 4 pixels wide. A 26-pixel-high title bar is positioned directly below the top portion of the border, and a 23-pixel-high menu bar can be added to the window directly below the title bar. The remaining area of the window determines the width and height of the content pane, which are the window's dimensions minus the surrounding borders, title bar, and menu bar.

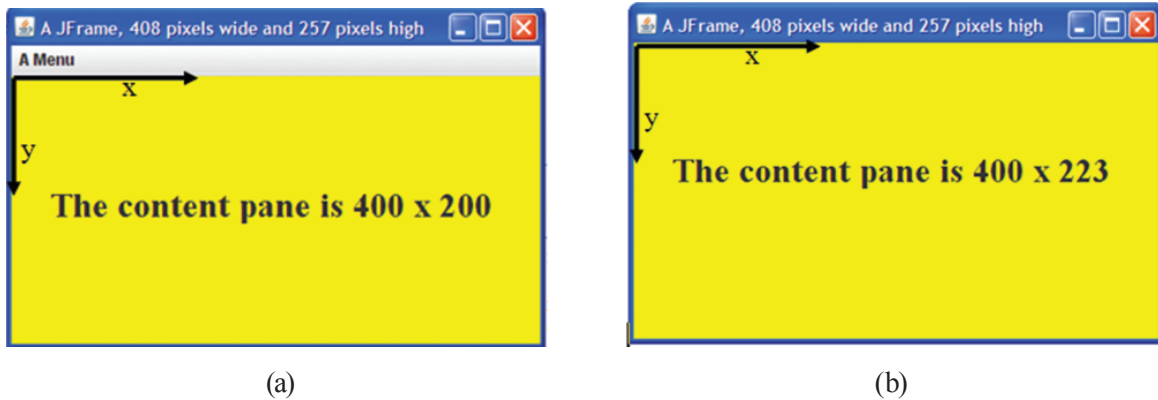


Figure 11.7

Two `JFrame` windows 408 pixels wide and 257 pixels high.

For example, the windows in Figure 11.7 are both 408 pixels wide and 257 pixels high. The left window's content pane is 400 pixels wide (408 minus the left and right border widths) and 200 pixels high (257 minus the top border, title bar, menu bar, and bottom border heights). The right window's content pane is 223 pixels high because the height of the menu bar (23 pixels) in the window to its left is now part of the content pane.

The origin of the coordinate system used to position components added to the content pane is located at the upper-left corner of the content pane. The positive *x* direction is to the right, and the positive *y* direction is downward. The text shown on the two content panes in Figure 11.7 was added at the same (*x*, *y*) locations. Because the origin of the content pane in the window without the menu bar is higher in the window, the text appears closer to the top of the window.

11.3.2 Creating and Displaying a Program Window

The one-parameter constructor of the `JFrame` class is used to create a program window. The string parameter passed to it is displayed on the left side of the window's title bar. Alternately, the default (no-parameter) constructor can be used to create a window with no title. Once created, the size of the window must be specified, and then the window must be made visible. The following code fragment creates the 708 x 434 pixel window shown in Figure 11.8, which is displayed with its upper left corner at the default location (0, 0) on the monitor.

```

JFrame appWindow = new JFrame("A GUI Window");
appWindow.setSize(708, 434); //content pane is 700 x 400
appWindow.setVisible(true);

```

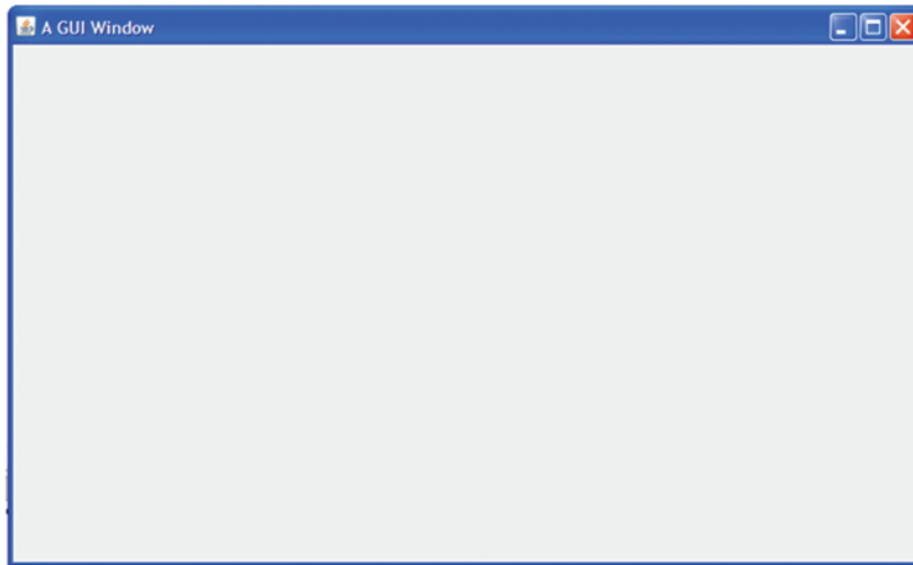


Figure 11.8
A program window.

Normally, three additional methods are invoked when a window is created. One method is used to reposition the upper left corner of the window from its default (0, 0) monitor location, and the second method is used to change the color of the content pane (which defaults to light gray). The third method is used to direct the Runtime Environment to terminate the program when the user closes the window. The default is to continue the program's execution.

The signatures of these three methods, the `JFrame` constructors, and the methods `setSize` and `setVisible`, and one additional method are shown in Table 11.3. All of the methods in the table are non-static methods, and all but the last one are members of the `JFrame` class. The last method is a member of the class `Component`.

Table 11.3
Methods Used to Create and Display a Window

Method Signature	Function
<code>JFrame () ;</code>	Construct a window with no title
<code>JFrame (String windowTitle)</code>	Construct a window with the title <code>windowTitle</code>
<code>setSize (int width, int height)</code>	Sets the width of the window to <code>width</code> pixels and the height of the window to <code>height</code> pixels
<code>setVisible (boolean isVisible)</code>	Displays the window on the monitor when <code>isVisible</code> is true

(Contd.)

Method Signature	Function
<code>setLocation(int x, int y)</code>	Positions the upper left corner of the window to pixel location (x, y) of the monitor
<code>setDefaultCloseOperation(int action)</code>	Specifies the action to be taken when the window is closed action: 3 terminates the program
<code>getContentPane()</code>	Returns a <code>Container</code> reference to the window's content pane
<code>setBackground(Color paneColor);</code>	Sets the color of the window's content pane to <code>paneColor</code> , a <code>Component</code> class method

The application `GUIWindow` shown in Figure 11.9 creates and displays the window shown in Figure 11.10. Line 10 positions the upper left corner of the window 100 pixels below and to the right of the upper left corner of the monitor. Line 11 sets the color of the content pane to pink by invoking the `setBackground` method on the content-pane object whose address is returned from the `getContentPane` method. The `JFrame` class inherits this method from the `Component` class.

The argument passed to the `setDefaultCloseOperation` invoked on line 12 is a static constant defined in the `JFrame` class used to specify the action to be taken when the window is closed. When used in this context, the application is terminated when its window, referenced by `appWindow`, is closed. Although the value of the constant (three) could have been coded as a numeric literal, the use of the constant (`EXIT _ ON _ CLOSE`) makes the line more readable and is considered good programming practice.

If line 12 were eliminated from the program, the Java Runtime Environment would still consider the program active, even though the window has been closed and the main method has ended. One result of this would be that the IDE used to execute the program would still consider the program to be running. We will learn more about this in Chapter 14 when we study the concept of *threads*.

```

1  import javax.swing.*;
2  import java.awt.Color;
3
4  public class GUIWindow
5  {
6      public static void main(String[] args)
7      {
8          JFrame appWindow = new JFrame("A GUI Window");
9          appWindow.setSize(708, 434);
10         appWindow.setLocation(100, 100);
11         appWindow.getContentPane().setBackground(Color.PINK);

```

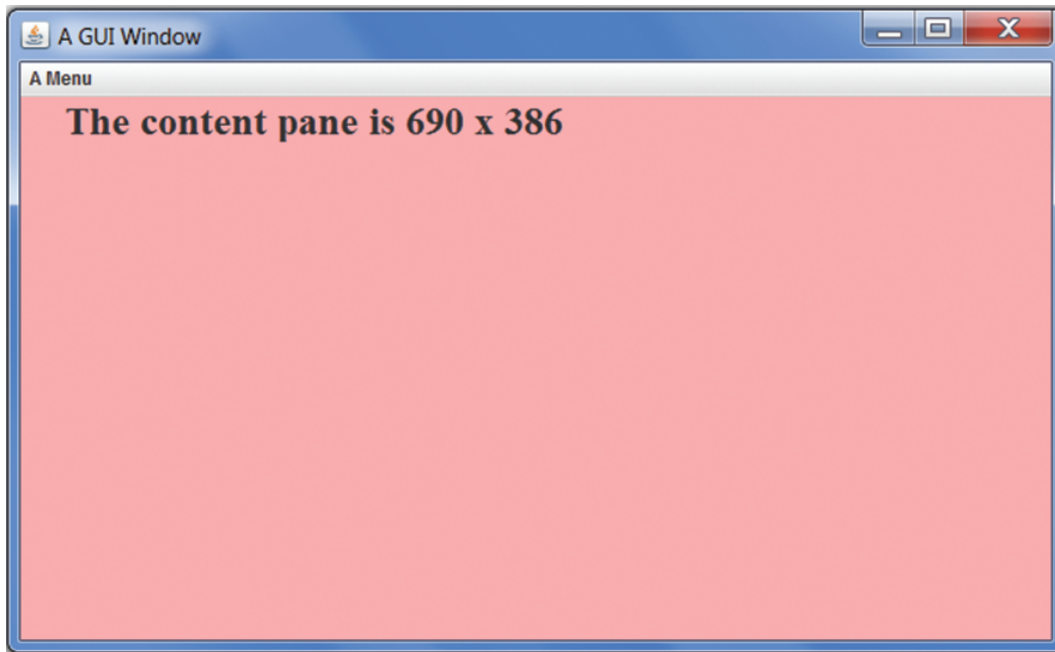
```

12     appWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     appWindow.setVisible(true);
14 }
15 }

```

Figure 11.9

The application `GUIWindow`.

**Figure 11.10**

The window produced by the application `GUIWindow`.

GUI-Builder Worker Classes

Consistent with the concept of divide-and-conquer, most often the building of the user interface is not performed in the main method, but rather, it is relegated to one or more worker classes. This is particularly useful when the application will contain one or more windows because each window can be built by a separate worker class. For this reason, it is the approach taken by IDE's that contain drag-and-drop GUI builders.

Line 5 of the application `GUIWindowBuilder`, shown in Figure 11.11, creates the same window as the application shown in Figure 11.9, but this time, the window is an instance of the worker class `WindowBuilder` shown in Figure 11.12. The worker class extends the class `JFrame`, but it does not add any data members or methods. It simply contains a constructor: lines 6–15 of Figure 11.12.

Line 8 of the worker class invokes `JFrame`'s one-parameter constructor to construct the window, passing it the string to be displayed in the window's title bar, then lines 10–15 set the window's size, location, color, close action, and visibility. This is the identical code used on lines 9–13 of the

application shown in Figure 11.9, except that the methods are invoked on the window created on line 8 of the constructor. The invocation on lines 10–14 of Figure 11.12 could have been preceded by the keyword **this** followed by a dot to more clearly indicate that they were being invoked on this object that was created on line 8, but the coding shown in the figure is more commonly used.

```

1  public class GUIWindowBuilder
2  {
3      public static void main(String[] args)
4      {
5          WindowBuilder appWindow = new WindowBuilder("A GUI Window");
6      }
7  }

```

Figure 11.11

The application **GUIWindowBuilder**.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class WindowBuilder extends JFrame
5  {
6      public WindowBuilder(String title)
7      {
8          super(title);
9
10         setSize(708, 434);
11         setLocation(100, 100);
12         getContentPane().setBackground(Color.PINK);
13         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14         setVisible(true);
15     }
16 }

```

Figure 11.12

The class **WindowBuilder**.

Depending on the application, the GUI-builder worker class may include one or more overloaded constructors to allow the client to specify not only the window's title, but also its size, location, color, close action and visibility.

The coding style of relegating the task of creating a GUI window to a separate worker class that extends `JFrame` will be used in the remainder of this chapter.

11.3.3 Adding GUI Components to a Window

The Swing package contains a rich assortment of GUI components, some of which are shown in Figure 11.1, that can be added to a window. While there is some overlap in the roles that they play in the I/O process, each component has been designed to facilitate a particular I/O function.

For example, the functionality of radio buttons makes them the best components to use to acquire one choice from a small set of mutually exclusive choices.

The most common components used in GUI interfaces are buttons, text fields, labels, check boxes, radio buttons, and combo boxes. Table 11.4 lists the constructor methods used to create these components grouped by their intended functionality and gives a brief description of the I/O function they were designed to facilitate.

Table 11.4
Commonly Used Java Swing GUI Components

Component Constructors	Targeted Use
Input Components	
JCheckBox(String text) JCheckBox(String text, boolean selected)	Select one or more inputs from a group of suggested inputs by clicking a box
JRadioButton(String text) JRadioButton(String text, boolean selected)	Select one input from a group of suggested inputs by clicking a button
JComboBox(E[] items)	Select one input from a group of suggested inputs by clicking an item
Input or Output Component	
JTextField(String text)	Keyboard input, String output
Annotation or Output Component	
JLabel(String text)	Annotate a window including placing prompts at text boxes; String output to the window
Initiate Processing Component	
JButton(String text)	Execute instructions associated with the click of the button
Collect Other Related Components and 2D Graphics Components	
JPanel() JPanel(LayoutManager layout)	Group other related components and draw 2D shapes

The names of all of the Swing component class names begin with a capital J. They are all direct or indirect descendants of the class `JComponent`, whose inheritance chain is shown in Figure 11.13. The only exceptions to this are the top-level component classes, previously discussed in this chapter, whose inheritance chains are shown in Figure 11.6.

Just as the top-level components can contain other components, some non-top-level components can also contain other components. A `JPanel` is an example of this type of component, while buttons and text fields are designed to be **atomic** components.

Definition

Atomic components are GUI components that cannot contain other components.

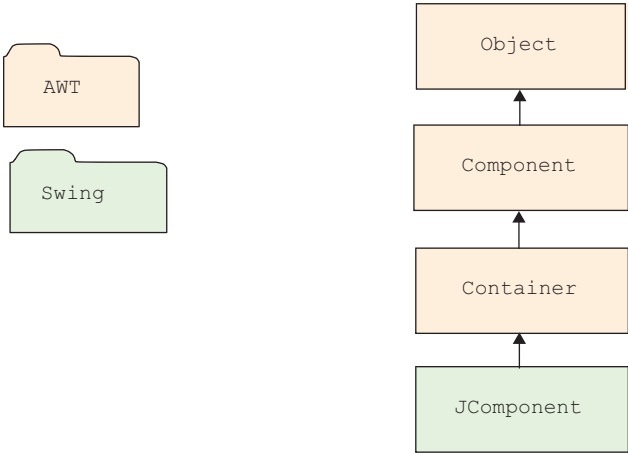


Figure 11.13
The `JComponent` class's inheritance chain.

Designing the Interface

Before adding a GUI component to a window, it is very useful to make a quick sketch of the interface that includes all of the components to be added and their position in the window. The choice of which components to add is based on the I/O requirements of the program and the component's targeted use listed in Table 11.4. As noted in the table, the `JTextField` component can be used for both input and output.

If the program involves a series of inputs that should be entered in a particular order, adding input components to the window from left to right and top to bottom in order of entry enhances the friendliness of the interface. A sketch of an adding machine GUI is shown in Figure 11.14. Its level of detail is typical of that contained in a design sketch.

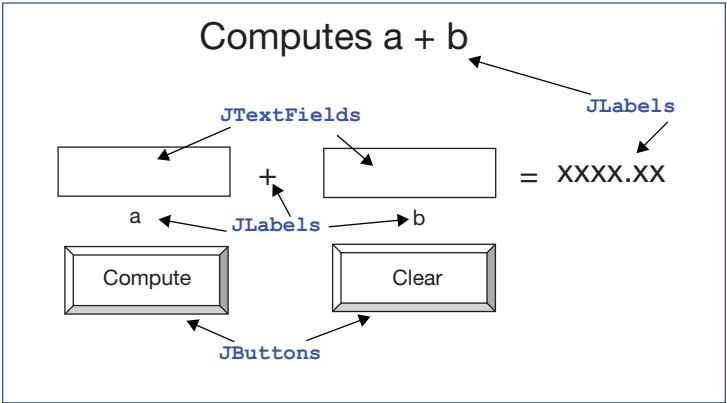


Figure 11.14
The GUI design for an adding machine.

Adding the Components

When an IDE is being used that has a GUI-builder feature, the program's window appears in the GUI builder. The components are added to the window by simply selecting them from a list of components, and then dragging them to their position in the window. Their size is typically adjusted by dragging resizing handles, and properties such as font color, font size, and initial visibility are specified via a menu of properties appropriate to each component.

While the particulars of GUI builders vary from one IDE to another, they all provide a rapid way of selecting, positioning, locating, and sizing an application's GUI components. Perhaps more importantly, while this process is being performed, the GUI builder adds the code to the application that creates, locates, and displays the components. It also adds templates for the code that responds to the runtime interaction with the components. These features greatly reduce the time to develop an application's GUI, which is the reason an IDE with a GUI builder should always be used when developing an application that has a graphical user interface.

Java also provides several *layout managers* that can facilitate the tedious process of locating components in a window when an IDE with a GUI builder is not available, but their use is limited to the development of simple IDEs. We will discuss layout managers in Section 11.5.

In the remainder of this section, we will become familiar with the methods used to construct, locate, size, and add components to a window and some of the methods used to adjust their properties. Invocations to these methods are added to our application when the interface is created with an IDE GUI builder. Even when the invocations are generated by the IDE, knowledge of these methods and their use is essential to altering the generated code and to completing the code templates added to the application.

A three or (for radio buttons) four step coding process is used to add a component to a window or to other non-atomic containers such as a `JPanel` object:

1. Create the component object
2. Specify the component's properties such as size, location, font style, tool tip, and visibility
3. Add mutually exclusive radio buttons to a common button group
4. Add the component to the window or non-atomic container

The more commonly used constructors used in step 1 of this process to create the components described in Table 11.4 are given in that table. The string passed to these constructors and the array passed to the combo box's constructor become the annotation that will appear on, within, or next to the component. For example, the following statement creates a button with the text *Click Me* displayed on it, and a text field with the text *Hamburger* displayed in it. The component's size must be wide enough to accommodate the width of the text, or the text will not be displayed:

```
JButton aButton = new JButton("Click Me");
JTextField entree = new JTextField ("Hamburger");
```

Table 11.5 gives the methods used in steps 2 and 4 of the process to specify a component's properties and add the component to a window or some other non-atomic container. The

techniques and methods used in step 3 to group mutually exclusive radio buttons will be discussed in Chapter 12, as will the development of interfaces that contain check boxes and radio buttons. The `Component` and `Container` classes contain `get` methods for each of their `set` methods presented in the Table 11.5. All `JComponents` are visible by default.

Table 11.5

Methods Used to Specify a Component's Properties and Add it to a Container

Method Signature	Description
JComponent and Component Class Methods Invoked on Components	
<code>setToolTipText(String tip)</code>	Adds the tool tip <code>tip</code> to the component, displayed when the mouse pointer hovers over it
<code>setBounds(int x, int y, int width, int height)</code>	Sets the component's location to <code>(x, y)</code> and its width and height to <code>width</code> and <code>height</code>
<code>setLocation(int x, int y)</code>	Sets the component's location to <code>(x, y)</code>
<code>setSize(int width, int height)</code>	Sets the component's width and height to <code>width</code> and <code>height</code>
<code>setText(String newText)</code>	Changes the text displayed on the component to <code>newText</code>
<code>setVisible(boolean visible)</code>	The component is visible when <code>visible</code> is passed the value <code>true</code> , invisible when passed <code>false</code>
<code>setFont(Font fontStyle)</code>	Sets the font style of the container or component that invoked the method to <code>fontStyle</code>
Container Class Methods	
<code>setLayout(LayoutManager layout);</code>	Sets the container's layout to <code>layout</code> , to specify location/size of components: <code>layout = null</code>
<code>add(Component theComponent)</code>	Adds <code>theComponent</code> to the container or component that invoked the method

When coded inside the constructor of a GUI-builder worker class, such as the one shown in Figure 11.12, the following code fragment adds a 300-pixel-wide by 30-pixel-high `JLabel` to the GUI that contains the text *Computes a + b*. The upper left corner of the text is located on the window's content pane at (120, 0), and the font type, style, and size of the displayed text is *Sherif*, bold, 24 point.

```
setLayout(null);
JLabel description = new JLabel("Computes a + b");
description.setBounds(120, 0, 300, 30);
description.setFont(new Font("Sherif", Font.BOLD, 24));
add(description);
```

The `null` value passed to the invocation of the `setLayout` method permits the use of the `setBounds` method to size and locate a component. Once invoked, all components subsequently added to the window can be positioned and sized using the `setBounds` or `setLocation` and `setSize` methods. The use of a non-atomic component's layout manager to size and position components added to it will be discussed in Section 11.5.

The following code fragment, when coded inside the constructor of a GUI-builder worker class, adds a 90-pixel-wide by 25-pixel-high `JButton` to the GUI. The button contains the text *Clear* and contains a tool tip. The upper left corner of the button is located at (235, 110). The fragment assumes that the `setLayout` method has been invoked and passed a `null` value before the code it executes.

```
JButton clear = new JButton("Clear");
clear.setLocation(235, 110);
clear.setSize(90, 25);
clear.setToolTipText("Clears a, b and the sum");
add(clear);
```

The `setLocation` and `setSize` methods were used to locate and size the button simply to demonstrate the use of these methods. When both the location and the size of a component are being set, one invocation of the `setBounds` method is the preferred programming style.

The worker class `AddingMachineGUI`, shown in Figure 11.15, builds the graphical user interface, shown in Figure 11.16, whose design is shown in Figure 11.14. The application `AddingMachine`, shown in Figure 11.17, creates an instance of this interface on line 8, sets its default close operation on line 9, and makes the interface visible on line 10. The last two tasks could have been performed by the GUI-builder class as they were in the class presented in Figure 11.12. The removal of these window initialization tasks from the GUI-builder class is often employed when more than one GUI is used in a program.

The code of the class `AddingMachineGUI` (Figure 11.15) follows the four-step process (without step 3 because the GUI does not contain radio buttons) discussed in this section to add the GUI components to the window it creates. The coding style used performs a step of the process on all of the components before moving on to the next step. Step 1 for all components is performed on lines 18–27, step 2 for all components is performed on lines 30–46, and step 4 is performed on lines 49–58.

Within each step, the components are processed in the order in which they appear in the GUI design, starting at the top and moving from left to right. Consistent with this approach, line 18 constructs the descriptive label that appears at the top of the GUI design, line 19 constructs the text field that is below it and to its left, line 20 constructs the label that contains the plus sign to the right of this text box, etc.

This coding style makes the program more readable and facilitates the coding process by reducing the number of iterations used to properly locate the components in the window. In addition, it is consistent with the requirements of one of the layout managers discussed later in this chapter and is the coding style often used in the code generated by an IDE's drag-and-drop GUI builder.

The class extends `JFrame` on line 4 of Figure 11.15 because it is going to create a window with a title bar containing a title and window management icons. All of the code to create the window and add the components to it is written inside the constructor. Line 12 constructs the window by invoking `JFrame`'s one-parameter constructor and passing it the parameter `title` declared on line 10. As the comments at the end of the lines 12 and 13 state, all subsequent invocations of methods on unnamed objects operate on the window created on line 12.

Lines 13 and 14 specify the size of the window and the location of the upper left corner of the window when it is displayed on a monitor. Line 15 permits the programmer to take control of the sizing and positioning of the GUI components away from the `JFrame`'s default layout manager. The remaining three sections of code create the GUI components, specify their properties, and add them to the `JFrame` window. The variables that reference the components are declared as class-level variables on lines 6–8 to allow methods that will be added to the class in the next section of this chapter to access the components.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class AddingMachineGUI extends JFrame
5  {
6      JLabel description, plus, equals, sum, a, b;
7      JTextField aValue, bValue;
8      JButton compute, clear;
9
10     public AddingMachineGUI(String title)
11     {
12         super(title);           //Creates the window. All subsequent invocations
13         setSize(500, 250);      //on an unnamed object operate on this window.
14         setLocation(200, 100);
15         setLayout(null);
16
17         //Step 1 create the components
18         description = new JLabel("Computes a + b");
19         aValue = new JTextField();
20         plus = new JLabel("+");
21         bValue = new JTextField();
22         equals = new JLabel("=");
23         sum = new JLabel("x,xxx.xx");
24         a = new JLabel("a");
25         b = new JLabel("b");
26         compute = new JButton("Compute");
27         clear = new JButton("Clear");
28
29         //Step 2 specify the component's properties
30         description.setBounds(120, 0, 300, 30);
31         description.setFont(new Font("Sherif", Font.BOLD, 24));
32         aValue.setBounds(60, 50, 100, 30);
33         plus.setBounds(190, 50, 20, 30);
34         plus.setFont(new Font("Sherif", Font.BOLD, 20));
35         bValue.setBounds(230, 50, 100, 30);
36         equals.setBounds(350, 50, 20, 30);
37         equals.setFont(new Font("Sherif", Font.BOLD, 20));
38         sum.setBounds(380, 50, 100, 30);
39         sum.setFont(new Font("Sherif", Font.BOLD, 20));
40         a.setBounds(105, 75, 20, 30);
41         a.setFont(new Font("Sherif", Font.BOLD, 20));

```

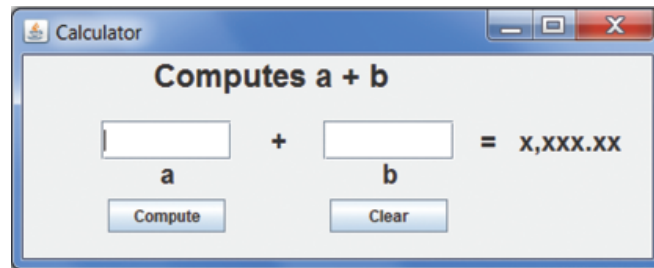
```

42     b.setBounds(275, 75, 20, 30);
43     b.setFont(new Font("Sherif", Font.BOLD, 20));
44     compute.setBounds(65, 110, 90, 25);
45     clear.setBounds(235, 110, 90, 25);
46     clear.setToolTipText("Clears a, b and the sum");
47
48     //Step 4 add the component to the container (Step 3 not relevant)
49     add(description);
50     add(aValue);
51     add(plus);
52     add(bValue);
53     add(equals);
54     add(sum);
55     add(a);
56     add(b);
57     add(compute);
58     add(clear);
59 }
60 }

```

Figure 11.15

The class **AddingMachineGUI**.

**Figure 11.16**

The graphical user interface created by the class **AddingMachineGUI**.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class AddingMachine
5  {
6      public static void main(String[] args)
7      {
8          AddingMachineGUI calculator = new AddingMachineGUI("Calculator");
9          calculator.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         calculator.setVisible(true);
11     }
12 }

```

Figure 11.17

The application **AddingMachine**.

11.4 EVENT PROCESSING

After the graphical interface is built, the next step in the GUI programming process is to identify the components in the interface that require application-dependent processing to be performed when the user interacts with them. For our adding machine, these would be the *Compute* and *Clear* buttons. When the compute button is clicked, the two text field entries are added, and the result is output. A click of the Clear button should clear the text boxes and the output sum.

In GUI jargon, when the user interacts with a component on the interface, we say that an *action* has been performed on the interface, or that an *event* has occurred. A click on one of our buttons is an example of an event. Other examples include the completion of an entry into a text box denoted by the striking of the Enter key, the movement of the mouse pointer over the window, or simply a click into a text box.

GUI events are detected by the Java Runtime Environment and some of them are dealt with, or processed, with no effort on the programmer's part. For example, when the user of a program's GUI interface clicks into one of its text fields, the insertion point cursor (*caret*) appears in the text field. This event is processed, or handled, by the code of API methods associated with the text field. When the click event occurs, the Runtime Environment executes a process to notify these methods that the event has occurred. After being notified of the event, they execute and display the insertion caret in the text field. Other sections of code associated with the text field subsequently handle keystroke events by displaying a typed character in the text field, and moving the caret to the right.

Events such as a click into a text box can be handled by the API methods because the action to be taken (display the caret at the position of the click) when the event occurs are part of the predefined look and feel of the GUI component. These application-independent GUI events are always processed by API methods. Other GUI events that occur on an application's interface, such as the clicking of the *Compute* button on our adding machine's interface, require the application programmer to process the event because the action to be taken is unique to this particular application.

More accurately, the application programmer partially processes these types of events because most often some processing that is part of the look and feel of the component also needs to be performed. For example, when the user clicks the *Compute* button of our adding machine, the API responds to the event by executing code that makes the button appear to have been depressed because this is part of a predefined look and feel of a button. Then, the application performs the processing particular to it: add two numbers together and display the result.

To initiate application-dependent processing when a GUI event occurs, the Java event-handling process permits the application programmer to add methods written as part of the application to the list of methods notified, or more accurately, executed, when the event occurs. We say that the programmer can add to the list of methods that are *listening* for the event to occur, and these methods are generically referred to as *event handlers*. Figure 11.18 illustrates this concept and the event processing execution path it produces.

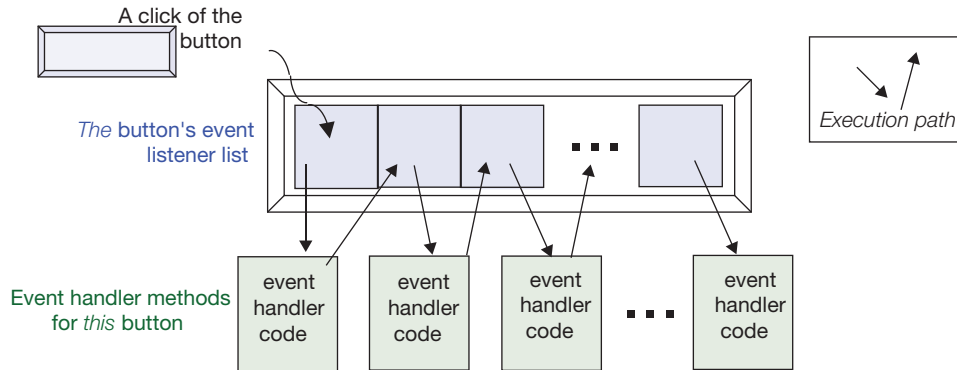


Figure 11.18
Java's event-processing process.

The process can be thought of as the execution of a switch statement, without any break statements, that executes all of its cases. Adding a method to a GUI component's event listener list would then be analogous to adding a case clause to the switch statement that invokes a method (the event handler method) to perform the processing for that case.

NOTE *Each component object added to the interface has its own event listener list.*

11.4.1 Implementing Event Handler Methods

Because the application-dependent event handler methods we code are invoked within API methods that are part of each GUI object added to the interface, the signatures of the methods have to be defined within the API implementation. The set of API interfaces whose names end in *Listener* defines the signatures of the application-dependent event handler methods invoked when a GUI event occurs.

Table 11.6 presents the names of some of the more commonly used listener interfaces and the names and parameter lists of the event handler methods they define. They are all void methods, and in most cases, the names of the methods imply the events that they handle. The most obvious exception to this is the signature of the `actionPerformed` method at the top of the table. The events it handles, as well as those handled by the other methods presented in the table, are summarized in the middle column. The check mark in the rightmost column of the table indicates that the interface on that row has an *adapter* class associated with it. Adapter classes will be discussed in Section 11.4.4.

Table 11.6

Several API Listener Interfaces and Commonly Used Event Handler Methods they Define

Event Handler Interfaces and Their Void Method(s)	Events Handled	Adapter Class
ActionListener Interface Method		
<code>actionPerformed(ActionEvent e)</code>	A button click, a timer time out, or Enter key strike into a text field	
FocusListener Interface Methods		
<code>focusGained(FocusEvent e)</code>	A clickable component (e.g., a button or a text box) is clicked	
<code>focusLost(FocusEvent e)</code>	Another clickable component is clicked	
KeyListener Interface Methods		√
<code>keyPressed(KeyEvent e)</code>	A key is pressed	
<code>keyReleased(KeyEvent e)</code>	A key is released	
<code>keyTyped(KeyEvent e)</code>	A key is typed (pressed and released)	
MouseListener and MouseMotionListener Methods		√
<code>mouseClicked(MouseEvent e)</code>	A mouse click on a GUI component	
<code>mousePressed(MouseEvent e)</code>	A mouse button is pressed down on a GUI component	
<code>mouseDragged(MouseEvent e)</code>	Left mouse button is held down and the mouse is moved on a component	
<code>mouseReleased(MouseEvent e)</code>	A mouse button is released	

The application-dependent processing to be performed when a particular event occurs is coded inside an implementation of the event handler method that handles that event. The name, signature, and interface of the method can be determined by searching the middle column of Table 11.6 for the event to be handled. For example, to perform some processing whenever a key is typed, the processing would be coded inside the `keyTyped` method whose signature is defined in the `KeyListener` interface. The following code fragment counts the number of times the mouse is clicked.

```
public static int numOfClicks = 0;
public void mouseClicked(MouseEvent e)
{
    numOfClicks++;
}
```

The heading of a class that contains the implementation of an event handler method must contain an `implements` clause indicating that it implements the method's interface. This class is usually coded as an *inner* class of the class that declared the GUI component, although the method can be coded in same class that declared the GUI component.

It is good programming practice that the name of the inner class implies the component and the event its method handles or listens for. Consistent with this practice, its name usually ends with the

word *Handler* or *Listener*. For example: `ComputeClickHandler` or `ComputeClickListener` would be good names for an inner class that contained an event handler that performs the processing associated with a click event on a button object named `compute`.

11.4.2 Registering the Event Handler

To complete the process of implementing an event handler, the event handler method must be added to, or *registered* with, the list of methods invoked when the event occurs. As indicated in Figure 11.18, the list is called a *listener list*, and each GUI component added to an application's interface maintains its own listener list.

To add an event handler method to the listener list maintained in a particular GUI component object, a method is invoked whose name begins with *add* followed by the name of the interface that defined the signature of the method: for example, `addMouseListener` or `addActionListener`. The method is invoked on the GUI component object and passed an instance of the class that contains the code of the event handler method. This invocation is said to *register* the event handler in the component's listener list.

The following code fragment adds (registers) the `actionPerformed` method coded in the inner class `ComputeClickHandler` in the listener list of the `JButton` object named `compute`:

```
ComputeClickHandler click = new ComputeClickHandler();
compute.addActionListener(click);
```

This two-step process of registering an event handler method with a GUI component is summarized below:

1. Create an instance of the inner class that implements the event handler method
2. Invoke the method `addNameOfTheInterface` on the GUI component passing it the object created in step 1

A more concise coding of this two-step process uses an anonymous object to register the event handler method, as shown in the following code fragment:

```
compute.addActionListener(new ComputeClickHandler());
```

When the event handler method is not coded inside an inner class, the event handler method is registered by passing the keyword `this` to the method invoked in step 2, as shown in the following code fragment:

```
compute.addActionListener(this);
```

Summary of the Process to Implement an Event Handler

The following four step process is used to implement an event handler method.

1. Use the middle column of Table 11.6 to determine the API method that handles the event

2. Add that method to an inner class that implements the method's interface, which is identified above the method in Table 11.6
3. Add the instructions to perform the event handling processing to the method
4. Add the method to (register it with) the listener list of the GUI component on which the event will occur: `guiComponent.addInterfaceName(innerClassObject) ;`

NOTE

All of the interface's methods must be implemented in the class. Some of the implementations can contain an empty code block.

Having gained an understanding of events and event handler methods, we will conclude this section with incorporating the event process into our adding machine GUI builder worker class (Figure 11.15), and a discussion of the `getSource` method.

Completion of the Adding Machine Application

The class `AddingMachineGUI2`, shown in Figure 11.19, adds two button click event handler methods to the class `AddingMachineGUI` shown in Figure 11.15 and registers them with the GUI's button objects. One method adds two inputs and outputs the sum with two digits of precision, and the other method clears the inputs and the computed sum. The program's window after the user enters two inputs and clicks the *Compute* button is shown in Figure 11.20a, and Figure 11.20b shows the window after a subsequent click of the *Clear* button.

The inner classes `ComputeClickHandler` and `ClearClickHandler` (lines 67–82 and 83–91, respectively) are the two event handler classes. Each class implements the API interface `ActionListener` (lines 67 and 83) because they will handle a button click event. The middle column of Table 11.6 was used to determine that the inner classes would have to implement the interface `ActionListener` to handle the user button click events, and to determine that the name of the event handler methods would have to be `actionPerformed`.

`ComputeClickHandler`'s implementation of the interface's `actionPerformed` method (lines 69–81) computes and displays the sum of the user inputs when the `compute` button is clicked because it is added to (registered with) that object's listener list on line 51. This registration is performed on line 51 by passing an anonymous instance of the event handler's class `ComputeClickHandler` to the `addActionListener` method invoked on the `compute` object.

Similarly, line 52 adds the implementation of the `actionPerformed` method coded on lines 85–90 to the listener list of the `clear` button object by passing an anonymous instance of its class, `ClearClickHandler`, to the `addActionListener` method invoked on the `clear` object. Line 4 imports the event interfaces and classes used in the GUI builder class.

The revised code of the application `AddingMachine`, which is named `AddingMachineV2`, is shown in Figure 11.21. The program window declared on line 8 of the application is now an instance of the class `AddingMachineGUIV2`.

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.text.DecimalFormat;
4  import java.awt.event.*;
5
6  public class AddingMachineGUIV2 extends JFrame
7  {
8      JLabel description, plus, equals, sum, a, b;
9      JTextField aValue, bValue;
10     JButton compute, clear;
11
12     public AddingMachineGUIV2(String title)
13     {
14         super(title);    //Creates the window. All subsequent invocations
15         setSize(500, 250); //on unnamed object operate on this window.
16         setLocation(200, 100);
17         setLayout(null);
18
19         //Step 1 create the components
20         description = new JLabel("Computes a + b");
21         aValue = new JTextField();
22         plus = new JLabel("+");
23         bValue = new JTextField();
24         equals = new JLabel("=");
25         sum = new JLabel("x,xxx.xx");
26         a = new JLabel("a");
27         b = new JLabel("b");
28         compute = new JButton("Compute");
29         clear = new JButton("Clear");
30
31         //Step 2: specify the component's properties
32         description.setBounds(120, 0, 300, 30);
33         description.setFont(new Font("Sherif", Font.BOLD, 24));
34         aValue.setBounds(60, 50, 100, 30);
35         plus.setBounds(190, 50, 20, 30);
36         plus.setFont(new Font("Sherif", Font.BOLD, 20));
37         bValue.setBounds(230, 50, 100, 30);
38         equals.setBounds(350, 50, 20, 30);
39         equals.setFont(new Font("Sherif", Font.BOLD, 20));
40         sum.setBounds(380, 50, 100, 30);
41         sum.setFont(new Font("Sherif", Font.BOLD, 20));
42         a.setBounds(105, 75, 20, 30);
43         a.setFont(new Font("Sherif", Font.BOLD, 20));
44         b.setBounds(275, 75, 20, 30);
45         b.setFont(new Font("Sherif", Font.BOLD, 20));
46         compute.setBounds(65, 110, 90, 25);
47         clear.setBounds(235, 110, 90, 25);
48         clear.setToolTipText("Clears a, b and the sum");

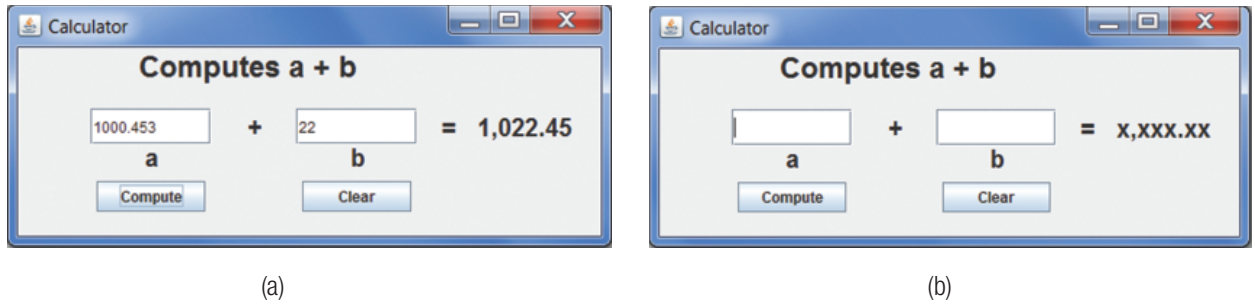
```

```

49
50 // Register the event handler methods
51 compute.addActionListener(new ComputeClickHandler());
52 clear.addActionListener(new ClearClickHandler());
53
54 //Step 4: add the component to the container
55 add(description);
56 add(aValue);
57 add(plus);
58 add(bValue);
59 add(equals);
60 add(sum);
61 add(a);
62 add(b);
63 add(compute);
64 add(clear);
65 }
66 //Event handler inner classes and methods
67 public class ComputeClickHandler implements ActionListener
68 {
69     public void actionPerformed(ActionEvent e)
70     {
71         String s;
72         double a, b, result;
73         DecimalFormat f = new DecimalFormat("#,##0.00");
74
75         s = aValue.getText();
76         a = Double.parseDouble(s);
77         s = bValue.getText();
78         b = Double.parseDouble(s);
79         result = a + b;
80         sum.setText(f.format(result));
81     }
82 }
83 public class ClearClickHandler implements ActionListener
84 {
85     public void actionPerformed(ActionEvent e)
86     {
87         aValue.setText("");
88         bValue.setText("");
89         sum.setText("x,xxx.xx");
90     }
91 }
92 }

```

Figure 11.19The class **AddingMachineGUIV2**.

**Figure 11.20**

Output generated by the class **AddingMachineGUIV2**'s Compute and Clear button.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class AddingMachineV2
5  {
6      public static void main(String[] args)
7      {
8          AddingMachineGUIV2 calculator = new AddingMachineGUIV2("Calculator");
9          calculator.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         calculator.setVisible(true);
11     }
12 }

```

Figure 11.21

The application **AddingMachineV2**.

The getSource Method

When an event handler method is invoked, the argument passed to it contains information about the event, which includes the address of the GUI object involved in the event. For example, when the event handler that begins on line 69 of Figure 11.19 is invoked, the information passed to its parameter *e* contains the address of the object *compute*, declared on line 28 of the figure. This is the only object that could have been involved in the event because the object *clear* (declared on line 29) added the other implementation of the *actionPerformed* event handler to its listener list (line 52). As a result, the information passed to the parameter *e* was not used.

When an alternative coding style is used to implement the event handler, the information passed to the parameter *e* plays an essential role in the coding of the event handler. In this alternative approach, only one implementation of the event handler is coded, and all of the objects on which this event could occur register that implementation into their listener list. Assuming the inner class containing the one event handler method implementation is named *ButtonClickHandler*, lines 51 and 52 of Figure 11.19 would become:

```

51     compute.addActionListener(new ButtonClickHandler());
52     clear.addActionListener(new ButtonClickHandler());

```


To discern which of the two buttons, `compute` or `clear`, was involved in a button click event, the GUI component object address contained in the information passed to the parameter `e` is compared to the addresses stored in the variables `compute` and `clear`. The method `getSource` in the `EventObject` class is used to fetch the address from the information passed to `e`. Figure 11.22 presents the code that would replace lines 67–82 and lines 83–92 of Figure 11.19 when this alternative implementation style is used. It presupposes that the previously discussed recoding of lines 51 and 52 has been performed.

The name of the (one) inner class whose heading is on line 67 of Figure 11.22 is now the generic name `ButtonClickHandler`, and the Boolean condition of the `if-else` clauses that begin on lines 75 and 84 use the `getSource` method to determine the object involved in the event. If the GUI contained more than two buttons, additional nested `if-else` clauses would be added to the method to process their click events.

```

67     public class ButtonClickHandler implements ActionListener
68     {
69         public void actionPerformed(ActionEvent e)
70         {
71             String s;
72             double a, b, result;
73             DecimalFormat f = new DecimalFormat("#,##0.00");
74
75             if(e.getSource() == compute)
76             {
77                 s = aValue.getText();
78                 a = Double.parseDouble(s);
79                 s = bValue.getText();
80                 b = Double.parseDouble(s);
81                 result = a + b;
82                 sum.setText(f.format(result));
83             }
84             else if(e.getSource() == clear)
85             {
86                 aValue.setText("");
87                 bValue.setText("");
88                 sum.setText("x,xxx.xx");
89             }
90         }
91     }
92 }
```

Figure 11.22

An alternative coding style of the event handlers that begin on line 67 of Figure 11.19.

11.4.3 Paint Events, JPanels, and Two-Dimensional Graphics

A *paint event* is a generic term for any event that causes a graphical object to be drawn or redrawn. The most obvious paint events are the initial display of an application's GUI window and maximizing a window after it has been minimized. A more subtle paint event is continuously redrawing a window as it is dragged across the monitor.

To display a window on the monitor, the window (`JFrame`) object and all of the components added to it (`JButton` objects, `JTextFields` objects, etc.) have to be drawn, as do any two-dimensional (2D) shapes that were drawn in the window by the application using the methods in the `Graphics` class. To accomplish this, each component has a paint event handler, or call back method, that is invoked when a paint event occurs. The names of these event handlers typically begin with the word *paint*, for example `paintComponents`, `paintComponent`, and `paint`.

When they are invoked, these methods are passed a `Graphics` object, which they use to render (draw) the component. The `Graphics` object passed to the `JPanel` class's `paintComponent` call back method can be used to draw 2D graphics shapes defined in the `Graphics` class on a `JPanel` object. To accomplish this, we simply add a class to our application that extends `JPanel` and overrides its `paintComponent` method. Then, the panel is added to the application's window. The drawing of the 2D graphics is done inside the overridden version of the `paintComponent` method.

For example, the following code fragment would be added to a class that extends `JPanel` to draw a filled red rectangle 70 pixels wide by 30 pixels high, whose upper left corner is at (300, 200). The parent (`JPanel`) class's overridden method should always be invoked as the first line child's version of the method.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    g.setColor(Color.RED);
    g.fillRect(300, 200, 70, 30);
}
```

The class `BoxedSnowmanV3`, shown in Figure 11.23, extends the class `JPanel` and overrides its `paintComponent` method on lines 19–34. The overridden version of the method uses the `Graphics` object `g` passed to it to invoke the drawing methods of the `Graphic` class on lines 22–33.

The application `Graphics2D` shown in Figure 11.24 produces the output shown in Figure 11.25. The application does not import the game environment to perform the graphics displayed in its window. Instead it declares an instance of the class `BoxedSnowmanV3`, shown in Figure 11.24 on line 10, named `s1` and adds this extended `JPanel` GUI component (line 11) to the `JFrame` window created on line 8. Every time the `JFrame` window is drawn or redrawn, the Java Runtime Environment invokes the `JPanel` component's `paintComponent` method to redraw component `s1`, just as it would do for any other GUI component that is added to a `JFrame`. Because the method is overridden, the 2D shapes are drawn or redrawn.

NOTE

When one component is added to a `JFrame`, by default, it occupies `JFrame`'s entire content pane.

The `paintComponent` method begins by invoking `JPanel`'s overridden version of the method on line 21. As previously mentioned, this should always be the first line of the overridden version of the method. Lines 22–31 draw a snowman with a rectangle around it, and lines 32–33 change the font from its default value and draw the string shown at the top of the application's window.

The difference between this graphical output and the graphical outputs produced by programs that use the game environment is that the method that performs the 2D graphics (lines 19–34 of Figure 11.23) is invoked by the Java Runtime Environment rather than the game environment.

There is also a subtle but very important difference between the `BoxedSnowmanV3` class shown in Figure 11.23 and the `BoxedSnowman` class shown in Figure 4.4, which was part of an application that imported the game environment. The names of the `set` and `get` methods on lines 35–50 of Figure 11.23 have been changed from the names `setX`, `getX`, `setY`, and `getY` used in Figure 4.4. The `get` method names were changed because the `BoxedSnowman` class inherits methods named `getX` and `getY` from the `JPanel` class. Because these methods could be invoked by the Runtime Environment to determine the location of the `JPanel` when a paint event occurs, they should not be overridden. The names of the snowman's `set` and `get` methods were changed to `setXS`, `getXS`, `setYS`, and `getYS` for consistency and readability.

```

1  import java.awt.*;
2  import javax.swing.*;
3
4  public class BoxedSnowmanV3 extends JPanel
5  {
6      private int x = 8;
7      private int y = 30;
8      private Color hatColor = Color.BLACK;
9      private int dx = 0;
10     private int dy = 0;
11     private int time = 0;
12
13     public BoxedSnowmanV3(int initialX, int initialY, Color hatColor)
14     {
15         x = initialX;
16         y = initialY;
17         this.hatColor = hatColor;
18     }
19     public void paintComponent(Graphics g)
20     {
21         super.paintComponent(g);
22         g.setColor(hatColor);
23         g.fillRect(x + 15, y, 10, 15); //hat
24         g.fillRect(x + 10, y + 15, 20, 2); //brim
25         g.setColor(Color.WHITE);

```

```

26     g.fillOval(x + 10, y + 17, 20, 20); //head
27     g.fillOval(x, y + 37, 40, 40); //body
28     g.setColor(Color.RED);
29     g.fillOval(x + 19, y + 53, 4, 4); //button
30     g.setColor(Color.BLACK);
31     g.drawRect(x, y, 40, 77); //inscribing rectangle
32     g.setFont(new Font("Sherif", Font.BOLD, 20)); //time format
33     g.drawString("Time: " + time, 300, 50); //time
34 }
35 public int getXS()
36 {
37     return x;
38 }
39 public void setXS(int newX)
40 {
41     x = newX;
42 }
43 public int getYS()
44 {
45     return y;
46 }
47 public void setYS(int newY)
48 {
49     y = newY;
50 }
51 }

```

Figure 11.23

The class **BoxedSnowmanV3**.

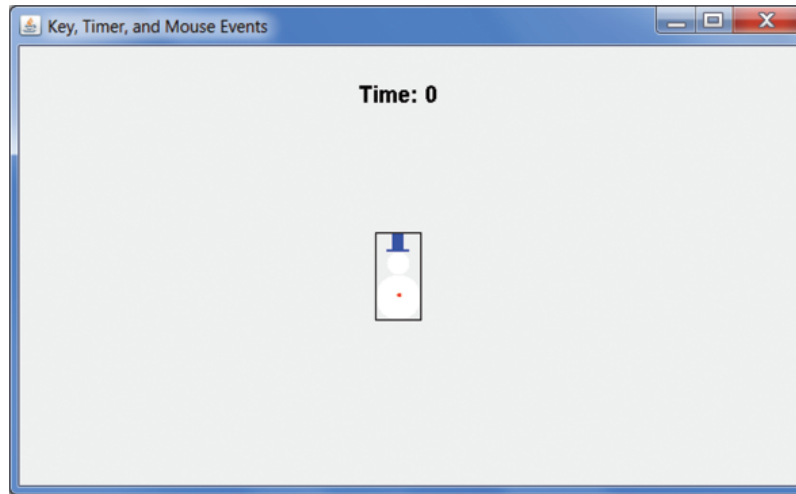
```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class Graphics2D extends JFrame
5  {
6      public static void main(String[] args)
7      {
8          JFrame window = new JFrame("Graphics");
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         BoxedSnowmanV3 s1 = new BoxedSnowmanV3(315, 165, Color.BLUE);
11         window.add(s1);
12         window.setSize(708, 434);
13         window.setVisible(true);
14     }
15 }

```

Figure 11.24

The application **Graphics2D**.

**Figure 11.25**

The output produced by the application **Graphics2D**.

11.4.4 Mouse, Keyboard, and Timer Events

Many of the GUI components included in the API respond to mouse-click events and keyboard events such as the pressing or typing of a key. The most obvious examples of this are radio-button event handlers and check-box event handlers that change the appearance of these components when they are clicked. Another example is the appearance of the caret in a text field when it is clicked, and the subsequent display of typed characters in the text field to the left of the caret. The GUI support in the API includes implementations of these non-application-dependent event handlers and their registrations.

Just as GUI components respond to these events to maintain their look and feel, there is often the need for an application to respond to a mouse or keyboard event in an application-dependent manner. Application-dependent processing of a `JButton` click event has already been discussed in this chapter. Other mouse events, such as the dragging of the mouse or clicking on a portion of the application's window that does not contain a GUI atomic component may be an important event in a particular application. For example, a mouse-drag event could be used to move the snowman shown in Figure 11.25 to another location in the program window, or a key-typed event such as typing the cursor control (arrow) keys could be used to move the snowman around the window. Similarly, a timer-tick event is certainly an important event to many applications that are time dependent.

In this section, we will learn how to incorporate mouse, keyboard, and timer events into any Java application. Although we have reacted to some of these events in our game programs, the details of the timer, keyboard, and mouse event handler implementations and their registration was performed by the game environment. Essentially, in the remainder of this section, we will gain insights into how this was accomplished by the game environment, so we can process these events in any application we write.

Keyboard and mouse events are processed within an application using the techniques discussed in Sections 11.4.1 and 11.4.2. An event handler method, whose signature is defined in an API interface, is implemented, and the method is added to (registered in) a list of methods invoked when the event occurs. The application-specific processing to be performed when the event occurs is coded into the event handler method. Timer events are processed in a similar way, but the syntax of registering the event handler is slightly different from that discussed in Section 11.4.2.

Mouse Events

There are eight mouse events associated with event handler methods whose signatures are defined in the API interfaces `MouseListener`, `MouseMotionListener`, and `MouseWheelListener`. The signatures of the eight methods and the interfaces that define them are given below, and four of the more commonly used methods are also described in Table 11.6.

```
void mouseClicked(MouseEvent e)    //defined in MouseListener
void mouseEntered(MouseEvent e)    //  "      "      "
void mouseExited(MouseEvent e)     //  "      "      "
void mousePressed(MouseEvent e)    //  "      "      "
void mouseReleased(MouseEvent e)   //  "      "      "
void mouseDragged(MouseEvent e)    //defined in MouseMotionListener
void mouseMoved(MouseEvent e)      //  "      "      "
void MouseWheelMoved(MouseEvent e) //defined in MouseWheelListener
```

The names of the methods are representative of the mouse events they handle (the events that cause them to be invoked). The `mouseEntered` and `mouseExited` methods are invoked when the mouse cursor enters and exits the perimeter of a GUI component on which the event is registered.

To perform application-dependent processing when a mouse event occurs, the worker class that defines the GUI implements the relevant mouse event handler method and registers it using the `addMouseListener`, `addMouseMotionListener`, or `addMouseWheelListener` methods. The application-dependent processing is coded inside the method's code block. If the API interface that defines the method's signature contains multiple signature definitions, as the interfaces `MouseListener` and `MouseMotionListener` do, all the methods defined in the interface must be implemented in the worker class or within one of its inner classes. These additional implementations can contain empty code blocks. To avoid coding the additional methods and their empty code blocks, the class can extend the API `MouseAdapter` class instead of implementing one of the three mouse listener interfaces.

Event Handler Adapter Classes

An adapter class is a class that implements all the methods defined in one or more interfaces and provides empty implementations of the methods defined in the interface(s). The advantage of this is that their child classes inherit the empty implementations and can then simply implement, or more accurately, override the methods that handle the events of interest to them. For example, if only mouse-released events were to be processed by a GUI class, it would only have to override the `mouseReleased` method if it extended the API adapter class `MouseAdapter`, instead of implementing the `MouseListener` interface.

NOTE

It is good programming style to end the name of an adapter class with the word adapter.

Many of the API interfaces that define multiple method signatures have adapter classes associated with them. The class `MouseListenerAdapter` implements all eight mouse event handler methods. Because a class can only inherit from one class, most often the extension of an adapter class is performed by an inner class, and the relevant event handler methods are overridden within it.

The code fragment shown below registers an implementation of a mouse-entered event handler that outputs the message *Button b1 was Entered* every time the mouse pointer enters the boundaries of the `JButton` object `b1`.

```
JButton b1 = new JButton("Enter Test");
b1.addMouseListener(new MouseHandler()); //register the event handler

public class MouseHandler extends MouseAdapter
{
    public void mouseEntered(MouseEvent e) //only method implemented
    {
        System.out.println("Button b1 was Entered");
    }
}
```

The argument passed to the mouse event handler method's parameter `e` is the address of a `MouseEvent` object. All eight of the mouse event handler methods are passed a reference to an instance of this class. The class contains several methods that can be used to obtain information related to the mouse event. For example the (x, y) position of the mouse when the event occurred can be determined by invoking the class's `getX` and `getY` methods on the object `e` references. The button on the mouse that was pressed can be determined from the integer returned from the method `getButton()`, and the number of mouse clicks performed on an GUI component can be determined from the integer returned from the method `getClickCount()`. The GUI component object on which the mouse event occurred can be determined by invoking `MouseEvent`'s inherited method `getSource()`.

The following code fragment outputs the location at which the mouse cursor entered the boundaries of the `JButton` object `b1`:

```
JButton b1 = new JButton("Enter Test");
b1.addMouseListener(new MouseHandler()); //register the event handler

public class MouseHandler extends MouseAdapter
{
    public void mouseEntered(MouseEvent e) //only method implemented
    {
        System.out.println("Button b1 was Entered at pixel location (" +
            e.getX() + ", " + e.getY() + ")");
    }
}
```


Keyboard Events

There are three keyboard events associated with event handler methods whose signatures are defined in the API interface `KeyListener`. These signatures are presented in Table 11.6 and described below:

```
void keyPressed(KeyEvent e)
void keyReleased(KeyEvent e)
void keyTyped(KeyEvent e)
```

The `keyPressed` event handler method is invoked whenever a key is struck, and the `keyReleased` method is invoked when the key is released. In addition, the `keyTyped` method is invoked whenever a *non-action* key is struck (i.e., a key-typed event does *not* occur when an *action* key is struck). The action keys include the Shift, Num Lock, End, Home, Caps Lock, function keys, arrow keys, etc.

Whenever a non-action key is held down, it repeatedly generates a key-pressed event followed by a key-typed event. If the key held down is an action key, only key-pressed events are generated. In either case, when the key is released, a single key-released event occurs.

A class that services a key event must implement all three methods defined in the API interface `KeyListener` or extend the adapter class `KeyAdapter`. Key event handler methods are registered using the `addKeyListener` method. The argument passed to the parameter of the three mouse event handler methods is the address of a `KeyEvent` object. This object's class contains several methods that can be used to determine the key that caused the keyboard event. The method `getKeyChar()` returns the character generated by the non-action keys (lower and upper case). The returned character can be used to determine which non-action key has been struck. The following key event handler performs its output when the key P is typed:

```
public void keyTyped(KeyEvent e)
{
    if(e.getKeyChar == 'P' || e.getKeyChar == 'p')
    {
        System.out.println("The P key was struck");
    }
}
```

When action keys are struck, they can be identified using the `getKeyCode` method in the class `KeyEvent`. It returns the integer key code of the key struck. This integer can be passed to the class's static `getKeyText` method, which returns a string that describes the key's code: e.g., "Right" for the right arrow key. The strings associated with each of the action keys can be easily identified by including the following statement inside an implementation of a `keyPressed` event handler method, assuming the variable `e` is the name of the method's parameter:

```
System.out.println(KeyEvent.getKeyText(e.getKeyCode()));
```

The game environment passes the first character of the strings associated with the action keys to its call back method `keyStruck` (e.g., 'R' when the right arrow key is struck).

Keyboard Focus

For a component to respond to a keyboard event, it must have the keyboard's focus, which can be thought of as assigning (or attaching) the keyboard to a GUI component. Generally speaking, only one component in an application can have the keyboard's focus at any given time. When a component has the keyboard's focus, the key event handler methods registered in the component's event listener list will be executed when a key event occurs.

The `requestFocusInWindow` method, inherited from the `JComponent` class, is invoked on a `Swing` component to transfer the application's input focus to the component in a platform-independent way. For the transfer to take place, the component must have already been added to the application (i.e., has been rendered, is visible, etc.). To ensure that this has taken place when the keyboard focus is requested, the invocation should be made from within an overwritten version of the `addNotify` method, which is invoked when a component is added to an application. `Swing` components inherit this method from the `JComponent` class.

The code fragment shown in Figure 11.26 would be added to the `BoxedSnowmanV3` class shown in Figure 11.23 to register an implementation of a key event handler that outputs *Key Pressed* to the system console followed by the text generated by the key whenever a key is pressed.

```
addKeyListener(new KeyHandler()); //add keyPressed to listener list

public void addNotify() //invoked when component is added to application
{
    super.addNotify(); //invokes JComponent's addNotify method
    requestFocusInWindow(); //obtains the keyboard's focus
}

public class KeyHandler extends KeyAdapter
{
    public void keyPressed(KeyEvent e)
    {
        System.out.println("Key Pressed");
        System.out.println(KeyEvent.getKeyText(e.getKeyCode()));
    }
}
```

Figure 11.26

Code to add a key-pressed event handler to a GUI component class.

Timer Events

A timer event is an event produced by an object in the class API class `Timer`. It can be likened to the bell sounding on an egg timer after its time interval has expired. The time interval of a `Timer` object is the first argument passed to the constructor when the object is created. This integer value specifies the time interval in milliseconds (e.g., 1000 for a one-second interval). The constructor's second parameter is used to register the timer event handler method, which must be an

instance of a class that implements the method `actionPerformed`. The signature of this method is defined in the `ActionListener` interface and given at the top of Table 11.6. A `Timer` object's interval begins with an invocation of the class's `start` method, and the timer generates an action event at the end of subsequent timer intervals (ticks).

The code fragment shown in Figure 11.27 simulates a three-minute egg timer that outputs *The egg is cooked* to the system console after three minutes have elapsed from the time when the `start` method is invoked on the timer object `eggTimer`.

```
int interval = 1000 * 60 * 3; //3 minutes (1,000 ms * 60 sec * 3 min)
Timer eggTimer = new Timer(interval, new EggTimerHandler());

eggTimer.start(); //the eggTimer's time interval begins

public class EggTimerHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e) //Timer's event handler
    {
        System.out.println("The egg is cooked");
    }
}
```

Figure 11.27

Implementing a timer object and processing its timer event.

The code fragment shown in Figure 11.27 produces a line of output every three minutes because, by default when a timer's time interval expires, it is restarted. To stop this process, the `Timer` class's `stop` method is invoked. For example, to cook one egg, the following line of code would be added to the end of the `actionPerformed` method's code block shown in Figure 11.27:

```
eggTimer.stop(); //stops eggTimer from generating timer events
```

Alternately, the `Timer` class's `setRepeats` method can be passed the value `false` before the timer is started to prevent the timer from generating more than one timer event. The `setDelay` method in the `Timer` class can be used to delay the restarting of a running timer after a time event has occurred.

The class `BoxedSnowmanV4` shown in Figure 11.28 demonstrates the processing of timer, mouse, and keyboard events. It adds a timer event handler, a key event handler, and five mouse event handlers to the `BoxedSnowmanV3` class shown in Figure 11.23. These event handlers are coded as lines 57–61 and 67–106 in Figure 11.28. Lines 1–56 of that figure are the original version of the class with six (highlighted) lines added to it. The application `MouseKeyboardAndTimerEvents` shown in Figure 11.29 creates an instance of a boxed snowman on line 10 and adds the object (line 11) to the `JFrame` window it creates on line 8. Finally, it displays the window (line 13), which is shown in Figure 11.30a.

The number of seconds since the program has been launched is displayed at the top of the window. The program user can reposition the snowman in the window by dragging it to a new location

(Figure 11.30b), clicking its new window position (Figure 11.30c), or moving it right by pressing the keyboard's right arrow action key (Figure 11.30d).

The event handler `actionPerformed`, coded on lines 57–61 of the class shown in Figure 11.28, processes the timer events generated by the timer declared on line 13 and started on line 23. The timer's increment, passed to the constructor on line 13, is 1000 milliseconds (1 second) and the keyword `this`, passed to the constructor's second parameter, registers the `actionPerformed` method coded in this class as the timer's event handler. The class's heading (line 5) indicates that it implements the `ActionListener` interface.

After each timer event which is separated by one second, line 59 of the timer event handler increments the variable `time` and line 39 outputs the elapsed time to the top of the application's window (Figure 11.30b). The invocations of the `repaint` method on lines 60, 75, 85, and 96 have been added to the event handler methods to force a repainting of the class's `JPanel` at the end of their execution. These invocations cause the overridden version of the `paintComponent` method (lines 25–40) to execute.

All the mouse event handlers are coded inside the inner class `MouseHandler` that begins on line 79, and they are registered in the `JPanel`'s listener list on lines 20 and 21 using an anonymous instance of the inner class. When the left mouse button is pressed to initiate the dragging of the snowman to a new position, the `mousePressed` event handler coded on lines 87–91 executes. It computes the *x* and *y* separation (*dx* and *dy*) between the snowman's upper left corner (*xS*, *yS*) and the mouse pointer's current location returned from the invocations `e.getX()` and `e.getY()`.

As the mouse is dragged, the `mouseDragged` event handler, coded on lines 81 to 86, is continually invoked. It subtracts the *x* and *y* separations (*dx* and *dy*) from the current position of the mouse pointer to determine the new location of the snowman's upper left corner. This gives the appearance that the snowman is being dragged by the mouse pointer. Figure 11.30b shows the snowman's position ten seconds after the game began and after the mouse was dragged to the upper left portion of the window.

When the mouse is clicked, the `mouseClicked` event handler, coded on lines 92–97, executes. It sets the location of the upper left corner of the snowman to the location of the mouse pointer on lines 94–95. This gives the appearance that the snowman has jumped to the clicked location. Figure 11.30c shows the snowman's position 15 seconds after the game began and after the mouse was clicked in the lower right portion of the window.

Whenever the mouse pointer enters or exits the boundaries of the `BoxedSnowmanV4` `JPanel`, which was added to the application's window (line 11 of Figure 11.29) and occupies its entire content pane, the `mouseEntered` or `mouseExited` event handler methods (Figure 11.28, lines 98–105) execute. Lines 100 and 104 then produce the output *Entered* or *Exited* on the system console. The system console output, shown at the bottom of Figure 11.30, was produced by moving the mouse cursor on to the window's content pane after the program was launched, then moving it off the pane.

The `addNotify` method that begins on line 62 in Figure 11.28 is invoked when the application adds the `BoxedSnowmanV4` object `s1` to the program's window. Line 65 transfers the application's

input focus to the object's `JPanel`. Subsequent key-pressed events then invoke the event key handler `KeyPressed` (lines 69–77), which is coded inside the inner class `KeyHandler` (lines 67–78). The event handler is registered with the `JPanel`'s listener list on line 22. The method uses the string returned from the `KeyEvent` class's `getKeyText` method, invoked on line 71, to determine if the right arrow action key has been pressed (line 72). When the key is pressed, the code on line 74 moves the snowman three pixels to the right. The lower portion of Figure 11.30 shows the position of the snowman 15 seconds after the game began (Figure 11.30c) and after the right arrow action key has been held down for one second (Figure 11.30d).

```

1  import java.awt.*;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  public class BoxedSnowmanV4 extends JPanel implements ActionListener
6  {
7      private int xS = 8;
8      private int yS = 30;
9      private Color hatColor = Color.BLACK;
10     private int dx = 0;
11     private int dy = 0;
12     private int time = 0;
13     private Timer aTimer = new Timer(1000, this);
14
15     public BoxedSnowmanV4(int initialX, int initialY, Color hatColor)
16     {
17         xS = initialX;
18         yS = initialY;
19         this.hatColor = hatColor;
20         addMouseListener(new MouseHandler());
21         addMouseMotionListener(new MouseHandler());
22         addKeyListener(new KeyHandler());
23         aTimer.start();
24     }
25     public void paintComponent(Graphics g)
26     {
27         super.paintComponent(g);
28         g.setColor(hatColor);
29         g.fillRect(xS + 15, yS, 10, 15); // hat
30         g.fillRect(xS + 10, yS + 15, 20, 2); // brim
31         g.setColor(Color.WHITE);
32         g.fillOval(xS + 10, yS + 17, 20, 20); // head
33         g.fillOval(xS, yS + 37, 40, 40); // body
34         g.setColor(Color.RED);
35         g.fillOval(xS + 19, yS + 53, 4, 4); //button
36         g.setColor(Color.BLACK);
37         g.drawRect(xS, yS, 40, 77); // inscribing rectangle
38         g.setFont(new Font("Sherif", Font.BOLD, 20));

```

```

39     g.drawString("Time: " + time, 300, 50);
40 }
41 public int getXS()
42 {
43     return xS;
44 }
45 public void setXS(int newX)
46 {
47     xS = newX;
48 }
49 public int getYS()
50 {
51     return yS;
52 }
53 public void setYS(int newY)
54 {
55     yS = newY;
56 }
57 public void actionPerformed(ActionEvent e)
58 {
59     time++;
60     repaint();
61 }
62 public void addNotify()
63 {
64     super.addNotify();
65     requestFocusInWindow();
66 }
67 public class KeyHandler extends KeyAdapter
68 {
69     public void keyPressed(KeyEvent e)
70     {
71         String key = KeyEvent.getKeyText(e.getKeyCode());
72         if(key.equals("Right"))
73         {
74             xS = xS + 3;
75             repaint();
76         }
77     }
78 }
79 public class MouseHandler extends MouseAdapter
80 {
81     public void mouseDragged(MouseEvent e)
82     {
83         xS = e.getX() - dx;
84         yS = e.getY() - dy;
85         repaint();
86     }
87     public void mousePressed(MouseEvent e)

```

```

88     {
89         dx = e.getX() - xS;
90         dy = e.getY() - yS;
91     }
92     public void mouseClicked(MouseEvent e)
93     {
94         xS = e.getX();
95         yS = e.getY();
96         repaint();
97     }
98     public void mouseEntered(MouseEvent e)
99     {
100         System.out.println("Entered");
101     }
102     public void mouseExited(MouseEvent e)
103     {
104         System.out.println("Exited");
105     }
106 }
107 }

```

Figure 11.28

The class **BoxedSnowmanV4**.

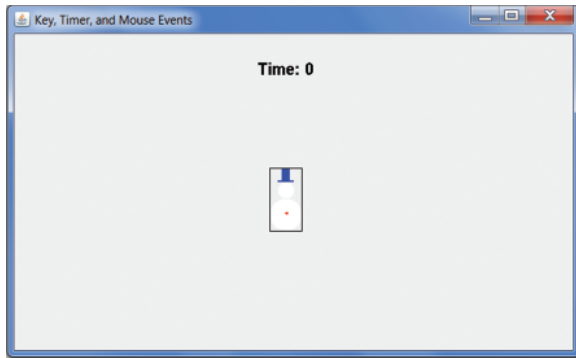
```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class MouseKeyboardAndTimerEvents extends JFrame
5  {
6      public static void main(String[] args)
7      {
8          JFrame window = new JFrame("MOUSE, KEYBOARD, AND TIMER EVENTS");
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         BoxedSnowmanV4 s1 = new BoxedSnowmanV4(315, 165, Color.BLUE);
11         window.add(s1);
12         window.setSize(708, 434);
13         window.setVisible(true);
14     }
15 }

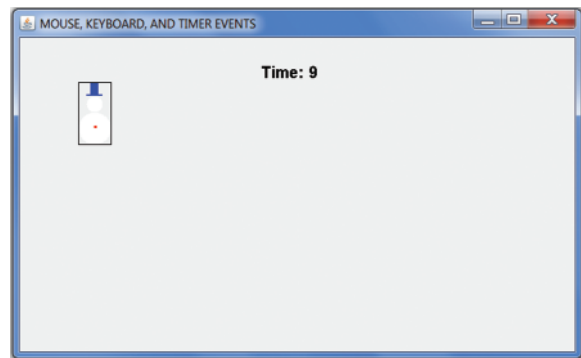
```

Figure 11.29

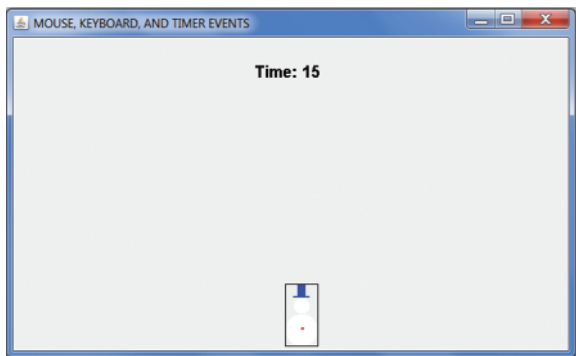
The application **MouseKeyboardAndTimerEvents**.



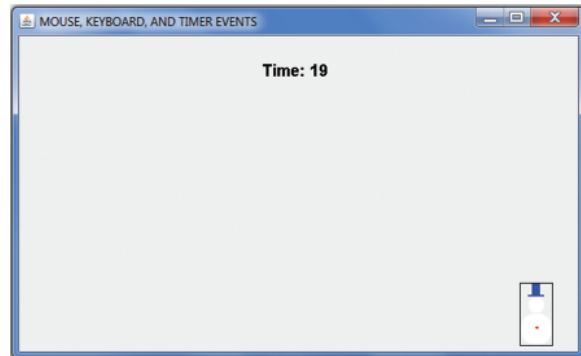
(a)



(b)



(c)



(d)

Console output:

Entered

Exited

Figure 11.30

The window and console output produced by the application `MouseKeyboardAndTimerEvents`.

11.5 LAYOUT MANAGERS

Java provides three *layout managers* that can facilitate the tedious process of locating components in a window when an IDE with a GUI builder is not available. By default they manage the placement and sizing of the components added to a `JPanel` or `JFrame` container. Their names are border layout, flow layout, and grid layout. Each layout manager uses a different approach to position and size the components added to a container.

Components are added to the container using the `Container` class's `add` method discussed in Section 11.3.3. The method is overloaded, and the version of the method used depends on the layout manager being used to position and size the components. The following code fragment uses the one-parameter version of the method to add the string annotation *Computes $a + b$* to a GUI that is using its default border layout manager to position and size components:


```
JLabel description = new JLabel("Computes a + b");
add(description);
```

When layout managers are used to build a graphical interface, most often the GUI's atomic components are added to `JPanels`, and the layout manager is used to position the panels which in the interface. This greatly extends the usefulness of the layout managers. In addition, because atomic components are centered in panels, the use of panels makes the interface visually appealing. The following code fragment adds a `JLabel` containing the string annotation *Computes a + b* to a `JPanel` named `panel1`, and then `panel1` is added to the GUI interface:

```
// Add a JLabel to a JPanel, and the JPanel to a GUI
JPanel panel1 = new JPanel();
JLabel description = new JLabel("Computes a + b");
panel1.add(description); //add the JLabel to the JPanel panel1
add(panel1); //add panel1 to the GUI
```

The positioning and sizing processes performed by the layout managers are implemented in the API classes `BorderLayout`, `FlowLayout`, and `GridLayout`. `JPanels` and `JFrame` objects store a reference to an object in one of these classes whose methods are used by the layout manager to size and locate the components added to them.

11.5.1 Designating the Layout Manager

By default, a `JPanel` uses *flow* layout and a `JFrame` uses *border* layout. These defaults can be overridden, or grid layout can be selected by invoking the `Container` class's `setLayout` method on a `JPanel` or `JFrame` object and passing it an instance of the layout manager class to be used to position the components added to the container. For example, the following code fragment sets the layout manager of a `JPanel` to border layout, overriding its default flow layout. Normally, a nameless object is passed to the method.

```
JPanel myPanel = new JPanel();
myPanel.setLayout(new BorderLayout()); //use border layout
```

To take control of the positioning and sizing of components within a container, a `null` value is passed to `setLayout`'s parameter. When this is done, the components added to the container are positioned and sized using invocations to the `setBounds`, `setSize`, and `setLocation` methods, as discussed in Section 11.3.3.

```
// Programmer will specify the atomic components' size/location
JPanel myPanel = new JPanel();
myPanel.setLayout(null);
```

NOTE

A `null` or a non-default layout manager must be designated before components are added to an interface.

While the use of a layout manager can facilitate the building of some GUI interfaces, it limits the programmer's ability to position and size the GUI components that make up the interface.

Table 11.7 summarizes the number of components, size, and positioning limitations imposed by the layout manager classes.

Table 11.7
Component Capacity, Size and Positioning Restrictions of the Layout Managers

Maximum Number of Components	Component Size	Positioning of Components
Border Layout (default layout manager for <code>JFrames</code> and applets: see Section 11.6)		
Five	The height and/or width of each component are adjusted to fit the region to which they are assigned	Components are placed in one of five regions; the region assigned to each component is specified by the programmer, one component per region.
Flow Layout (default layout manager for <code>JPanels</code>)		
Unlimited	No restriction	Components are placed in rows in the order they are added to the container, beginning at the top left of the container. The row height is set to the largest component in the row.
Grid Layout		
Implied as the grid's rows \times columns	All components are sized to the size of the maximum-sized component	Components are placed in rows that contain cells, in the order they are added to the container, beginning with the top left cell, one item per cell. The number of rows and columns in the grid is specified by the programmer.

11.5.2 Border Layout

When the border layout is assigned to a container, it is divided into five regions named north, south, east, west, and center. The positioning of these regions in the container is shown in Figure 11.31. The font size of the text displayed in the south region of the figure was set larger than that of the text displayed in the other regions.



Figure 11.31
The positioning and sizing of the five border layout regions.

The height and width of the five components added to a container are adjusted in the following sequence, regardless of the order in which the program adds the components to the regions:

1. A component added to the north or south region maintains its height, and its width is the width of the container to which it is added
2. A component added to the east or west region maintains its width, and its height is set to the height between the north and south regions
3. The height and width of the component added to the center region is resized to fit between the other four regions

NOTE *When using BorderLayout:*

1. The width of the north and south regions are always equal
2. The height of the east, west, and center regions are always equal

The class `AddingMachineGUI3` shown in Figure 11.32 builds the GUI interface of the adding machine discussed in Section 11.3.3 using the border layout manager. The application `BorderLayout` shown in Figure 11.33 declares an instance of this class on line 8 of the figure and displays the graphical object on line 10. The application's GUI, built by the class `AddingMachineGUI3`, is shown in Figure 11.34. Although it is not identical to the GUI shown in Figure 11.16, which was built by the class `AddingMachineGUI` shown in Figure 11.15, the use of the border layout manager greatly facilitated the implementation of this GUI.

Because the atomic components of the GUI shown in Figure 11.34 are positioned in the upper, center, and lower areas of the interface, the border layout's north, center, and south regions were used in its implementation. In addition, because the center and lower areas contain multiple atomic components, `JPanels` were included in the design to collect each region's components.

The three `JPanels` used to collect the atomic components are declared on lines 9–11 of Figure 11.32, and line 18 designates that the border layout manager will be used to position them in the interface. Because the class extends `JFrame` (line 4), whose default layout manager is `BorderLayout`, this line is not necessary; it is included as an example of how to specify the layout manager. The atomic components are added to the three panels on lines 43–53, and the panels are added to the border layout's north, center, and south regions (lines 54–56).

Before the atomic components are added to the panels, the components are constructed (lines 21–31), and their properties are set (lines 34–40). Lines 23 and 26 use the one-parameter constructor of the `JTextField` class to specify the width of the text fields. The `JLabel` created on line 30 is added to the `panel3` on line 52, in between the `compute` and `clear` buttons, to provide some additional separation between them.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class AddingMachineGUI3 extends JFrame
5  {
6      JLabel description, a, plus, b, equals, sum, centerSpace;

```

```

7      JTextField aValue, bValue;
8      JButton compute, clear;
9      JPanel panel1 = new JPanel();
10     JPanel panel2 = new JPanel();
11     JPanel panel3 = new JPanel();
12
13     public AddingMachineGUI3(String title)
14     {
15         super(title);
16         setSize(475, 150);
17         setLocation(200, 100);
18         setLayout(new BorderLayout());
19
20         //create the atomic components
21         description = new JLabel("Computes a + b");
22         a = new JLabel("a");
23         aValue = new JTextField(5);
24         plus = new JLabel(" + ");
25         b = new JLabel("b");
26         bValue = new JTextField(5);
27         equals = new JLabel(" = ");
28         sum = new JLabel("x,xxx.xx");
29         compute = new JButton("Compute");
30         centerSpace = new JLabel(" ");
31         clear = new JButton(" Clear ");
32
33         //specify the component's properties
34         description.setFont(new Font("Sherif", Font.BOLD, 24));
35         plus.setFont(new Font("Sherif", Font.BOLD, 20));
36         equals.setFont(new Font("Sherif", Font.BOLD, 20));
37         sum.setFont(new Font("Sherif", Font.BOLD, 20));
38         a.setFont(new Font("Sherif", Font.BOLD, 20));
39         b.setFont(new Font("Sherif", Font.BOLD, 20));
40         clear.setToolTipText("Clears a, b and the sum");
41
42         //add the components to the window or non-atomic container
43         panel1.add(description);
44         panel2.add(a);
45         panel2.add(aValue);
46         panel2.add(plus);
47         panel2.add(b);
48         panel2.add(bValue);
49         panel2.add(equals);
50         panel2.add(sum);
51         panel3.add(compute);
52         panel3.add(centerSpace);
53         panel3.add(clear);
54         add(panel1, BorderLayout.NORTH);

```

```

55     add(panel2, BorderLayout.CENTER);
56     add(panel3, BorderLayout.SOUTH);
57 }
58 }

```

Figure 11.32

The class **AddingMachineGUI3** that uses the **BorderLayout** manager.

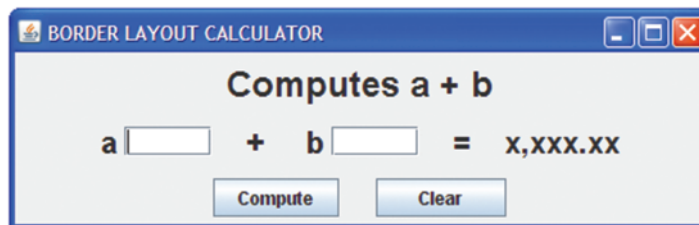
```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class BorderLayout
5  {
6      public static void main(String[] args)
7      {
8          AddingMachineGUI3 calculator = new AddingMachineGUI3("BORDER LAYOUT");
9          calculator.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         calculator.setVisible(true);
11     }
12 }

```

Figure 11.33

The application **BorderLayout**.

**Figure 11.34**

The application **BorderLayout**'s graphical user interface.

11.5.3 Flow Layout

When flow layout is assigned to a container, it is divided into rows, with all rows being the width of the window's content pane. Each row's height is set to the height of the tallest component in the row. Figure 11.35 shows a GUI built using the flow layout. Five components were added to it after setting its layout manager to `FlowLayout`:

```
setLayout(new FlowLayout());
```

Beginning with the top row, components are positioned in the rows from left to right in the order in which they are added to the container. The row height is adjusted to the height of the tallest component in it, and the components are centered within the row. When a row fills up, the next component is added to the row below it. If the window height cannot accommodate all the rows necessary to display the components, they are not shown or are partially shown. If a single component is too wide for a row, it is only partially displayed.

**Figure 11.35**

A GUI built using the **FlowLayout** manager.

The class `FlowLayoutGUI`, shown in Figure 11.36, builds the graphical interface shown in Figure 11.35 using the flow layout manager. Line 15 is needed to change the container's layout manager to flow layout because the class extends `JFrame` whose default manager is `BorderLayout`.

Following our GUI-building coding process, the atomic components are constructed in Figure 11.35 (lines 18–24), their properties are set (lines 27–29), and then they are added to the `JFrame` (lines 32–36). The application `FlowLayout` shown in Figure 11.37 declares an instance of this class on line 8 and displays the window on the monitor (line 10).

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class FlowLayoutGUI extends JFrame
5  {
6      JButton first, second, third;
7      JLabel fourth;
8      JTextField fifth;
9
10     public FlowLayoutGUI(String title)
11     {
12         super(title);
13         setSize(650, 180);
14         setLocation(200, 100);
15         setLayout(new FlowLayout()); //Override the default BorderLayout
16
17         //Step 1 create the components
18         first = new JButton("Added First");
19         second = new JButton("Added Second, Large Font Sets Row's Height");
20         third = new JButton("Added Third, Could Not Fit on Row 1");
21         fourth = new JLabel("Added Fourth, Component's Width Forces " +
22                             "Fifth Component to the Next Row");
23         fifth = new JTextField("Added Fifth, Components are Centered " +
24                                "in the Rows");
25
26         //Step 2 specify the component's properties
27         second.setFont(new Font("Sherif", Font.BOLD, 22));

```

```

28     fourth.setFont(new Font("Sherif", Font.BOLD, 16));
29     fifth.setFont(new Font("Sherif", Font.BOLD, 14));
30
31     //Step 4 add the components to the container (Step 3 is skipped)
32     add(first);
33     add(second);
34     add(third);
35     add(fourth);
36     add(fifth);
37 }
38 }

```

Figure 11.36

The class **FlowLayoutGUI**.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class FlowLayout
5  {
6      public static void main(String[] args)
7      {
8          FlowLayoutGUI flowWindow = new FlowLayoutGUI("FLOW LAYOUT");
9          flowWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         flowWindow.setVisible(true);
11     }
12 }

```

Figure 11.37

The application **FlowLayout**.

11.5.4 Grid Layout

The grid layout manager establishes a grid of cells arranged in rows and columns within a container. The number of rows and columns are specified by arguments passed to the `GridLayout` class's two-parameter constructor when the layout manager is specified:

```
setLayout(new GridLayout(4, 2)); //4 rows, 2 columns: eight cells
```

Every cell has the same height and width, which is set by the layout manager. The grid of cells always fills up the container, and the height and width of the cells are set to accommodate this. For example, a 4 x 2 grid would result in a cell height of one-quarter of the container height and a cell width of one-half the container width.

Components are added to the grid beginning with the cell in the upper left corner of the grid proceeding across a row before moving to the next row. If a component added to the grid is too large to fit into a cell, it is only partially displayed. Unused cells appear in the background color of the container.

Figure 11.38 shows a GUI interface built using a 4 x 2 grid layout. Five components were added to the `JFrame` container. The three buttons were added first, followed by a label and a text field. The label's size is larger than the size of the cells because of its large font size and the length of its text. As a result, the bottom of its text is truncated, and an ellipsis is shown to indicate that the remainder of its text could not be displayed. The cell in the lower-right corner of the grid was not used.

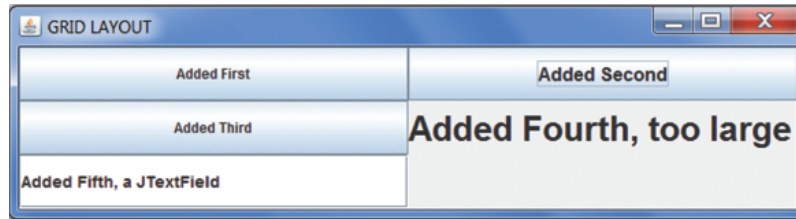


Figure 11.38

A GUI built using the `GridLayout` manager.

The class `GridLayoutGUI` shown in Figure 11.39 builds the graphical interface shown in Figure 11.38 using the grid layout manager. Line 15 changes the container's layout manager to `GridLayout` because the class extends `JFrame`, whose default manager is `BorderLayout`. Line 15 also specified the number of rows (three) and the number of columns (two) in the grid. The application `GridLayout` shown in Figure 11.40 declares an instance of this class on line 8 and displays the window on the monitor (line 10).

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class GridLayoutGUI extends JFrame
5  {
6      JButton first, second, third;
7      JLabel fourth;
8      JTextField fifth;
9
10     public GridLayoutGUI(String title)
11     {
12         super(title);
13         setSize(650, 100);
14         setLocation(200, 100);
15         setLayout(new GridLayout(3, 2));
16
17         //Step 1 create the components
18         first = new JButton("Added First");
19         second = new JButton("Added Second");
20         third = new JButton("Added Third");
21         fourth = new JLabel("Added Fourth, too large");

```



```

22     fifth = new JTextField("Added Fifth, a JTextField");
23
24     //Step 2 specify the component's properties
25     second.setFont(new Font("Sherif", Font.BOLD, 16));
26     fourth.setFont(new Font("Sherif", Font.BOLD, 30));
27     fifth.setFont(new Font("Sherif", Font.BOLD, 14));
28
29     //Step 4 add the components to the container
30     add(first);
31     add(second);
32     add(third);
33     add(fourth);
34     add(fifth);
35 }
36 }

```

Figure 11.39

The class **GridLayoutGUI**.

```

1  import javax.swing.*;
2  import java.awt.*;
3
4  public class GridLayout
5  {
6      public static void main(String[] args)
7      {
8          GridLayoutGUI gridWindow = new GridLayoutGUI ("GRID LAYOUT");
9          gridWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         gridWindow.setVisible(true);
11     }
12 }

```

Figure 11.40

The application **GridLayout**.

11.6 APPLET

The programs we have written up to this point in the textbook are called applications. Applications are intended to run from your desktop, and once launched, they run autonomously, that is, outside the scope of another program. Another type of Java program is an applet, which is a program intended to run from within another program called an *applet container*. Typically, the applet container program is a Java-enabled Web browser because Java applets are intended to perform tasks associated with Web pages that are beyond the capabilities of the language used to define Web page content: HyperText Markup Language (HTML). These tasks include handling mouse events, displaying GUI components, and performing calculations.

NOTE

HTML is a scripting language used to create Web pages and launch applets within a Web browser.

The Java Development Kit (JDK) contains an applet container program referred to as the *applet viewer*, which can also be used to run applets. It was included in the JDK to facilitate the testing of applets during their development. Applets, like applications, are developed using an integrated development environment, and most IDEs run applets from inside the JDK's applet viewer.

When we use a browser to visit a Website, the HTML document associated with the Website is downloaded to our computer from the Website's server. The browser interprets the text of the document to build and display the Web page. If the HTML document contains a reference to a Java applet, the translated bytecodes of the applet are downloaded from the Web server's disk, and then they are executed on our computer.

To prevent an executing applet from performing malicious activities, such as writing to its hard drive, on the computer on which it is running, the designers of Java restricted the range of the Java instruction set that can be included in an applet program. An attempt to execute restricted instructions contained in an applet results in a runtime error, whether is being run from within a Web browser or the applet viewer. During the development of an applet, restricted instructions that were unintentionally included in it are identified and removed during the testing phase. This Java-enforced level of security imposed on applets makes them safe to download and run from within a Web browser.

NOTE

Web browsers will download Java applets but not Java applications.

11.6.1 Developing an Applet

To create an applet, we write an applet class and an HTML document that contains a reference to the class. When the applet container (i.e., a Web browser or the applet viewer) processes the HTML document, it creates an instance of the applet class and initiates its execution by invoking several of its methods.

Applet containers can only create instances of classes that extend the class `Applet` or its child class `JApplet`, which means that all applet classes that we code must extend one of these classes. Unlike application programs that begin their execution with the first executable statement in the method `main`, applet programs begin their execution with the first executable statement in a method named `init`. When an applet is launched, the applet container invokes this method, followed by the method `start`, and then the method `paint`. The method `paint` is reinvented whenever the applet needs to be redisplayed.

Applets usually perform some of their processing inside overridden versions of these three inherited methods. Applets, like `JPanels`, are containers. They can be used to add a GUI interface to a Web page using the techniques previously discussed in this chapter or to add graphics to a Web page using the same drawing techniques used to add graphics to an application.

Figure 11.41 contains the code of the applet class `CH11HelloWebWorld` that produces the window, shown in Figure 11.42, when launched within the applet viewer container. The HTML document processed by the applet viewer is shown in Figure 11.43.

The class `CH11HelloWebWorld` is an applet because it extends the class `Applet` on line 4. It overrides its inherited `paint` method on lines 6–13. The applet container invokes this method to draw the applet, passing it a `Graphics` object in the same way as the `paintComponent` method discussed in Section 11.4.3 is invoked to draw a `JPanel`. Line 8 invokes the parent’s version of the method, passing it the `Graphics` object `g`. This should always be the first executable statement in the overridden version of the method. The object `g`, passed to the method by the applet container, is used on lines 9–12 to perform the applet’s graphical output.

```

1  import java.awt.*;
2  import java.applet.*;
3
4  public class CH11HelloWebWorld extends Applet
5  {
6      public void paint(Graphics g)
7      {
8          super.paint(g);
9          g.setFont(new Font("Sherif", Font.BOLD, 16));
10         g.drawString("Hello Web World", 170, 130 );
11         g.setColor(Color.BLUE);
12         g.fillOval(200, 140, 70, 70);
13     }
14 }
```

Figure 11.41

The applet `CH11HelloWebWorld`.



Figure 11.42

The output produced by the applet `CH11HelloWebWorld` running in the applet viewer.

```

1  <html>
2      <title>
3          Hello Applet
4      </title>
5      <body>
6          <applet code = "CH11HelloWebWorld.class"
7                          width = "500"
8                          height = "300">
9      </applet>
10     </body>
11 </html>

```

Figure 11.43

The HTML document that launches the applet **CH11HelloWebWorld**.

11.6.2 HTML Document Basics

The HTML document shown in Figure 11.43 would be used by an applet container to download and launch the applet shown in Figure 11.41. Like all HTML documents, it consists of instructions, called *elements*, describing how to display a Web page. The elements are enclosed within *tags*, which are enclosed in angle brackets and come in pairs. Pairs end with the same characters, which are part of the HTML scripting language. The second tag in a pair begins with a forward slash indicating the end of an instruction or section; the first tag in a pair does not begin with a slash. For example, the tags `<html>` and `</html>` that appear on lines 1 and 11 of Figure 11.43 are a pair. All HTML documents begin with the first tag of this pair and end with the pair's second tag. The remainder of the document in Figure 11.43 consists of a title element (lines 2–4) followed by a body element (lines 5–10) that contains an applet element (lines 6–9).

The title element contains the text of the Web page's title that will be displayed in a Web browser's window, in this case, *Hello Applet*. When the applet is launched from the applet viewer, the title is not displayed. The body element is meant to contain text and other HTML elements that will be displayed on the Web page. The body of the document shown in Figure 11.43 simply contains an applet element that references the class of our applet and sets the size of the applet when it is displayed by an applet container.

The first tag of the applet element contains the name of the applet's bytecode file and its extension. This is followed by the designation of the width and height of the applet: 500 and 300 on lines 7 and 8. The closing angle bracket for the first tag of the applet element is at the end of line 8. If the path name is not specified in front of the applet's file name, the file is assumed to be in the same folder as the HTML document. The tag on line 9 ends the applet element.

The generalized format of an applet element is shown below. The highlighted portions of it are application dependent:

```

<applet code = "CH11HelloWebWorld.class" width = "500" height = "300">
</applet>

```

The applet can be centered within the Web page by placing it inside center tags:

```
<body>
  <center>
    <applet code = "CH11HelloWebWorld.class"
           width = "500"
           height = "300">
  </applet>
</center>
</body>
```

The use of indentation and new lines in the HTML documents is simply to improve the document's readability.

In addition to the applet element, text to be displayed on the Web page by including it in the body element. There are several HTML tags that can be embedded in the text to control its formatting (e.g., its size, position, font style) the details of which are beyond the scope of this book.

11.6.3 The Applet Execution Path

The execution path of an applet is dictated by a protocol that is used by applet containers to launch, redisplay, and close an applet. The progression of steps contained in this protocol is often referred to as an applet's *life cycle*. The protocol includes a definition of the signatures of an applet's `init`, `start`, `paint`, `stop`, and `destroy` methods in the Java API classes `Applet` and `Container`, and a definition of when the applet container will invoke these methods. Table 11.8 shows the signature of these methods and their role in the life cycle of an applet. Overriding these methods within an applet class, to perform processing consistent with their intended use, produces a properly functioning applet.

Table 11.8

Methods Invoked by Applet Containers

When Invoked by the Applet Container	Intended Use
<code>public void init()</code>	
Invoked once: when the applet is launched	Perform initialization tasks normally relegated to a constructor, such as declaring and initializing class level variables, building the applet's GUI, registering event handlers, and creating threads (see Chapter 14)
<code>public void start()</code>	
Invoked after the <code>init</code> method and each time the Web page is revisited	Perform tasks that are associated with revisiting the Web page containing the applet, such as starting/restarting animation timers

(Contd.)

When Invoked by the Applet Container	Intended Use
<code>public void paint()</code>	
Invoked after the <code>init</code> and <code>start</code> methods and whenever the applet needs to be redisplayed, for example, when the applet container is minimized and then maximized	Repaint the applet; perform 2D graphics in applet classes that extend the class <code>Applet</code>
<code>public void stop()</code>	
Invoked when the Web page is left and before the <code>destroy</code> method is invoked	Perform tasks that are associated with leaving the Web page containing the applet, such as stopping animation timers
<code>public void destroy()</code>	
Invoked once when the applet is removed from memory, e.g., closed	Deallocate resources allocated to the applet

11.6.4 Incorporating GUIs and Two-Dimensional Graphics into Applets

Applets are containers. The `Applet` and `JApplet` classes both have the class `Container` in their inheritance chain. This gives them the ability to contain GUI components, process events, and draw 2D shapes on GUI interfaces using the same techniques and syntax discussed previously in this chapter to incorporate these features into `JPanels` and `JFrames`. An applet's default layout manager is `BorderLayout`.

Two-Dimensional Graphics in a `JApplet`

In Figure 11.41, we coded an applet to draw graphical text and a 2D shape (a filled oval) on a Web page. The applet's class extended the class `Applet` and performed its drawing inside an overridden version of the `paint` method. If the class extended the class `JApplet`, another approach to performing 2D graphics would be taken. The drawing would be performed inside a class that extends `JPanel` using an overridden version of the `paintComponent` method, and an instance of that class would be added to the applet. If this is not done, the graphics are not redrawn as part of the redrawing of the Web page.

The applet `HelloWebWorldV2`, shown in Figure 11.44, presents a revised version of the `HelloWebWorld` applet shown in Figure 11.41. This version of the applet extends the class `JApplet` and performs its 2D graphics by declaring an instance of the class `GraphicsPanel` (lines 6 and 10), shown in Figure 11.45, and adding it to the applet (line 11 in Figure 11.44). The `GraphicsPanel` class extends `JPanel`, overrides its `PaintComponent` method on lines 6–13, and performs the drawing inside of it.

The HTML document processed by the applet container to generate the Web page shown in Figure 11.46 is the same as the document shown in Figure 11.43, except that line 3 would be changed to `Hello Applet V2`, and line 6 has to be changed to:

```
<applet code = "CH11HelloWebWorldV2.class">
```

As indicated in Table 11.8, the overridden version of the `init` method shown in Figure 11.44 performs tasks normally relegated to constructors in application programs.

```

1  import java.awt.*;
2  import javax.swing.*;
3
4  public class CH11HelloWebWorldV2 extends JApplet
5  {
6      GraphicsPanel aPanel;
7
8      public void init()
9      {
10         aPanel = new GraphicsPanel();
11         add(aPanel);
12     }
13 }

```

Figure 11.44

The applet `CH11HelloWebWorldV2`.

```

1  import java.awt.*;
2  import javax.swing.*;
3
4  public class GraphicsPanel extends JPanel
5  {
6      public void paintComponent(Graphics g)
7      {
8          super.paintComponent(g);
9          g.setFont(new Font("Sherif", Font.BOLD, 16));
10         g.drawString("Hello Web World", 170, 130);
11         g.setColor(Color.BLUE);
12         g.fillOval(200, 140, 70, 70);
13     }
14 }

```

Figure 11.45

The class `GraphicsPanel`.

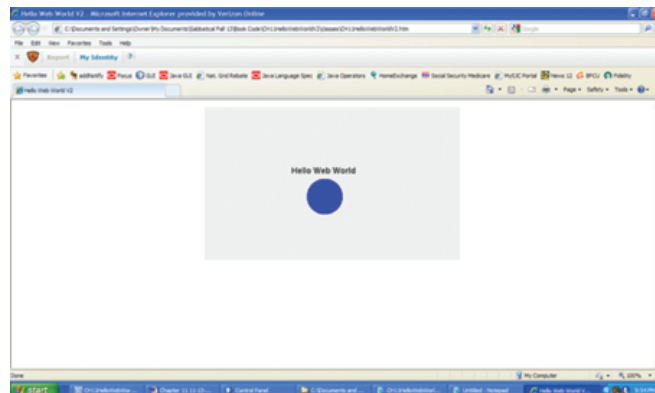


Figure 11.46

The Web page containing the applet `HelloWebWorldV2`.

GUI Components and Event Handling

A GUI interface for a Web page is built using the same techniques used to build a GUI interface for an application, which were previously discussed in this chapter, with one exception. Instead of building the interface and coding and registering the event handlers inside of a class that extends `JPanel` or `JFrame`, this processing is done in an applet class that becomes an element in the Web page's HTML document.

Tasks that would have been performed by a constructor when implementing a GUI class for an application, such as declaring GUI components, adding them to the interface, registering event handler methods, and specifying a layout manager are performed inside the applet's `init` method. GUI components can be positioned in the interface using the applet's default layout manager `BorderLayout`, or the layout manager can be changed using the `setLayout` method to `FlowLayout`, `GridLayout`, or set to null to use the `setSize`, `setLocation`, and `setBounds` methods to position the components. As is the case with applications, event handlers are coded as methods registered in listener lists, but now their code is placed inside the applet class or within inner classes added to it.

An example of building a graphical interface for a Web page is illustrated in Figure 11.47; it presents the applet `GuessingGame`, which builds the interface and implements the game. When the applet is launched its interface, shown in Figure 11.48, appears on the Web page.

Initially, the numbers 1 through 8 are displayed on a pair of buttons randomly selected from the 16-button grid positioned at the top of the applet's GUI. After studying the arrangement of the numbers, the player clicks the *Begin* button, and the numbers are hidden. The object of the game is to reveal all of the number pairs with a minimum number of clicks. To accomplish this, the player repeatedly attempts to click a pair of buttons whose numbers match. When a correct match is clicked, the number pair remains visible. When an incorrect match is clicked, the number pair remains visible for two seconds, an error count is incremented, and then numbers are hidden. The game continues until all the pairs of numbers have been revealed.

The applet consists of four major sections: the class-level variable declarations (lines 9–22), the `init` method (lines 24–55), three worker methods (lines 57–93), and the event handlers (lines 96–169).

The `init` method (lines 24–55), invoked by the applet container program to launch the applet, is used to build the GUI and initialize the game. A two-second (2,000 millisecond) timer is declared on line 22, and its second argument adds the timer event handler coded on lines 158–169 to its listener list. Line 28 changes the applet's default `BorderLayout` to a five-row by four-column `GridLayout`. Lines 17 and 26 create a `JButton` array named *cell*. Each time through the `for` loop that begins on line 40, a new button is created (line 42), the click event handler method defined on lines 114–156 is added to its listener list (line 44), and the button is added to a cell of the GUI's grid (line 45).

Lines 49–52 build the bottom row of the grid by adding two buttons, *begin* and *reset*, and two labels, *errors* and *numberOfErrors*. The `actionPerformed` event handler method de-

lined on lines 98–111 is added to these buttons' listener lists on lines 36–37. Line 31 invokes the `setHorizontalAlignment` method passing it the static constant `RIGHT` to position the `errors` and `numberOfErrors` labels, which were added to the bottom row of the grid, next to each other. Finally, line 54 of the `init` method invokes the worker method `initializeGame` (lines 57–67), to generate (line 59) and display (line 64) a randomly placed set of numbers on the 16 button grid.

The `generateNumbers` method, invoked on line 59 of the `initializeGame` method and on line 109 when the *Reset* button is clicked, generates and places the eight number pairs into the array `values` (lines 76–79), and then they are randomly swapped within the array (lines 80–84). The loop that begins on line 62 of the `initializeGame` method sets each button's text to one of these 16 randomized values by passing their `setText` method the string version of an element of the array (line 64). Line 64 uses the arrays `cell` and `values` as parallel arrays, as do the loops that process a pair of button clicks (lines 122 and 137).

```

1  import java.awt.*;
2  import java.applet.*;
3  import javax.swing.*;
4  import java.util.Random;
5  import java.awt.event.*;
6
7  public class GuessingGame extends JApplet implements ActionListener
8  {
9      boolean firstClick = true;
10     int firstClickValue = 0;
11     int firstClickIndex = 0;
12     int secondClickIndex = 0;
13     int errorCount = 0;
14     boolean correct = false;
15     int[] values = new int[16];
16     JButton b = new JButton();
17     JButton[] cell;
18     JLabel errors = new JLabel("Errors: ");
19     JLabel numberOfErrors = new JLabel("0");
20     JButton begin = new JButton("Begin");
21     JButton reset = new JButton("Reset");
22     Timer timer1 = new Timer(2000, new TimerHandler());
23
24     public void init()
25     {
26         cell = new JButton[16]; //the number buttons
27
28         setLayout(new GridLayout(5, 4)); //override default BorderLayout
29
30         //set properties of GUI components
31         errors.setHorizontalAlignment(JLabel.RIGHT);
32         errors.setFont(new Font("Serif", Font.BOLD, 30));
33         numberOfErrors.setFont(new Font("Serif", Font.BOLD, 30));
34

```

```

35     //add event handlers to listener lists
36     begin.addActionListener(new BeginResetHandler());
37     reset.addActionListener(new BeginResetHandler());
38
39     //create number buttons, set properties, register event handlers
40     for(int i = 0; i < 16; i++)
41     {
42         cell[i] = new JButton("0");
43         cell[i].setFont(new Font("Serif", Font.BOLD, 40));
44         cell[i].addActionListener(this);
45         add(cell[i]);
46     }
47
48     //add the lower buttons and labels
49     add(begin);
50     add(errors);
51     add(numberOfErrors);
52     add(reset);
53
54     intializeGame();
55 }
56
57 public void intializeGame()
58 {
59     generateNumbers();
60     numberOfErrors.setText("0");
61     errorCount = 0;
62     for(int i = 0; i < 16; i++)
63     {
64         cell[i].setText(Integer.toString(values[i]));
65         cell[i].setForeground(Color.BLACK);
66     }
67 }
68
69 public void generateNumbers() //generate buttons' numbers
70 {
71     Random rn = new Random();
72     int number = 0;
73     int cellNumber = 0;
74     boolean done;
75
76     for(int i = 1; i <= 16; i++) //all number buttons
77     {
78         values[i-1] = i % 8 + 1; //place numbers 1->8 twice
79     }
80     for(int i = 0; i < 16; i++) //all number buttons
81     {
82         number = rn.nextInt(15);
83         swap(values, i, number); //swap button value with a random button

```

```

84     }
85 }
86
87 public void swap(int[] array, int indx1, int indx2)
88 {
89     int temp;
90     temp = array[indx1];
91     array[indx1] = array[indx2];
92     array[indx2] = temp;
93 }
94
95 //event handlers *****
96 public class BeginResetHandler implements ActionListener
97 {
98     public void actionPerformed(ActionEvent e)
99     {
100         if(e.getSource() == begin) // start the game
101         {
102             for(int i = 0; i < 16; i++)
103             {
104                 cell[i].setText(" "); //hide the numbers
105             }
106         }
107         else //generate and a new game
108         {
109             intializeGame();
110         }
111     }
112 }
113
114 public void actionPerformed(ActionEvent e) //number buttons' handler
115 {
116     if(firstClick) //show the number
117     {
118         for(int i = 0; i<16; i++) //all number buttons
119         {
120             if(e.getSource() == cell[i]) //button clicked found
121             {
122                 cell[i].setText(Integer.toString(values[i]));
123                 firstClick = false;
124                 firstClickValue = values[i];
125                 firstClickIndex = i;
126                 break;
127             }
128         }
129     }
130     else //second click processing
131     {

```

```

132     timer1.start(); //two seconds
133     for(int i = 0; i<16; i++) //all number buttons
134     {
135         if(e.getSource() == cell[i]) //button clicked found
136         {
137             cell[i].setText(Integer.toString(values[i]));
138             firstClick = true;
139             secondClickIndex = i;
140             if(firstClickValue == values[i]) //correct match
141             {
142                 correct = true;
143                 cell[firstClickIndex].setForeground(Color.BLUE);
144                 cell[secondClickIndex].setForeground(Color.BLUE);
145             }
146             else //incorrect match
147             {
148                 correct = false;
149                 errorCount++;
150                 numberOfErrors.setText(Integer.toString(errorCount));
151             }
152             break;
153         } // end if
154     } // end for
155 } // end else
156 }
157
158 public class TimerHandler implements ActionListener
159 {
160     public void actionPerformed(ActionEvent e) //Timer's event handler
161     {
162         if(correct == false) //no match
163         {
164             timer1.stop(); //after a two second pause
165             cell[firstClickIndex].setText(" "); //hide the numbers
166             cell[secondClickIndex].setText(" ");
167         }
168     }
169 }
170 }

```

Figure 11.47The applet **GuessingGame**.

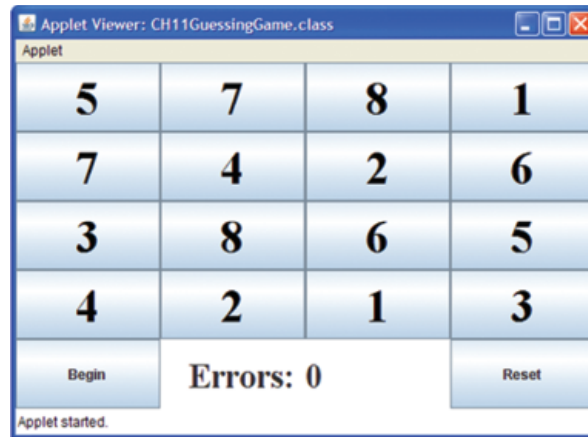


Figure 11.48
The GUI produced by the applet `GuessingGame`.

11.6.5 Portability and Security Issues

When designing a GUI interface for an applet, the highest level of portability across Web browsers is achieved when AWT components, rather than Swing components, are used in the interface. Alternately, the portability of Swing-based GUI applets can be extended across the range of available browsers by installing a Java plugin on the computer on which the browser is running. The plugin is freely available and is part of the JDK.

Although AWT GUI components used in applets offer more browser portability, Swing provides a richer complement of components. The more commonly used Swing components, such as labels, buttons, text fields, and check boxes do have AWT counterparts, but several Swing components (e.g., radio buttons) do not have an AWT counterpart. The AWT component class names do not include the leading `J` used in the Swing class names. For example, the name of the AWT button class is `Button`, and the name of the AWT label class is `Label` (Swing's class names are `JButton` and `JLabel`).

In addition to the difference in the names of the component classes, the names of some of the Swing and AWT methods used to perform common operations on GUI components are also different. For example, the Swing methods `setText` and `getText` can be used to change and fetch the annotation on a `JButton`. Their AWT counterparts are the methods `setLabel` and `getLabel`.

The applet shown in Figure 11.47 extends the class `JApplet` on line 7, and its graphical interface is built entirely with Swing components. The applet's portability can be extended by making it a child of the `Applet` class (extending `Applet` instead of `JApplet`), eliminating the `J` that begins all of the GUI component class names used in the applet, and changing the `setText` method invocations (e.g., on line 64) to invocations of the AWT `Button` class's `setLabel` method.

Often, it is desirable to allow an applet to perform certain operations that are beyond the range of those performed by the default Java instruction subset that can be included in an applet, such

as installing Web-based software updates to a program stored on in a computer's hard drive. To accomplish this, applet developers obtain a digital security certificate from a certificate authority organization, which is attached to the applet. Before a certified applet is run inside a Web browser, information about the applet's developer and the restricted operation it will perform is announced to the user, and the user is asked if the developer is a trusted source. An acknowledgement of trust in the developer permits the applet to run and gain access to various resources of the user's computer, such as its hard drive, that would normally be restricted.

11.7 CHAPTER SUMMARY

The overloaded versions of the `showInputDialog` and `showMessageDialog` methods in the `JOptionPane` class can be used to display more informative and user-friendly dialog boxes than those versions used in previous chapters. A dialog box's default icon can be replaced with one of four other selections, its default title can be changed, and the window within which it is centered can be specified. In addition, a default input or a set of valid inputs from which to choose can be displayed in an input dialog box.

User-friendly graphical interfaces are incorporated into a program by adding a worker class that contains the code to build the interface. The program declares an instance of the class and makes it visible. Ordinarily, the worker class either extends `JFrame` for non-Web-based programs or the `JApplet` for Web-based programs. These API classes are referred to as top-level containers. The worker class contains code to create and add instances of `JButton`, `JTextField`, and `JLabels` to a container after setting their visibility, annotation, color, font, and size properties and their location within the container. An instance of the class `Timer` can be added to a container. Its time interval is set when it is created, and it is initiated by invoking the `start` method on the timer object.

A flow layout manager can be used to facilitate the positioning and sizing of components within a container. A container's default `FlowLayout` manager can be changed by invoking the `setLayout` method on the container or nullified by passing the method a `null` value. Some integrated development environments have a GUI-builder feature that generates the worker class as the programmer drags and drops GUI components onto a container and selects their properties from lists and dialog boxes.

The GUI-builder worker class also contains code to perform processing as the user interacts with the interface or when a timer's interval expires. These interactions are called events, and the code is placed inside call back methods referred to as event handlers. The signatures of these methods are defined in a group of API interfaces, three of which are named `ActionListener`, `KeyListener`, and `MouseMotionListener`. The methods defined in these three interfaces are invoked when a button is clicked, a key is typed, the mouse is clicked or dragged, or a time interval expires.

Event handler methods are associated with particular components in the GUI by registering them with the component via an invocation of methods such as `addActionListener`, `addKeyListener`, and `addMouseMotionListener` passing them an instance of the class in which the

event handler method is coded. In the case of a `Timer` object, the instance of the event handler's class is passed to the `Timer` object's constructor when it is created. The event handlers can be part of the GUI-builder worker class or an inner class defined within it. The heading of a class that contains the implementation of the event handler method must contain an `implements` clause indicating that it implements the method's interface. Alternately, the class can extend one of the API adapter classes that provide empty implementations of some of the API listener interfaces. The component on which an event occurred can be identified by invoking the `getSource` method on the argument passed to the event handler method, or the key struck on a keyboard can be determined by invoking the `getKeyCode` method on the argument.

A paint event is any event that causes a graphical object to be drawn or redrawn, such as maximizing a window after it has been minimized. When these events occur, a call back method is invoked by the Java Runtime Environment, which can be overridden to draw shapes on a GUI. These method names begin with the word “paint” (for example, `paintComponent`) and are ordinarily overridden in a separate drawing worker class. Then, an instance of the class is declared within the program (e.g., the method `main`), and the `add` method is invoked on the program's GUI-builder object (its window) to add the instance

In addition to application programs, Java also supports applets, which run from within another program, such as a Web browser or applet viewer. Applet containers create instances of classes that extend the class `Applet` or its child class `JApplet`. Unlike application programs that begin their execution with the first executable statement in the method `main`, applet programs begin their execution with the first executable statement in a method named `init`. The applet container invokes this method when it launches the applet, followed by invocations of the methods `start` and `paint`. It invokes the method `stop` to suspend the applet and `destroy` to terminate the applet. Applets like `JPanels` are containers, and they can be used to add a GUI interface or 2D shapes to a Web page using the same techniques used to add a GUI and shapes to applications.

Applets are considered to be both portable and secure. The highest level of portability across Web browsers is achieved when AWT components, rather than Swing components are used. Security is provided for Java applets by restricting some of the functions they can perform and by attaching a digital security certificate to the applet.

Knowledge Exercises

1. True or false:
 - a) GUI stands for Grand Unified Interaction.
 - b) Although they take longer to develop, GUIs reduce the time and effort needed to interact with a program.
 - c) To create a graphical user interface, declare an instance of a top-level container class object and add the GUI components to it.
 - d) A paint event causes a graphical object to be drawn or redrawn.
 - e) Check boxes permit multiple selections to be made.
 - f) Multiple components in an application can have the keyboard's focus at the same time.

- g) Java applications and applets both begin executing in their `main` method.
 - h) An enhanced dialog box can be used to display several valid inputs from which a user can choose.
 - i) The default layout manager for `JFrames` is the grid layout manager.
 - j) The border layout manager specifies five regions for the components.
 - k) The north and south regions of the border layout are always the width of the window.
 - l) All the cells of the grid layout have the same height and width.
 - m) Java applets are considered to be both portable and secure.
2. Give a statement that asks a person for his or her age using a dialog box containing an information message icon and the title *Happy Birthday*.
 3. Name at least three GUI components and explain their usual uses.
 4. Give the code to:
 - a) Make the `JButton` `b1` disappear
 - b) Relocate `JButton` `b2` to location (200, 400)
 - c) Change the text displayed in `JTextBox` `tb1` to *Correct*
 - d) Create a `JButton` named `b3` whose annotation is *Click Here*
 - e) Change the text of `JLabel` `lb1` to *The Answer is Yes*
 - f) Add the `JButton` `b4` to the `JPanel` `p1`
 - g) Make the size of `JTextBox` `tb2` 100 pixels wide and 50 pixels high
 - h) Attach the tool tip *Click After Entering a Number* to `JButton` `b5`
 5. Name the five regions of the border layout and describe their location, height and width.
 6. Name and describe the three layout managers that Java provides.
 7. Give the code to:
 - a) Allow components to be positioned and sized in the `JPanel` `p1` using the `setBounds` method
 - b) Change a `JPanel` `p2`'s layout manager to `BorderLayout`
 8. Sketch the position of five components (`c1`, `c2`, `c3`, `c4`, and `c5`) as each layout manager would present them after they were added to a container in the order `c1` through `c5`. Clearly state your assumptions, where necessary, regarding the width and placement of the components.
 9. Give the code to display a 600 x 800-pixel red window from within the method `main` without the use of a GUI-builder worker class.
 10. What is the result of omitting the `setDefaultCloseOperation` when creating a window?
 11. Explain the purpose of event handlers.
 12. Give the name of the event handler that processes a click on a `JButton` object.
 13. Three `JButton` objects named `b1`, `b2`, and `b3` are registered with the same event handler. Give the code in the event handler method to output *Button 2* when `b2` is clicked.

14. What are the three keyboard handler methods defined in the API `KeyListener` interface, and what events do they handle?
15. What are adapter classes, and what is the advantage of a class extending them?
16. What does it mean to say that a component must have the keyboard's focus?
17. How can a component get the keyboard's focus?
18. What are some of the differences between the AWT and Swing GUI packages?
19. State one advantage of using the classes in the AWT package over the classes in the Swing package, and vice versa.
20. Discuss the major difference between Java applications and applets.
21. How do Java applets provide security and portability?

Programming Exercises

1. Write a program with a GUI that allows the user to input the length and width of a rectangle. The GUI will have three buttons. When one of the buttons is clicked, the area of the rectangle is displayed in square feet. When the second button is clicked, the perimeter of the rectangle is displayed. When the third button is clicked, the GUI is restored to the condition it was in when the program was launched. Each button will have its own event handler.
2. Modify the program in Exercise 1 so one event handler performs the calculations, and the restoration for the GUI to its launch condition is performed by a second event handler.
3. Modify the program in Exercise 2 to include an additional input for the cost of carpet per square foot and a fourth button to calculate and display the cost of the carpet.
4. Write a GUI application whose window's title is *Favorite Color* and is initially blue. When a button on the interface is clicked, an enhanced dialog box is displayed from which the user can choose a color from among eight colors. When the OK button on the dialog box is clicked, the background of the window should change to the selected color.
5. Create a program to display a GUI window with the following features: its size is 400 x 450 pixels, and its location is 100, 100. The window title is *Surprise!* The window should contain a button labeled *Press Here*. When the button is pressed, the background color should change to your favorite color and a circle (or a smiley face) should appear.
6. Write a program using a GUI with three buttons, labeled *RED*, *YELLOW* and *BLUE*, which cause a filled rectangle to be drawn in the appropriate color when pressed.
7. Write a GUI application that will react to a mouse-click event by displaying the x and y coordinates of the position in the window where the mouse was clicked.
8. Create a GUI window with a mouse and a piece of cheese drawn on it, each at a random location. The keyboard directional arrow keys can be used to move the mouse to the piece of cheese.

9. Expand the application described in Exercise 8 so the cheese can be dragged around the window to the mouse.
10. Write a GUI application called `StopWatch` that displays the minutes and seconds that have elapsed since the interface's Start button was clicked. When the interface's Reset button is clicked, the elapsed time should return to zero, and the timer should stop. Each button should have its own event handler.
11. Design and implement a four-function GUI calculator with buttons for the digits from 0 to 9 and the arithmetic operations +, -, *, and /. The button for = should cause the operation to be performed and the result to be displayed in a text field. Also, include a Clear button to clear the calculator so another operation can be performed.
12. Write an applet named *Know Your Shapes* that displays a colorful circle, square, rectangle, and ellipse, each with a text field below it. After the user types the names of each shape in the text boxes and clicks the Done button, display *Correct* in the text fields in which the typed names are correct and the correct name of the shape in the text boxes in which the typed name is incorrect. After five seconds have elapsed, clear the text boxes and output *Try again* to the GUI.

Enrichment

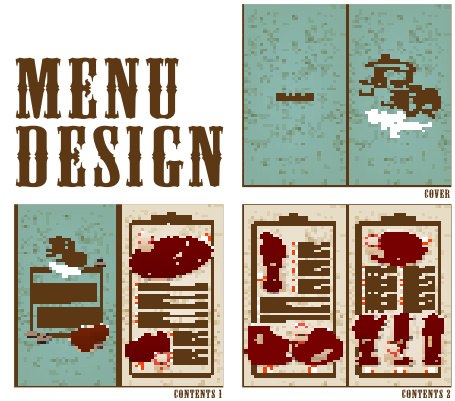
Use Java's `Timer` object to create animation for a graphical object on a GUI interface.

References

Boese, Elizabeth Sugar. *An Introduction to Programming with Java Applets*. 3rd Ed. Sudbury, MA: Jones and Bartlett Publishers, 2010.

GRAPHICAL USER INTERFACES: A SECOND LOOK

12.1	<i>Borders Checkboxes and Radio Buttons</i>	550
12.2	<i>Combo Boxes and Lists</i>	563
12.3	<i>Menus</i>	572
12.4	<i>File Chooser and Color Chooser Dialog Boxes</i>	585
12.5	<i>Chapter Summary</i>	590



In this chapter

In Chapter 11, we became familiar with the techniques used to create a GUI application's program window and add labels, buttons, and text fields to it, and how to respond to the user's interaction with these components via key strokes or mouse clicks and drags. In this chapter, we expand our knowledge of the other GUI component classes available in the API Swing package. Check boxes, radio buttons, combo boxes, and lists allow a user to select one or more inputs from a set of valid inputs. The procedure for adding a component to a window is expanded to include the grouping of radio buttons and the placement of titled borders around GUI components.

Menus are a common component of GUIs, and both drop-down and pop-up menus are discussed in this chapter, as are submenus and hot keys. In addition, the file-chooser and color-chooser dialog boxes used to facilitate disk I/O and color selection are presented in this chapter.

After successfully completing this chapter, you should:

- Be able to create and position check boxes, radio buttons, combo boxes, and lists and perform processing when the user selects inputs associated with these components
- Know how to enclose GUI components within titled borders and how to change the color, style, and thickness of borders
- Understand how to add scroll bars to combo boxes and lists
- Be able to assign hot keys to GUI components and perform processing in response to hot-key strokes
- Understand how to implement drop-down and pop-up menus and submenus and perform processing in response to menu selections
- Know how to use API defined dialog boxes to facilitate the input of file I/O paths, file names, and color selection

12.1 BORDERS CHECKBOXES AND RADIO BUTTONS

The GUI components check box and radio button are shown in the upper center and upper right portion of Figure 12.1. Groupings of check boxes and radio buttons are used to facilitate the selection of one or more inputs from a small set of valid inputs. When the user can select several of the inputs from the set, check boxes are used in the GUI. When the inputs are mutually exclusive, that is, only one of the valid inputs can be selected, radio buttons are used.

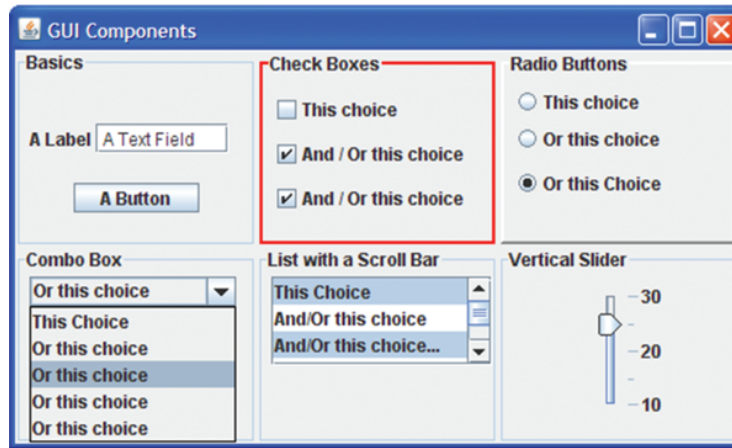


Figure 12.1
GUI components.

Ordinarily, a set of check boxes or a set of radio buttons is added to an instance of a `JPanel`, and then the panel is added to the window's content pane. This makes the check box set or radio button set easier to reposition in the window because only the panel's location needs to be changed, rather than changing the position of each of the boxes or buttons individually. In addition, the panel's border can be made visible to give the visual impression that the check boxes or buttons are part of a set, and a title can be added to the panel's border to provide additional information about the box or button set. This is the approach that was taken when the GUI illustrated in Figure 12.1 was built.

12.1.1 Borders

A border can be placed around any component that extends the class `JComponent`, although most often, the use of a border is associated with `JPanel` and `JLabel` components. This is done by invoking the `setBorder` method on the component and passing it the object returned from an invocation of one of the static methods in the `BorderFactory` class. These static methods create most of the borders available in the API Swing package. The following code fragment was used to place the border containing the title *Basics* around the `JPanel` component `p1` that contains the label, text field, and button on the top left side of Figure 12.1.

```
//Default light blue line border with a title
p1 = new JPanel();
p1.setBorder(BorderFactory.createTitledBorder("Basics"));
```

This one-parameter version of the `createTitledBorder` method creates a border drawn as a thin light blue line with the title displayed in the default position (i.e., the upper left corner of the border). The static method `createTitledBorder` is overloaded. All of the other versions of the method contain a `Border` type parameter, which is used to change the thickness, style, and color of the border, and some versions of the method contain parameters to change the title's vertical position, horizontal justification, and its font color and style.

In addition to the `createTitledBorder` method, the `BorderFactory` class contains static methods used to construct different styles of borders and specify the thickness and color of the border. These methods construct and return a `Border` object that describes the border, which is then passed to the overloaded versions of the `createTitledBorder` method. The following code fragment was used to create and place the thick red-colored line border around the `JPanel` that encloses the check boxes in Figure 12.1. The integer argument, 2, passed to the `BorderFactory` class's method `createLineBorder` changes the border thickness from the default value of 1 to 2:

```
//Double thick red colored line border with a title
p2 = new JPanel();
Border aBorder = BorderFactory.createLineBorder(Color.RED, 2);
p2.setBorder(BorderFactory.createTitledBorder(aBorder, "Check Boxes"));
```

Other border styles include an etched style, a beveled style, and a soft-beveled style. The `BorderFactory` class's `createBevelBorder` is used to change the style of the border from a line to a beveled appearance. The following code fragment was used to place the beveled border around the `JPanel` that encloses the radio buttons in Figure 13.1. The integer argument (0) passed to the `createBevelBorder` method specifies the type of bevel to use, in this case, raised:

```
//Raised beveled border with a title
p3 = new JPanel();
aBorder = BorderFactory.createBevelBorder(0);
p3.setBorder(BorderFactory.createTitledBorder(aBorder, "Radio Buttons"));
```

Borders can be created without titles by only passing the `setBorder` method a `Border` object returned from one of the `BorderFactory` class's static methods. The following code fragment places a raised beveled border with no embedded title around a `JPanel`:

```
//Double thick red colored line border, no title
p4 = new JPanel();
p4.setLayout(null);
p4.setBorder(BorderFactory.createBevelBorder(0));
```

12.1.2 Check Boxes

A grouping of check boxes is used on a GUI to facilitate the selection of one or more inputs from a small set of valid inputs. When the user clicks a check box, a check either appears in the check box or is removed from it. Using the techniques discussed in this section, check boxes are created, added to, and positioned in GUI containers. Processing is initiated based on their checked or unchecked status.

Creating Check Boxes

Check boxes are instances of the API class `JCheckBox` and can be created using the class's one (`String`) parameter constructor or the class's default constructor. The text that appears beside the check box when it is displayed is the string passed to the one-parameter constructor or the string passed to the class's `setText` method. The following code fragment creates the first two check boxes shown in the top center of Figure 12.1:

```
//Create check boxes and initialize their text.
JCheckBox cb1 = new JCheckBox("This choice");
JCheckBox cb2 = new JCheckBox();
cb2.setText("And / Or this choice");
```

The `setText` method sets the text property of most GUI components that display text, and it can be used to initially set or to change the text associated with a component. As discussed in Chapter 11, the properties of atomic components and containers can be set and fetched using the methods shown in Table 11.5, which is recreated as Table 12.1 for convenience.

Table 12.1
Methods Used to Specify a Component's Properties and Add it to a Container

Method Signature	Description
JComponent and Component Class Methods Invoked on Components	
<code>setToolTipText(String tip)</code>	Adds the tool tip <code>tip</code> to the component, displayed when the mouse pointer hovers over it
<code>setBounds(int x, int y, int width, int height)</code>	Sets the component's location to (<code>x</code> , <code>y</code>) and its width and height to <code>width</code> and <code>height</code>
<code>setLocation(int x, int y)</code>	Sets the component's location to (<code>x</code> , <code>y</code>)
<code>setSize(int width, int height)</code>	Sets the component's width and height to <code>width</code> and <code>height</code>
<code>setText(String newText)</code>	Changes the text displayed on the component to <code>newText</code>
<code>setVisible(boolean visible)</code>	The component is visible when <code>visible</code> is passed the value <code>true</code> , invisible when passed <code>false</code>
<code>setFont(Font fontStyle)</code>	Sets the font style of the container or component that invoked the method to <code>fontStyle</code>
Container Class Methods	
<code>setLayout(LayoutManager layout);</code>	Sets the container's layout to <code>layout</code> , to specify location/size of components: <code>layout = null</code>
<code>add(Component theComponent)</code>	Adds <code>theComponent</code> to the container or component that invoked the method

By default, a check box is initially displayed unchecked (without a check mark). An additional Boolean argument can be passed to the `JCheckBox` class's one-parameter constructor to specify that the box will contain a check mark when it is initially displayed:

```
//Display a check in a check box
JCheckBox cb1 = new JCheckBox("Check box is checked", true);
```

Adding Check Boxes to Containers and Positioning Them

Check boxes are added to a GUI container using the `add` method, which is described at the bottom of Table 12.1. The following code fragment adds two check boxes to the `JPanel` container `p1`:

```
//Add two check boxes to a JPanel container
p1 = new JPanel();
JCheckBox cb1 = new JCheckBox("Hamburger");
JCheckBox cb2 = new JCheckBox("Taco");
p1.add(cb1);
p1.add(cb2)
```

Check boxes, like other components added to a container, are positioned within it using the techniques discussed in Chapter 11 (Section 11.3.3 when a layout manager is not used and Section 11.5 when a layout manager is used). When a layout manager is used, the check boxes will be positioned by the manager in the order in which they are added to the container.

If the container's layout manager has been set to `null`, the `setBounds` method described in Table 12.1 can be used to position and set the height and width of the component. The height and width passed to the method includes the box and its associated text. Alternately, the `setLocation` and `setSize` methods can be used to position a check box in the container and to specify its size.

The following code fragment was used to create the three check boxes shown in Figure 12.1 and the panel that contains them. The panel does not use a layout manager to position its components because the panel's `setLayout` method is passed a `null` value. This permits the use of the `setBounds` method to position and size the components added to it. The (x, y) position specified by the first two arguments sent to the method `setBounds` is relative to the upper left corner of the container (to which a component has been added).

```
// Position and size check boxes without using a layout manager
p2 = new JPanel();
p2.setLayout(null); //the default border manager is not used
Border aBorder = BorderFactory.createLineBorder(Color.RED, 2);
p2.setBorder(BorderFactory.createTitledBorder(aBorder, "Check Boxes"));

JCheckBox cb1 = new JCheckBox("This choice"); //initially unchecked
JCheckBox cb2 = new JCheckBox("And / Or this choice", true); //checked
JCheckBox cb3 = new JCheckBox("And / Or this choice", true); //checked

// Position and size the check boxes and their titles
cb1.setBounds(10, 30, 140, 20); //x, y, width, height
cb2.setBounds(10, 60, 140, 20);
cb3.setBounds(10, 90, 140, 20);
```



```
//Add them to the JPanel, p2
p2.add(cb1);
p2.add(cb2);
p2.add(cb3);
```

Determining a Check Box's Status

The status of a check box, checked or unchecked, can be determined by invoking the `isSelected` method on it. The method returns the Boolean value `true` if the box is checked when the method is invoked, otherwise, it returns `false`. The following code fragment outputs `true` to the system console because check box `cb2` was created with a check in it:

```
//Determine if a check box is checked (its status)
JCheckBox cb2 = new JCheckBox("And / Or this choice", true);
if(cb2.isSelected() == true) //cb2 is checked
{
    System.out.println("true");
}
```

Check Box Events

In most applications that use check boxes, the interface contains a button that is clicked after the user checks one or more of the check boxes, and then the processing associated with the checked boxes is performed from within the button's event handler method, `actionPerformed`. When this is the case, the status of the check boxes is determined by invoking the `isSelected` method within the event handler method `actionPerformed`. The coding of this method and the techniques used to register it in the button's event handler list are those discussed in Section 11.4.

In applications where it is important to perform some processing *immediately after* a check box is checked, a check box event handler is implemented and registered with the check box's listener list. When the user checks or unchecks a check box, an item event occurs. The application can immediately perform some processing in response to this event by implementing the event handler method `itemStateChanged` inside the class that declared the check boxes or inside an inner class. In either case, the class's heading must indicate that it implements the interface `ItemListener`. The event handler's signature, which is given below, is the only signature defined within the interface `ItemListener`:

```
public void itemStateChanged(ItemEvent e)
```

The `itemStateChanged` method is added to the check box's event listener list by invoking the `addItemListener` method on the check box object and passing it the keyword `this`. When the method is implemented within an inner class, an instance of the inner class is declared and passed to the method.

The code fragment shown in Figure 12.2 outputs *Hamburger Selected* or *Hamburger Unselected* when the check box `cb1`, declared on line 9, is selected (checked) or unselected (unchecked). The code assumes that the event handler method `itemStateChanged` (lines 12–25) is written as shown (not coded inside of an inner class); which is why the keyword `this` is passed to the method

invoked on line 10 to add the event handler method to `cb1`'s event listener list. As indicated at the top of the figure, two imports must be included in the class's source file, and the class's heading must indicate that it implements the interface `ItemListener`.

```

1  //Two imports needed and an implements clause in the class' heading
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  //Class heading and implements clause would be here
6  JCheckBox cb1; //class level variable
7
8  //coded in the class' constructor
9  cb1 = new JCheckBox("Hamburger");
10 cb1.addItemListener(this); //add event handler to cb1's list
11
12 public void itemStateChanged(ItemEvent e) //event handler method
13 {
14     if(e.getSource() == cb1) //cb1's box was clicked
15     {
16         if(cb1.isSelected() == true)
17         {
18             System.out.println("Hamburger Selected");
19         }
20         else
21         {
22             System.out.println("Hamburger Un-selected");
23         }
24     }
25 }
```

Figure 12.2

A code fragment that illustrates check box event handling without the use of an inner class.

12.1.3 Radio Buttons

Ordinarily, a radio button is grouped with other radio buttons into a mutually exclusive grouping because their most common use in GUIs is to facilitate the selection of one input from a small set of valid inputs. When the user clicks a radio button, a dot either appears on the button or is removed from it. Using the techniques discussed in this section, radio buttons can be created, added to, and positioned in GUI containers and made mutually exclusive. Processing is then initiated based on their selected or unselected status.

Creating Radio Buttons

A radio button is an instance of the API class `JRadioButton` and can be created using the class's one-parameter constructor or its default constructor. The text that appears beside the radio button when it is displayed is the string passed to the one-parameter constructor or the string

passed to the class's `setText` method. The following code fragment creates the first two radio buttons shown in the top-right portion of Figure 12.1:

```
//Create radio buttons and initialize their text.
JRadioButton rb1 = new JRadioButton("This choice");
JRadioButton rb2 = new JRadioButton();
rb2.setText("Or this choice");
```

The `setText` method sets the text property of most GUI components that display text, and it can be used to initially set or to change the text associated with a component. The properties of radio buttons can be set and fetched using the methods shown in Table 12.1.

By default, a radio button is initially displayed unselected, without a center dot on it. A Boolean argument can be passed to the `JButton` class's two-parameter constructor to specify that the button will contain a dot (be selected) when the radio button is initially displayed.

```
//Display a dot in a radio button
JRadioButton rb3 = new JRadioButton("Button is selected", true);
```

Making Radio Buttons Mutually Exclusive

By default, a set of radio buttons are not mutually exclusive: one, several, or all of them could be selected at the same time. Because they are ordinarily used to choose one input from a set of mutually exclusive inputs, a set of radio buttons is designated to be mutually exclusive. When this designation is made, after one button in the set is selected, the previously selected button in the set is simultaneously deselected.

To designate a set of radio buttons to be mutually exclusive, the buttons are added to an instance of the class `ButtonGroup`. The following code fragment creates a mutually exclusive set of three radio buttons:

```
// Designate a set of radio buttons to be mutually exclusive
JRadioButton rb1 = new JRadioButton("This choice", true);
JRadioButton rb2 = new JRadioButton("Or this choice");
JRadioButton rb3 = new JRadioButton("Or this Choice");

// Create a radio button grouping
ButtonGroup bg1 = new ButtonGroup();

// Add buttons to the grouping bg1
bg1.add(rb1);
bg1.add(rb2);
bg1.add(rb3);
```

If several of the buttons in a mutually exclusive grouping were declared to be selected by passing the two-parameter constructor the value `true` when they are created, only the first button created will be selected when the group is initially displayed. A set of radio buttons can be made mutually exclusive using this same technique: create a `ButtonGroup` object and then add the radio buttons to the object. This is usually not done because it is contrary to the common inclusive use of radio buttons in graphical interfaces.

Adding Radio Buttons to Containers and Positioning Them

Radio buttons are added to a GUI container using the `add` method, which is described at the bottom of Table 12.1. The following code fragment adds two mutually exclusive radio buttons to a `JPanel` container:

```
//Add two mutually exclusive radio buttons to a JPanel container
p1 = new JPanel();
JRadioButton rb1 = new JRadioButton("Hamburger");
JRadioButton rb2 = new JRadioButton("Taco");
ButtonGroup bg1 = new ButtonGroup();
bg1.add(rb1);
bg1.add(rb2);
p1.add(rb1);
p1.add(rb2);
```

Radio buttons, like other components added to a container, are positioned within it using the techniques discussed in Chapter 11 (Section 11.3.3 without a layout manager and Section 11.5 using a layout manager). When a layout manager is used, the radio buttons are positioned in the container by the manager in the order in which they are added to the container.

If the container's layout manager has been set to `null`, the `setBounds` method described in Table 12.1 can be used to position and set the height and width of the component and its associated text. Alternately, the `setLocation` and `setSize` methods, can be used to position a radio button in the container and to specify the height and width of the button and its associated text.

The following code fragment was used to create the three radio buttons shown in Figure 12.1, and the panel in which they are contained. The panel does not use a layout manager to position its components because the panel's `setLayout` method is passed a `null` value. This permits the use of the `setBounds` method to position and size the components added to it. The (x, y) position specified by the first two arguments sent to the method `setBounds` is relative to the upper left corner of the container to which a component has been added.

```
// Position and size radio buttons without using a layout manager
p3 = new JPanel();
p3.setLayout(null); //no border manager used
aBorder = BorderFactory.createBevelBorder(0);
p3.setBorder(BorderFactory.createTitledBorder(aBorder, "Radio Buttons"));

JRadioButton rb1 = new JRadioButton ("This choice");
JRadioButton rb2 = new JRadioButton ("Or this choice");
JRadioButton rb3 = new JRadioButton ("Or this choice", true);

ButtonGroup bg1 = new ButtonGroup();
bg1.add(rb1);
bg1.add(rb2);
bg1.add(rb3);

// Position and size the check boxes and their titles
rb1.setBounds(10, 30, 140, 20); //x, y, width, height
```

```
rb2.setBounds(10, 60, 140, 20);
rb3.setBounds(10, 90, 140, 20);

//Add them to the JPanel, p3
p3.add(rb1);
p3.add(rb2);
p3.add(rb3);
```

Determining a Radio Button's Status

The status of a radio button, selected or not selected, can be determined by invoking its `isSelected` method. The method returns the Boolean value `true` if the button is selected when the method is invoked, otherwise, it returns `false`. The following code fragment outputs `true` to the system console because the two-parameter constructor is passed the value `true` when the button is created:

```
//Determine if a radio button is selected
JRadioButton rb1 = new JRadioButton ("Radio button selected", true);
if (rb1.isSelected() == true) //rb1 is selected
{
    System.out.println("true");
}
```

Radio Buttons Events

In most applications that use radio buttons, the interface contains a `JButton` that is clicked after the user selects one of the radio buttons, and then the processing associated with the selection is performed from within the `JButton`'s event handler method, `actionPerformed`. When this is the case, the determination of which radio button in a group was selected is made by invoking the `isSelected` method on each of the buttons from within the `actionPerformed` method. The coding of this method and the techniques used to register it in the `JButton`'s event handler list were discussed in Section 11.4.

In applications where it is important to perform processing *immediately after* a radio button is selected, an event handler is implemented and registered with the radio button's listener list. The selection of a radio button generates an action event, which means the techniques used to perform processing when a radio button is selected are the same techniques used to perform processing when a `JButton` is clicked (discussed in Chapter 11). The application implements the event handler method `actionPerformed` inside the class that declared the radio button or within an inner class. In either case, the class's heading must indicate that it implements the interface `ActionListener`. The `actionPerformed` method's signature, which is given below, is the only signature defined within the interface `ActionListener`:

```
public void actionPerformed(ActionEvent e)
```

The method is added to the radio button's event listener list by invoking the `addActionListener` method on the radio button object and passing it the keyword `this`. When the method is implemented within an inner class, an instance of the inner class is declared and passed to the method.

The code fragment shown in Figure 12.3 outputs *Hamburger Selected* when the radio button `rb1`, declared on line 10, is selected, and it outputs *Taco Selected* when the radio button `rb2`,

declared on line 12, is selected. The code assumes that the event handler method `actionPerformed` (lines 15–25) is not coded inside of an inner class; the keyword `this` is passed to the method invoked on lines 11 and 13 to add the event handler method to the buttons' event listener lists. As indicated at the top of the figure, two imports must be included in the class's source file, and the class's heading must indicate that it implements the interface `ActionListener`.

```

1  //Need two imports and an implements clause in the class' heading
2  import javax.swing.*;
3  import java.awt.event*;
4
5  //Class heading and implements clause would be here
6  JRadioButton rb1; //class level variables
7  JRadioButton rb2;
8
9  //coded in the class' constructor
10  rb1 = new JRadioButton("Hamburger");
11  rb1.addItemListener(this); //add event handler method to rb1's list
12  rb2 = new JRadioButton("Taco");
13  rb2.addItemListener(this); //add event handler method to rb2's list
14
15  public void actionPerformed(ActionEvent e)
16  {
17      if(rb1.isSelected() == true) //or e.getSource() == rb1 can be used
18      {
19          System.out.println("Hamburger Selected");
20      }
21      if(e.getSource() == rb2) //or rb2.isSelected() == true can be used
22      {
23          System.out.println("Taco Selected");
24      }
25  }

```

Figure 12.3

A code fragment that illustrates radio button event handling without the use of an inner class.

The GUI application `DollarMeal`, shown in Figure 12.4, illustrates the use of check boxes and radio buttons in a GUI application. It declares (on line 7) and displays (on line 9) an instance of the class GUI-builder worker class `MealMenu` shown in Figure 12.5. This class's constructor (lines 11–65) builds the GUI, shown in Figure 12.6a, which the user can use to order a meal. A typical order is shown in Figure 12.6b. A summary of the order is output to the system console (bottom of Figure 12.6) when the Place Order button is clicked at the bottom of the GUI.

The class `MenuMeal` extends `JFrame` and adds two panels (named `p1` and `p2`, declared on lines 18 and 41) and a `JButton` (named `placeOrder`, defined on line 58) to the frame's content pane on lines 62–64. The layout managers of the content pane and the panels are set to `null` (lines 14, 19, and 42) to allow the panels and `JButton` to be positioned in the frame (lines 21, 44, and 59) and the panels' contents (check boxes and radio buttons) to be positioned inside of them using the `setBounds` method. The radio buttons and check boxes are created, located, and sized and then

added to the panels on lines 23–38 and lines 46–56, respectively. The radio buttons are also added to a `ButtonGroup` to make them mutually exclusive (lines 31–34.).

The string `order` (declared on line 71) is output to the console on line 104 of the `JButton`'s event handler `actionPerformed` (lines 68–105), which is added to the button's listener list on line 60. The method builds the output string using the check boxes' and radio buttons' `isSelected` method in a series of `if-else` and `if` statements to determine which of them have been selected (lines 73–103).

```

1  import javax.swing.*;
2
3  public class DollarMeal
4  {
5      public static void main(String[] args)
6      {
7          MealMenu window = new MealMenu();
8          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9          window.setVisible(true);
10     }
11 }
```

Figure 12.4

The application `DollarMeal`.

```

1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class MealMenu extends JFrame implements ActionListener
5  {
6      JPanel p1, p2;
7      JRadioButton hamburger, taco, blt;
8      JCheckBox cheese, ketchup, napkins;
9      JButton placeOrder;
10
11     public MealMenu()
12     {
13         super("Dollar Meals");
14         setLayout(null);
15         setSize(303, 200);
16
17         //Build the radio button entree panel
18         p1 = new JPanel(); //declare the panel
19         p1.setLayout(null);
20         p1.setBorder(BorderFactory.createTitledBorder("Entree"));
21         p1.setBounds(5, 10, 140, 110); //locate and size the panel
22
23         hamburger = new JRadioButton("Hamburger", true); //declare buttons
24         taco = new JRadioButton("Taco");
25         blt = new JRadioButton("BLT Sandwich");
26
27         hamburger.setBounds(10, 20, 120, 20); //locate and size the buttons
```

```

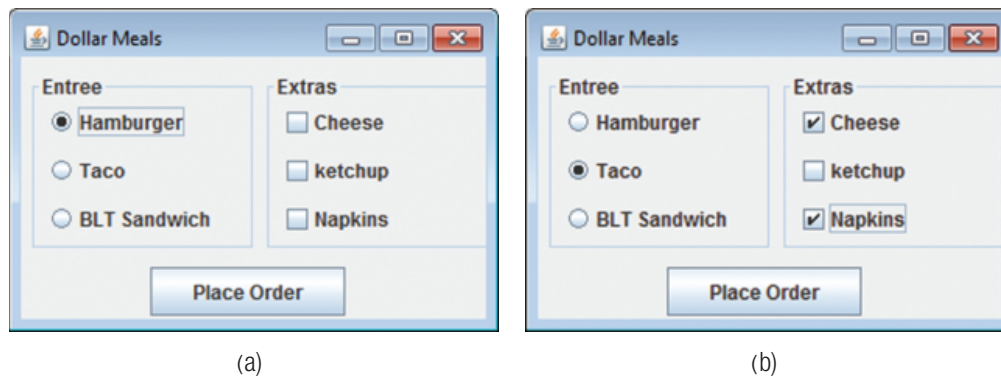
28     taco.setBounds(10, 50, 120, 20);
29     blt.setBounds(10, 80, 120, 20);
30
31     ButtonGroup bg1 = new ButtonGroup(); //group the buttons
32     bg1.add(hamburger);
33     bg1.add(taco);
34     bg1.add(blt);
35
36     p1.add(hamburger); //add the buttons to the panel
37     p1.add(taco);
38     p1.add(blt);
39
40     //Build the check box extras panel
41     p2 = new JPanel(); //declare the panel
42     p2.setLayout(null);
43     p2.setBorder(BorderFactory.createTitledBorder("Extras"));
44     p2.setBounds(150, 10, 140, 110); //locate and size the panel
45
46     cheese = new JCheckBox("Cheese"); //declare the check boxes
47     ketchup = new JCheckBox("Ketchup");
48     napkins = new JCheckBox("Napkins");
49
50     cheese.setBounds(10, 20, 120, 20); //locate and size the check boxes
51     ketchup.setBounds(10, 50, 120, 20);
52     napkins.setBounds(10, 80, 120, 20);
53
54     p2.add(cheese); //add the check boxes to the panel
55     p2.add(ketchup);
56     p2.add(napkins);
57
58     placeOrder = new JButton("Place Order"); //declare the JButton
59     placeOrder.setBounds(80, 130, 120, 30); //locate and size it
60     placeOrder.addActionListener(this); //register the event handler
61
62     add(p1); //add the panels and the JButton to the content pane
63     add(p2);
64     add(placeOrder);
65 }
66
67 //Place order button handler
68 public void actionPerformed(ActionEvent e)
69 {
70     int extras = 0;
71     String order = "";
72
73     if(hamburger.isSelected() == true)
74     {
75         order = order + "Hamburger ";
76     }

```

```

77     else if(taco.isSelected() == true)
78     {
79         order = order + "Taco ";
80     }
81     else if(blut.isSelected() == true)
82     {
83         order = order + "BLT sandwich ";
84     }
85     if(cheese.isSelected() == true)
86     {
87         order = order + " and cheese";
88         extras++;
89     }
90     if(ketchup.isSelected() == true)
91     {
92         order = order + " and ketchup";
93         extras++;
94     }
95     if (napkins.isSelected() == true)
96     {
97         order = order + " and napkins";
98         extras++;
99     }
100    if(extras == 0)
101    {
102        order = order + " no extras";
103    }
104    System.out.println(order);
105 }
106 }

```

Figure 12.5The class `MealMenu`.**Console output:***Taco and cheese and napkins***Figure 12.6**The GUI of the application `DollarMeal`, a user input, and the corresponding console output.

12.2 COMBO BOXES AND LISTS

The GUI components combo boxes and lists are shown in the lower left and lower center portion of Figure 12.1. These components are similar to a set of radio buttons and a set of check boxes in that they are used to facilitate the selection of one or more inputs from a set of valid inputs. When the number of elements in the set is small, most GUI designers use radio buttons and check boxes to present the selection alternatives. When the set contains a large number of elements, combo boxes and lists are the preferred components because they can include a scroll bar to permit the user to view the alternative selections without taking up a large portion of the program's window. Table 12.2 summarizes the terminology, features, and common uses of the GUI components combo boxes and lists.

Table 12.2

Terminology and Features of Combo Boxes and Lists

Combo Box	List
Element name	
An item	A value
Most Common Use	
Select one item from a set of valid items	Select one or more values from a set of valid values
Elements specified as	
An array containing the items	An array containing the values
Elements can be changed from their initial values during the program's execution	
No	Yes
Instance of	
JComboBox class	JList class
Scrollable	
Yes	Yes
User selection technique	
Click an item	Click an item, Control-Click for multiple items, or Shift-Click for an interval of items
User could type an input that is not an element	
Yes	No

The elements displayed in a combo box are called *items*, and those displayed in a list are called *values*. Only one item can be selected from a combo box, which makes it the component of choice for selecting one item from a large set of mutually exclusive items. A list is normally used when one or more values can be selected. The ability to select one or more values from a list is its default mode, but this can be restricted to a sequential set of values or only one value.

A single element in a combo box or list is selected by clicking it, which causes the previously selected item (or items, in the case of a list) to be simultaneously deselected. Multiple non-sequential values in a list can be selected by clicking them while holding down the Ctrl (control) key on the keyboard. Multiple sequential values in a list can be selected by clicking the first value in the sequence then holding down the Shift key and clicking the last value in the sequence. Table 12.3 summarizes the methods used to create and operate on combo boxes and lists and to service click events on them.

Table 12.3
Methods That Perform Common Combo Box and List Operations

Combo Box Named <code>aBox</code>	List Named <code>aList</code>
Creation	
<code>JComboBox aBox;</code> <code>aBox = new JComboBox(itemArray);</code>	<code>JList aList;</code> <code>aList = new JList(valueArray);</code>
Fetch the index selected or the first value in sequential order selected	
<code>int i = aBox.getSelectedIndex();</code> <code>Object item=aBox.getSelectedItem();</code>	<code>int i = aList.getSelectedIndex();</code> <code>Object value = aList.getSelectedValue();</code>
Fetch all selections	
Not applicable	<code>int[] i = aList.getSelectedIndices();</code> <code>Object[] value = getSelectedValues();</code>
Add a scroll bar	
<code>aBox.setMaximumRowCount(aLowCount);</code>	<code>aList.setVisibleRowCount(aLowCount);</code> <code>JScrollPane sp= new JScrollPane(aList);</code>
Change displayed elements	
Not permitted	<code>aList.setListData(newValueArray);</code>
Permit User to type a new element	
<code>aBox.setEditable(true);</code>	Not permitted
Event handling	
Interface: <code>ActionListener</code> Event handler: <code>actionPerformed(ActionEvent e)</code> Register event handler using: <code>addActionListener</code>	Interface: <code>ListSelectionListener</code> Event handler: <code>valueChanged(ListSelectionEvent e)</code> Register event handler using: <code>addListSelectionListener</code>

Creating Combo Boxes and Lists

A combo box is an instance of the class `JComboBox`, and a list is an instance of the class `JList`. The elements displayed in both types of instances are placed in an array passed to their class's one-parameter constructor when they are created. They maintain the index that was

associated with them in the array passed to the `JComboBox` and `JList` constructors and are displayed in ascending index order. The following code sequence creates a combo box and a list that displays the days of the week beginning with Sunday. The list will display all seven days, the combo box will display the seven days when the arrow at the top of it is clicked.

```
//Create a Combo Box and a List that display the days of the week
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

JComboBox aBox = new JComboBox(days);
JList aList = new JList(days);
```

Fetching the Selected Item and Value(s)

The index of the item selected in a combo box, or the *lowest* index of the values selected in a list, can be fetched by invoking the components' `getSelectedIndex` method. Alternately, the item selected in a combo box or the value with the lowest index selected in a list can be fetched using the `getSelectedItem` and the `getSelectedValue` methods, respectively. Both of these methods return a reference to an object, which must be cast into the type of the reference variable to which it is assigned. Assuming the program user selected Wednesday in the combo box and selected Monday and Thursday in the list, the following code sequence would output two lines to the system console containing *3 Wednesday* followed by *1 Monday*:

```
//Fetch the item and first value selected in a combo box and list
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};
int comboIndex, listIndex;
String item, value;

JComboBox aBox = new JComboBox(days);
JList aList = new JList(days);

//After the user makes selections, the following code is executed
comboIndex = aBox.getSelectedIndex();
item = (String) aBox.getSelectedItem();
listIndex = aList.getSelectedIndex();
value = (String) aList.getSelectedValue();

System.out.println(comboIndex + " " + item);
System.out.println(listIndex + " " + value);
```

The indices of all of the values selected in a list can be fetched by invoking the `getSelectedIndices` method on the component object. All of the selected values can be fetched by invoking the `getSelectedValues` method on the component object. Both methods return the address of an array. The indices are returned in an integer array, and the values are returned in an array of `Object` references. Assuming the program user selected Monday and Thursday in the list, the following code sequence would output two lines to the system console: *1 Monday* followed by *4 Wednesday*. The output would not depend on the order in which the user made the selections.

```
//Fetch all indices and values selected from a list
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};
int[] listIndices;
String[] values;
JList aList = new JList(days);

//After the user makes selections, the following code is executed
listIndices = aList.getSelectedIndices();
values = (String) aList.getSelectedValues();

for(int i = 0; i < values.length; i++)
{
    System.out.println(listIndices[i] + " " + values[i]);
}
```

Adding a Vertical Scroll Bar

A combo box can be, and a list is normally, displayed with a vertical scroll bar on their right side, which is used to scroll through the set of valid inputs. The techniques used to select and fetch elements from these components do not change when scroll bars are incorporated into them.

The `setMaximumRowCount` method is invoked on a combo box object to set the number of items displayed at one time *and* to add a scroll bar to its right side. This one invocation is all that is required to add a scroll bar to a combo box, as shown in following code fragment, which creates a combo box with a scroll bar that displays four days of the week at a time:

```
//Add a scroll bar to a combo box
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

JComboBox aBox = new JComboBox(days);
aBox.setMaximumRowCount(4); //four sequential items displayed at a time
```

When the component is a list, two steps are required. The `setVisibleRowCount` method is invoked on the list object and passed the number of items to be displayed at one time in the component. Then, an instance of a `JScrollPane` is created, and the list object is passed to the class's one-parameter constructor. The `JScrollPane` object, not the list (`JList`) object, is subsequently added to the GUI container. The following code fragment creates a list with a scroll bar that displays four days of the week at a time:

```
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

JList aList = new JList(days);
JPanel aPanel = new JPanel();

aList.setVisibleCount(4); //four sequential items displayed at a time
JScrollPane aScrollableList = new JScrollPane(aList);
aPanel.add(aScrollableList);
```

If a layout manager is not used to position the scroll pane in the container (e.g., the `JPanel`'s layout manager is set to `null`), the `setVisibleCount` method is not invoked to set the number of items displayed at one time. Instead, the height of the `JScrollPane` object, required to display the desired number of rows, is passed to the invocation of the `setBounds` method used to position and size the scroll pane object within the container, as shown below:

```
//Position and size the scroll pane object
String days = {"Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"};

JList aList = new JList(days);
JPanel aPanel = new JPanel();
aPanel.setLayout(null);

JScrollPane aScrollableList = new JScrollPane(aList);
aScrollableList.setBounds(10, 20, 120, 80); //80 displays 4 values
aPanel.add(aScrollableList);
```

The application `ExpandedDollarMeal`, shown in Figure 12.7, is a modified version of the application `DollarMeal` presented in Figure 12.4. It declares (line 8) and displays (line 10) an instance of the class `ExpandedMealMenu` shown in Figure 12.8. This class's constructor (lines 17–52) builds the GUI shown on the left side of Figure 12.9. A typical meal order is shown on the right side of the figure. When the button at the bottom of the interface is clicked, a summary of the order is output to the system console (bottom of Figure 12.9).

In a similar way to the class `MealMenu` shown in Figure 12.5, the `ExtendedMealMenu` class extends `JFrame` and adds two panels (named `p1` and `p2`, declared on lines 24 and 35) and a `JButton` (named `placeOrder`, defined on line 45) to the frame's content pane on lines 49–51. The layout managers of the content pane and the panels are set to `null` (lines 20, 25, and 36) to allow the panels to be positioned in the frame (lines 27 and 38) and the panels' contents (a combo box and a list) to be positioned inside of them using the `setBounds` method.

The class `ExtendedMealMenu` uses a combo box with a scroll bar to display the expanded number of entrees (seven), defined on lines 7–9, and a list with a scroll bar to display and the expanded number of extras (nine), defined on lines 10–12. These arrays are passed to the constructors used to create the combo box `entree` on line 29 and list `extrasList` on line 40; then the combo box and the list are located, sized, and added to the panels on lines 30–32 and lines 41–43, respectively. Line 30 adds a scroll bar to the combo box that scrolls through four items at a time. Line 41 adds the list to a `JScrollPane` object. The location and size of the scroll pane is set on line 42. The last argument, 80, sent to the invocation of the `setBounds` method on this line, is a height sufficient to display four values in the now-scrollable list, which is added to the panel (`p2`) on line 43.

The console output is produced on lines 59–66 of the `JButton`'s event handler `actionPerformed` (lines 55–67), which is added to the button's listener list on line 47.

Lines 59 and 60 use the combo box's `getSelectedIndex` and `getSelectedItem` methods to output the selected entrée's index and name. The array of selected extras returned from the `JList` class's `getSelectedValues` method on line 62 is output to system console inside the `for`

loop that begins on line 63. The array reference variable `extrasOrderedArray` is declared on line 57 to be an array of `Object` references, because the `getSelectedValues` method returns a reference to an array of objects.

```

1  import javax.swing.*;
2
3  public class ExpandedDollarMeal
4  {
5      public static void main(String[] args)
6      {
7          String title = "Expanded Dollar Meal";
8          ExpandedMealMenu window = new ExpandedMealMenu(title);
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11         window.setVisible(true);
12     }

```

Figure 12.7

The application **ExpandedDollarMeal**.

```

1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class ExpandedMealMenu extends JFrame implements ActionListener
5  {
6      JPanel p1, p2;
7      String[] entreeItems = {"Hamburger", "Taco", "BLT Sandwich",
8                             "Nachos", "Chicken Soup", "Hot Chili",
9                             "Salad"};
10     String[] extrasValues = {"Cheese", "Ketchup", "Napkins", "Mustard",
11                             "Mayonnaise", "Salsa", "Paper Plate",
12                             "Utensils", "Water"};
13
14     JComboBox entree;
15     JList extrasList;
16     JButton placeOrder;
17
18     public ExpandedMealMenu(String title)
19     {
20         super(title);
21         setLayout(null);
22         setSize(303, 200);
23
24         //Build the entree panel
25         p1 = new JPanel(); //declare the panel
26         p1.setLayout(null);
27         p1.setBorder(BorderFactory.createTitledBorder("Entree"));

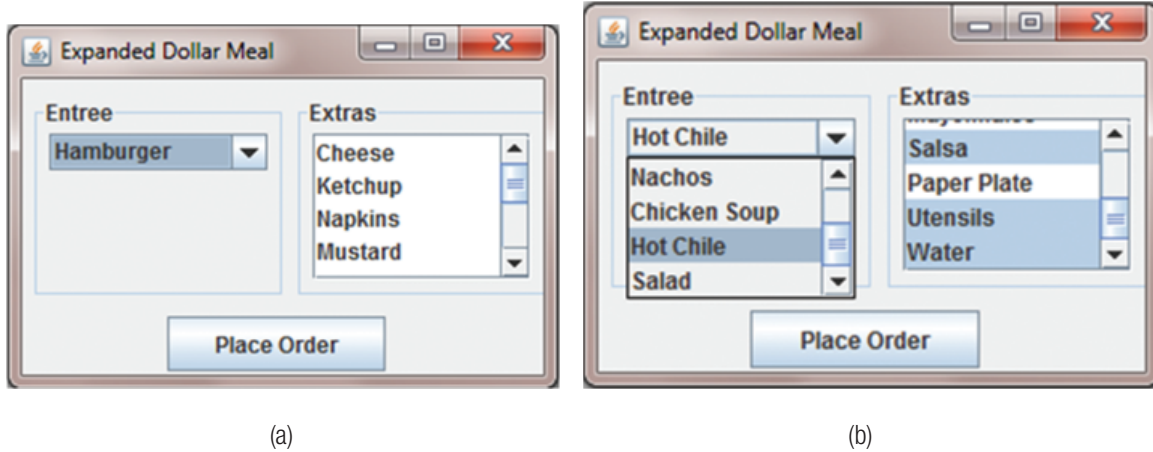
```

```

27     p1.setBounds(5, 10, 140, 110); //locate and size the entree panel
28
29     entree = new JComboBox(entreeItems);
30     entree.setMaximumRowCount(4);
31     entree.setBounds(10, 20, 120, 20);
32     p1.add(entree);
33
34     //Build extras panel
35     p2 = new JPanel(); //declare the panel
36     p2.setLayout(null);
37     p2.setBorder(BorderFactory.createTitledBorder("Extras"));
38     p2.setBounds(150, 10, 140, 110); //locate and size the extras panel
39
40     extrasList = new JList(extrasValues);
41     JScrollPane aScrollableList = new JScrollPane(extrasList);
42     aScrollableList.setBounds(10, 20, 120, 80); //80 displays 4 values
43     p2.add(aScrollableList);
44
45     placeOrder = new JButton("Place Order"); //declare the JButton
46     placeOrder.setBounds(80, 130, 120, 30); //locate and size it
47     placeOrder.addActionListener(this); //register its event handler
48
49     add(p1); //add the panels and the JButton to the content pane
50     add(p2);
51     add(placeOrder);
52 }
53
54 //Place order button handler
55 public void actionPerformed(ActionEvent e)
56 {
57     Object[] extrasOrderedArray;
58
59     System.out.print("\nEntree Number " + entree.getSelectedIndex() +
60                     ": " + entree.getSelectedItem() );
61
62     extrasOrderedArray = extrasList.getSelectedValues();
63     for(int i = 0; i < extrasOrderedArray.length; i++)
64     {
65         System.out.print(", " + extrasOrderedArray[i]);
66     }
67 }
68 }

```

Figure 12.8The class **ExpandedMealMenu**.

**Console Output:**

Entree Number 5: Hot Chili, Salsa, Utensils, Water

Figure 12.9

The GUI of the application `ExpandedDollarMeal`, a user input, and the corresponding console output.

The values initially displayed in a `JList` object, which are passed to the class's constructor when the list is created, can be changed. To do this, the method `setListData` is invoked on the `JList` object and passed an array containing the list's new values. The following line of code displays the objects contained in the array `newValueArray` in the `JList` object `aList`:

```
aList.setListData(newValueArray);
```

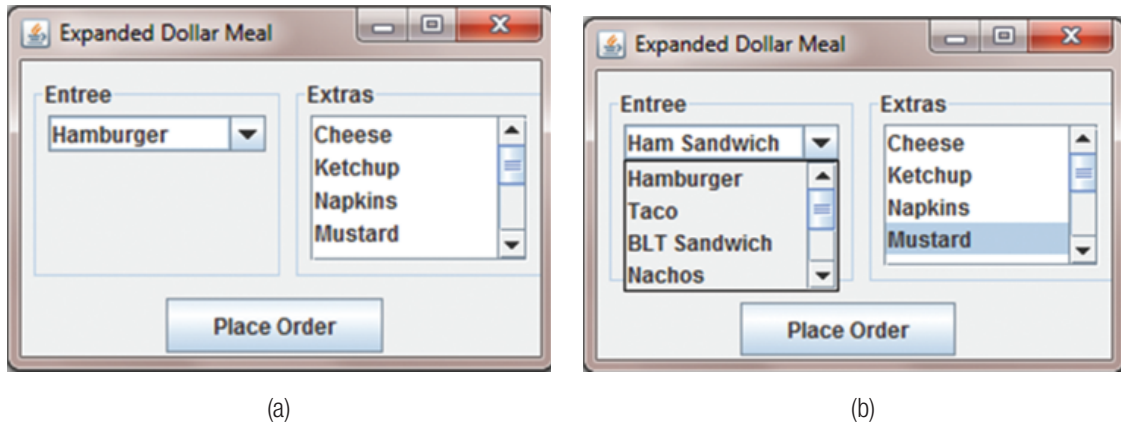
An Editable Combo Box

Each `JComboBox` object has a Boolean data member `isEditable` whose default value is `false`. When the data member's value is set to `true` using the class's `setEditable` method, a text box is displayed at the top of the combo box, as shown in Figure 12.10a. This now-editable combo box retains all the functionality of the un-editable version discussed in this section, and it also allows the user to type an item into the text box. The item typed does not have to be one of the items passed to the one-parameter constructor when the combo box was created.

For example, if the user were to type *Ham Sandwich* into the text box, the combo box would appear as shown in Figure 12.10b. The item typed in the text box would be returned from the next invocation of the box's `getSelectedItem` method, unless one of the other items in the combo box is selected before it is invoked.

The window shown in Figure 12.10a was produced by the application `ExpandedDollarMeal` (Figure 12.7) after the following line of code was added to the class `ExpandedMealMenu` (Figure 12.8) just before line 32:

```
entree.setEditable(true);
```

Console Output

Entree Number -1: Ham Sandwich, Mustard

Figure 12.10

The GUI of the application `ExpandedDollarMeal` with an editable combo box added to it.

The console output at the bottom of Figure 12.10 was produced after the user typed *Ham Sandwich* in the text box and clicked “Mustard” and the Place Order button in the window on the right side of the figure. Because *Ham Sandwich* is not one of the items in the array `entreeItems` (lines 7–9 of Figure 12.8) passed to `JComboBox`’s one-parameter constructor when the combo box was created, the index returned from the method `getItemSelected` is -1 (as shown at the bottom of Figure 12.10).

Combo Box and List Event Handling

When an item is selected in a combo box, an *action event* occurs; when a value is selected in a list, a *list selection event* occurs. In most applications, we do not respond to either of these events individually because the GUI usually contains a `JButton` object that is clicked after the user has interacted with all of the other input components on the interface. That was the case in the application `ExpandedValueMeal`, in which the user clicked the Place Order button after the meal selection was made. Until that button was clicked, the user could change the selections made in both the combo box and the list. The servicing of the button-click action event was performed by the event handler coded on lines 55–67 of the `ExpandedMealMenu` class presented in Figure 12.8, and this event handler was registered in the button’s listener list on line 47 of that class.

To service either a combo-box item selection or a list value selection at the time the selection is made, we implement event handlers and register them with the components’ listener lists. The techniques used to perform this are the same techniques used in Figure 12.8, which were discussed in Section 11.4:

1. Implement the event handler’s interface by coding the event handler method whose signature is defined in the interface
2. Register the method with the components’ event listener list.

In the case of a combo box action event, the interface and the method used to register the event are the same as those used to service a `JButton` click action event:

1. The interface is `ActionListener`;
2. The signature of the event handler is `actionPerformed(ActionEvent e)`
3. The method used to register the event is `addActionListener`, which is invoked on the `JComboBox` object and passed an instance of the class in which the event handler is coded

In the case of a list selection event, the interface and methods use to service the event are:

1. The interface is `ListSelectionListener`;
2. The signature of the event handler is `valueChanged(ListSelectionEvent e)`;
3. The method used to register the event is `addListSelectionListener`, which is invoked on the `JList` object and passed an instance of the class in which the event handler is coded

The names of these interfaces and methods are summarized at the bottom of Table 12.3. The methods previously discussed to fetch the index or element selected, which are also summarized in Table 12.3, are used inside the combo box and list event handler methods to identify and process the selection.

12.3 MENUS

A menu is a GUI component used to obtain one of several valid inputs from the program user. In this way, menus are similar to combo boxes. The advantage menus have over combo boxes and the other GUI components we have discussed is that they can be used to present a wide variety of valid inputs while occupying a relatively small portion of the program's window. In addition, because their placement in the window does not vary from one application to another, they present an input interface that is more familiar to the user.

The Java API supports two types of menus: drop-down menus and pop-up menus. Drop-down menus are positioned in a menu bar whose location in the window is platform dependent, and pop-up menus remain invisible until the user performs a platform-dependent mouse action or key action. The most common position for a menu bar is just below the window's title bar, and the most common mouse action to expose a pop-up menu is a right-button mouse click. We will begin our discussion of menus with drop-down menus contained in a menu bar.

12.3.1 Drop-Down Menus

The program widow in Figure 12.11a was generated on a platform that places the menu bar just below the window's title bar. The menu-bar object contains one drop-down menu object on its left side that has the string "A Menu" associated with it. The user has clicked this menu object to expose the menu's four drop-down objects and then clicked the last of these objects to expose another drop-down menu containing three more objects. Figure 12.11b gives the API classes of the objects that

make up the program's menu, displayed in Figure 12.11a. These classes are `JMenuBar`, `JMenu`, and `JMenuItem`. The two submenu objects shown in Figure 12.11a are instances of the class `JMenu`.



Figure 12.11

Drop-down menu components and their API classes.

The `JMenuItem` objects within a drop-down menu are the terminal components of the menu. These items are the set of valid inputs. The user selects one of these inputs by clicking it, which initiates an action event just as clicking a `JButton` object generates an action event. The processing associated with the selected menu item is performed within the `actionPerformed` event handler method whose signature is defined in the interface `ActionListener`.

Building a Drop-Down Menu System

Generally speaking, a drop-down menu system consisting of a menu bar containing one or more drop-down menus is added to an instance of a `JFrame` using the following four step process. The code fragments used to illustrate each step of the process were used to create the menu system shown in Figure 12.12. These fragments would be added to the constructor of a GUI-builder worker class that extends the class `JFrame`.

1. Create a menu bar object, which is an instance of `JMenuBar`, and add it to the `JFrame` instance using the `JFrame` class's method `setJMenuBar`

```
//Add the menu bar
JMenuBar aMenuBar = new JMenuBar(); //create the menu bar
setJMenuBar(aMenuBar); //add the menu bar to the JFrame
```

2. Create drop-down menu objects, which are instances of `JMenu`, and add them to the menu bar object by invoking the `add` method on them; the drop-down menu's string annotation, which is passed to `JMenu`'s constructor, appears on the menu bar from left to right in the order in which the menus are added to the menu bar

```
//Add two drop down menus to the menu bar
JMenu dropDownMenu1 = new JMenu("Dollar"); //create menu
JMenu dropDownMenu2 = new JMenu("Deluxe"); //create menu
aMenuBar.add(dropDownMenu1); //add a menu to the menu bar
aMenuBar.add(dropDownMenu2); //add a menu to the menu bar
```

3. For each drop-down menu added to the menu bar, repeatedly create and add either an instance of

- a) A clickable terminal menu item, which is an instance of the class `JMenuItem`;

```
JMenuItem saladItem = new JMenuItem("Salad"); //create items
JMenuItem soupItem = new JMenuItem("Chicken Soup");
dropDownMenu1.add(saladItem);
dropDownMenu1.add(soupItem);
```

or

- b) A submenu, which is an instance of the class `JMenu`

```
JMenu subMenu1 = new JMenu("Sandwich"); //create submenu
JMenu subMenu2 = new JMenu("Mexican"); //create submenu
dropDownMenu1.add(subMenu1); //add the 1st submenu to the menu
dropDownMenu1.add(subMenu2); //add the 2nd submenu to the menu
```

The menu items and submenus appear from top to bottom on the drop-down menu in the order in which they are added.

4. Repeat Step 3 for all the submenus; ordinarily, only a set of terminal items are added to the submenus

```
JMenuItem subItem1 = new JMenuItem("Hamburger"); //create item
JMenuItem subItem2 = new JMenuItem("BLT"); //create item
subMenu1.add(subItem1);
subMenu1.add(subItem2);
```



Figure 12.12

A drop-down menu system that contains two submenus.

Menu Separator Bars

Separator bars are added to a drop-down menu to visually group related elements. There are two separator bars in the drop-down menu shown in Figure 12.12: one above the *Sandwich* submenu and one below it. The `JMenu` method `addSeparator` is invoked on a drop-down `JMenu` menu object to add a separator bar to the menu. Menu items, menu separators, and submenus appear from top to bottom within the drop-down menu in the order in which they are added. The two

menu separators in Figure 12.12 were added to the drop-down menu `dropDownMenu1` using the following code fragment:

```
//Add menu separator bars
dropDownMenu1.add(saladItem); //add to the menu as 1st item
dropDownMenu1.add(soupItem); //add to the menu as 2nd item
dropDownMenu1.addSeparator();
JMenu subMenu1 = new JMenu("Sandwich"); //create submenu
dropDownMenu1.add(subMenu1); //add the submenu to the menu
dropDownMenu1.addSeparator();
```

Menu Mnemonics

A mnemonic is a keyboard event that is designated to be equivalent to the user clicking a GUI component such as a menu item or a button. Other names for mnemonics are *shortcut keys* or *hot keys*. The designation is made by invoking the `setMnemonic` method on the component. The method is overloaded and can be passed either a single character or one of the keyboard key codes defined in the class `KeyEvent` as a static integer constant. Every key on the keyboard, and its shifted version, has a unique key-code constant associated with it. For example, the constant static `VK_2` is associated with an unshifted 2 key keystroke, and the constant `VK_AT` is associated with a shifted `@` key keystroke. (VK stands for virtual key.)

When a hot key is designated to be equivalent to clicking a GUI component, striking it when the component is visible is equivalent to clicking the component. In addition, the first occurrence of the key's character in the string associated with the component (the string passed to the constructor when the component is created) is underlined on the menu. The following code fragment designates the S and C keys to be hot keys for the *Salad* and *Chicken Soup* menu items displayed in Figure 12.12:

```
//Designate Hot Keys
JMenuItem saladItem = new JMenuItem("Salad");
JMenuItem soupItem = new JMenuItem("Chicken Soup");
dropDownMenu1.add(saladItem);
dropDownMenu1.add(soupItem);
saladItem.setMnemonic('S');
soupItem.setMnemonic('C');
```

JMenuItem Object Action Events

When the user clicks a `JMenuItem` object or presses a hot key associated with the object, an action event occurs. The processing to be performed when this event occurs is coded in the event handler method `actionPerformed` whose signature is defined in the interface `ActionListener`. The event handler method is added to the `JMenuItem` object's listener list by invoking the method `addActionListener` on the object and passing the method an instance of the class in which the event handler is coded.

The application `Menus`, shown in Figure 12.13, declares (line 8) and displays (line 10) an instance of the class `MenuBarBuilder` shown in Figure 12.14. This class extends `JFrame`, adds a menu bar to the frame (lines 21–22), and then adds the drop-down menu shown on the top left and

right sides of Figure 12.15 to the menu bar (lines 25–26). The console output shown at the bottom of the figure was produced by the application after the user selected *Nachos* from the menu, as shown in Figure 12.15b.

The drop-down menu is created by the method `buildDollarMenu` (lines 30–82 of Figure 12.14) that returns a reference to an instance of a `JMenu`. This method is invoked on line 25 of the class's constructor, and the returned drop-down menu is added to the menu bar on line 26. It is good coding practice to build each drop-down menu added to a menu bar in a separate method because it makes our code more readable and easier to maintain.

Line 33 declares the `JMenu` object `dollarMenu` that will be returned by the method on line 81. Lines 36–47 create the drop-down menu's two items and two submenus (lines 36–39) and adds these four objects and two separators to the `dollarMenu` object (lines 42–47). The code on lines 50–54 creates five submenu items and then adds these five objects to the two submenu objects (lines 57–61).

Lines 64–70 designate hot keys for each of the menu's seven menu selections (`JMenuItem` instances). Lines 73–79 register an action event handler method into each of these instances' listener lists by invoking the `addActionListener` method on each instance and passing it an instance of the inner class `dollarMenuListener`.

The event handler method `actionPerformed` (lines 86–98) is coded within the inner class (lines 84–99). It invokes the `getSource` method on the `ActionEvent` object passed to the method (lines 90–96) to determine which menu item the user selected in Figure 12.15b. In the interest of brevity, a series of one-line `if` statements are used to make the determination and set the selection into the string `entree` declared on line 88. This string is output on line 97 of the event handler method.

```

1  import javax.swing.*;
2
3  public class Menu
4  {
5      public static void main(String[] args)
6      {
7          String title = "Expanded Dollar Meal Menu";
8          MenuBarBuilder window = new MenuBarBuilder(title);
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11         window.setVisible(true);
12     }

```

Figure 12.13

The application **Menus**.

```

1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class MenuBarBuilder extends JFrame
5  {
6      //Menu item references
7      private JMenuItem saladItem;
8      private JMenuItem chickenSoupItem;
9      private JMenuItem bLTSubItem;
10     private JMenuItem hamburgerSubItem;
11     private JMenuItem tacoSubItem;
12     private JMenuItem nachosSubItem;
13     private JMenuItem chiliSubItem;
14
15     public MenuBarBuilder(String title)
16     {
17         super(title);
18         setSize(303, 200);
19
20         //Step 1: Create and add the menu bar to the JFrame
21         JMenuBar menuBar = new JMenuBar();
22         setJMenuBar(menuBar);
23
24         //Step 2: Create and add the menus to the menu bar
25         JMenu dropDownMenu = buildDollarMenu();
26         menuBar.add(dropDownMenu);
27     }
28
29     //Step 3: Create and add the items and the submenus to the menu
30     public JMenu buildDollarMenu() //Builds and returns the dollar menu
31     {
32         //Create the drop down menu
33         JMenu dollarMenu = new JMenu("Dollar");
34
35         //Create the menu items and submenus
36         saladItem = new JMenuItem("Salad");
37         chickenSoupItem = new JMenuItem("Chicken Soup");
38         JMenu sandwichSubMenu = new JMenu("Sandwich");
39         JMenu mexicanSubMenu = new JMenu("Mexican");
40
41         //Add the menu items, submenus, and separators to the menu
42         dollarMenu.add(saladItem);
43         dollarMenu.add(chickenSoupItem);
44         dollarMenu.addSeparator();
45         dollarMenu.add(sandwichSubMenu);
46         dollarMenu.addSeparator();
47         dollarMenu.add(mexicanSubMenu);
48
49         //Create submenu items

```

```

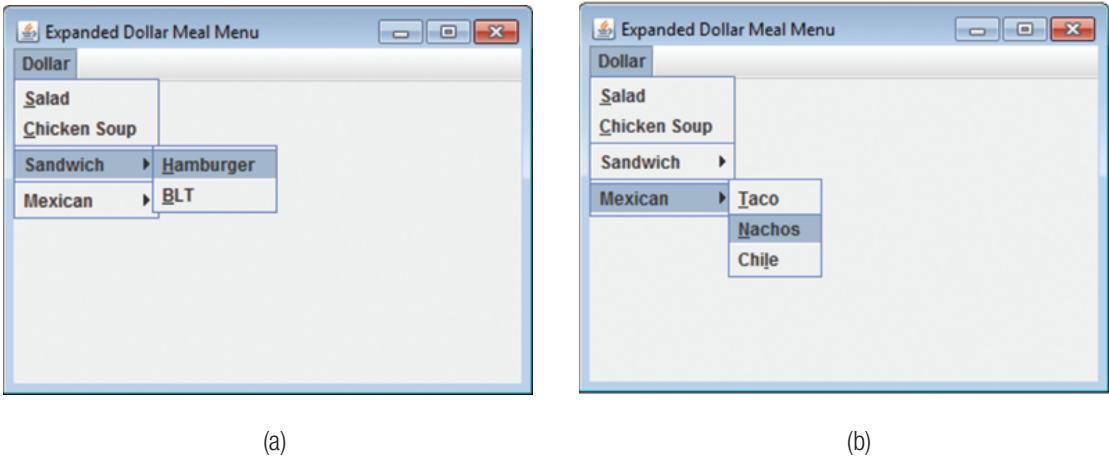
50     bLTSubItem = new JMenuItem("BLT");
51     hamburgerSubItem = new JMenuItem("Hamburger");
52     tacoSubItem = new JMenuItem("Taco");
53     nachosSubItem = new JMenuItem("Nachos");
54     chiliSubItem = new JMenuItem("Chili");
55
56     //Add the submenu items to the submenus
57     sandwichSubMenu.add(hamburgerSubItem);
58     sandwichSubMenu.add(bLTSubItem);
59     mexicanSubMenu.add(tacoSubItem);
60     mexicanSubMenu.add(nachosSubItem);
61     mexicanSubMenu.add(chiliSubItem);
62
63     //Assign mnemonics to the menu items
64     saladItem.setMnemonic('S');
65     chickenSoupItem.setMnemonic('C');
66     bLTSubItem.setMnemonic('B');
67     hamburgerSubItem.setMnemonic('H');
68     tacoSubItem.setMnemonic('T');
69     nachosSubItem.setMnemonic('N');
70     chiliSubItem.setMnemonic('L');
71
72     //Register event handlers
73     saladItem.addActionListener(new dollarMenuListener());
74     chickenSoupItem.addActionListener(new dollarMenuListener());
75     bLTSubItem.addActionListener(new dollarMenuListener());
76     hamburgerSubItem.addActionListener(new dollarMenuListener());
77     tacoSubItem.addActionListener(new dollarMenuListener());
78     nachosSubItem.addActionListener(new dollarMenuListener());
79     chiliSubItem.addActionListener(new dollarMenuListener());
80
81     return dollarMenu;
82 }
83
84 public class dollarMenuListener implements ActionListener
85 {
86     public void actionPerformed(ActionEvent e)
87     {
88         String entree = "";
89
90         if(e.getSource() == saladItem) entree = "Salad";
91         if(e.getSource() == chickenSoupItem) entree = "Chicken Soup";
92         if(e.getSource() == bLTSubItem) entree = "BLT Sandwich";
93         if(e.getSource() == hamburgerSubItem) entree = "Hamburger";
94         if(e.getSource() == tacoSubItem) entree = "Taco";
95         if(e.getSource() == nachosSubItem) entree = "Nachos";
96         if(e.getSource() == chiliSubItem) entree = "Chili";
97         System.out.println("Dollar Meal Entree: " + entree);

```



```
98     }  
99     }  
100 }
```

Figure 12.14
The class `MenuBarBuilder`.



Console Output:
Dollar Meal Entree: Nachos

Figure 12.15
The GUI menu and output produced by the application Menus.

Menu Radio Button and Check Box Items

In addition to `JMenuItem` objects, check boxes and radio buttons can be added to drop-down menus and selected from the menu by the program user. These menu items are instances of the `JCheckBoxMenuItem` and `JRadioButtonMenuItem` classes, respectively. Figure 12.16 shows a drop-down menu that contains four check boxes and four mutually exclusive radio buttons.

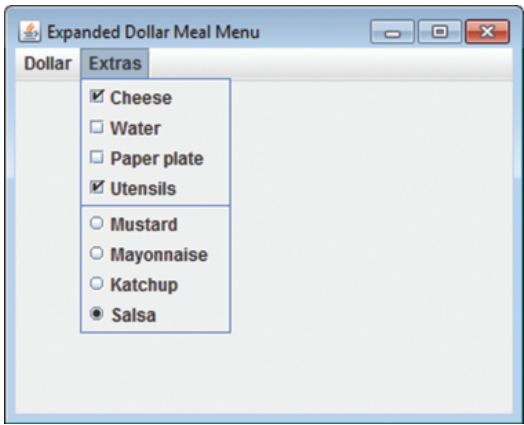


Figure 12.16
A GUI menu containing check boxes and radio buttons.

Once created, radio-button and check-box menu items are added to `JMenu` objects using the same techniques used to add `JMenuItem` objects to drop-down menus and submenus. The techniques used to perform the processing associated with their selection are also the same as the techniques used to process `JMenuItem` selections because radio-buttons and check-boxes added to menus generate action events when they are selected. Their event handler method is `actionPerformed` whose signature is defined in the interface `ActionListener`, and the event handler method is registered by invoking the method `addActionListener` on the menu item.

The following code fragment, when added to the end of the constructor of the class shown in Figure 12.14, creates the drop-down menu shown in Figure 12.16 and adds it to the menu bar created on line 21 of Figure 12.14. It invokes the menu-builder method `buildExtrasMenu` shown in Figure 12.17, which builds the drop-down menu and returns its address.

```
JMenu extras = buildExtrasMenu();
menuBar.add(extras);
```

The method `buildExtrasMenu` would be included as a member method of the class `MenuBarBuilder` shown in Figure 12.14, as would an inner class named `ExtrasMenuListener` that implements the `ActionListener` interface. This inner class would contain the event handler `actionPerformed` registered with the radio buttons and check boxes on lines 35–42 of Figure 12.17. The declarations of the reference variables on lines 7–14 of that figure would be added to the class `MenuBarBuilder` as class-level variables to make them accessible to the radio button and check box event handler.

```
1  public JMenu buildExtrasMenu() //Builds and returns the extras menu
2  {
3      //Create the menu object
4      JMenu extrasMenu = new JMenu("Extras");
5
6      //Create the menu items and submenus
7      cheeseItem = new JCheckBoxMenuItem("Cheese");
8      waterItem = new JCheckBoxMenuItem("Water");
9      paperPlateItem = new JCheckBoxMenuItem("Paper plate");
10     utensilItem = new JCheckBoxMenuItem("Utensils");
11     mustardItem = new JRadioButtonMenuItem("Mustard");
12     mayonnaiseItem = new JRadioButtonMenuItem("Mayonnaise");
13     ketchupItem = new JRadioButtonMenuItem("Ketchup");
14     salsaItem = new JRadioButtonMenuItem("Salsa");
15
16     //Create button group
17     ButtonGroup bg = new ButtonGroup();
18     bg.add(mustardItem);
19     bg.add(mayonnaiseItem);
20     bg.add(ketchupItem);
21     bg.add(salsaItem);
22
23     //Add the menu items to the menu
24     extrasMenu.add(cheeseItem);
```

```

25     extrasMenu.add(waterItem);
26     extrasMenu.add(paperPlateItem);
27     extrasMenu.add(utensilItem);
28     extrasMenu.addSeparator();
29     extrasMenu.add(mustardItem);
30     extrasMenu.add(mayonnaiseItem);
31     extrasMenu.add(ketchupItem);
32     extrasMenu.add(salsaItem);
33
34     //Register event handlers
35     cheeseItem.addActionListener(new ExtrasMenuListener());
36     waterItem.addActionListener(new ExtrasMenuListener());
37     paperPlateItem.addActionListener(new ExtrasMenuListener());
38     utensilItem.addActionListener(new ExtrasMenuListener());
39     mustardItem.addActionListener(new ExtrasMenuListener());
40     mayonnaiseItem.addActionListener(new ExtrasMenuListener());
41     ketchupItem.addActionListener(new ExtrasMenuListener());
42     salsaItem.addActionListener(new ExtrasMenuListener());
43
44     return extrasMenu;
45 }

```

Figure 12.17

The method `buildExtrasMenu`.

12.3.2 Pop-Up Menus

A pop-up menu is a space-saving alternative to a menu-bar-based drop-down menu. Unlike drop-down menus, pop-up menus are associated with a particular component in a graphical interface, and they remain invisible until the user performs a platform-dependent mouse or keyboard action on the GUI component. The most common action on the component is a right mouse click.

Pop-up menus are instances of the class `JPopupMenu` and can be created using the class's default constructor:

```

//Create a pop-up menu object
JPopupMenu aMenu = new JPopupMenu();

```

The techniques discussed in the previous section used to add menu items, submenus, separators, and hot keys to drop-down menus are the same techniques used to add these elements to pop-up menus. Menu items, separators, and submenus are added to the pop-up menu object using the `JPopupMenu` class's `add` method, and hot keys are added using the class's `setMnemonic` method. The following code fragment creates the pop-up menu shown in Figure 12.18a:

```

//Create a pop-up menu and add three menu items to it
JPopupMenu aMenu = new JPopupMenu();

JMenuItem blue = new JMenuItem("Blue"); //create the menu items
JMenuItem red = new JMenuItem("Red");
JMenuItem green = new JMenuItem("Green");

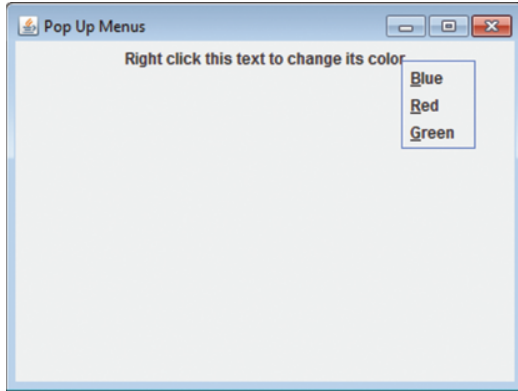
```

```

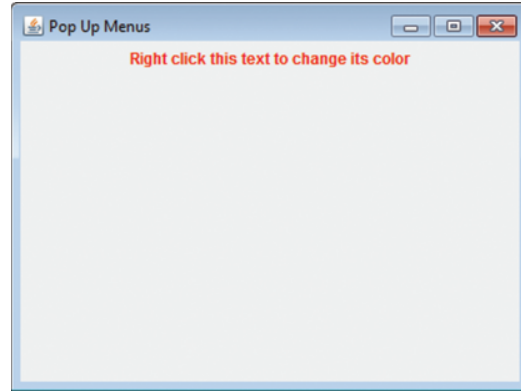
aMenu.add(blue); //add the menu items to the pop-up menu
aMenu.add(red);
aMenu.add(green);

blue.setMnemonic('B'); //designate the hot keys
red.setMnemonic('R');
green.setMnemonic('G');

```



(a)



(b)

Figure 12.18

Windows produced by the application **PopUpMenu**.

To associate a pop-up menu with a GUI component, the `setComponentPopupMenu` method is invoked on the GUI component, and the pop-up menu object is passed to the method. When the user performs the platform-dependent action on the component (e.g., right clicking the component), the menu becomes visible. The last line of the following code fragment was used to associate the menu shown in Figure 12.18a with the `JLabel` displayed at the top of the window:

```

//Associate a pop-up menu with a GUI component
JPopupMenu aMenu = new JPopupMenu();

JMenuItem blue = new JMenuItem("Blue");
JMenuItem red = new JMenuItem("Red");
JMenuItem green = new JMenuItem("Green");

aMenu.add(blue);
aMenu.add(red);
aMenu.add(green);

blue.setMnemonic('B');
red.setMnemonic('R');
green.setMnemonic('G');

//Associate the pop-up menu with a JLabel object
JLabel aLabel = new JLabel("Right click this text to change its color");
aLabel.setComponentPopupMenu(aMenu);

```

Selecting a menu item from a pop-up menu generates an action event, just as selecting an item from a drop-down menu does. As previously discussed, action events are serviced by implementing the event handler method `actionPerformed` whose signature is defined in the interface `ActionListener`. The event handler method is registered with a menu item's listener list by invoking the `addActionListener` method on the menu item object. Assuming the event handler `actionPerformed` was implemented in the class that defined the menu item `blue`, the following code fragment would add the event handler to the item's listener list:

```
//Register a drop-down menu item selection event handler
JMenuItem blue = new JMenuItem("Blue");
blue.addActionListener(this);
```

The application shown in Figure 12.19 uses a pop-up menu to change the color of the text displayed at the top of its window. The Figure 12.18a shows the window after the application is launched and the user has right clicked the text. Figure 12.18b shows the program's window after the user has selected *Red* from the pop-up menu, either by clicking the word *Red* or striking its hot key, R. These actions change the color of the text to red.

The application's window (Line 8 of Figure 12.19) is an instance of the class `PopUpMenuWindow`, shown in Figure 12.20, which extends `JFrame`. The class's constructor builds the pop-up menu and adds it to the `JFrame`. The pop-up menu (`aMenu`) and its three menu items (`blue`, `red`, and `green`), are created on lines 9–12 of the figure. The menu items are added to the menu (lines 20–22), their hot keys are designated (lines 24–26), and the event handler method coded on lines 37–42 is registered with their listener lists on lines 28–30. Line 32 associates the pop-up menu `aMenu` with the `JLabel` created on line 7. Finally, line 33 adds the label to the `JPanel` created on line 19, and then line 34 adds the panel to the `JFrame`.

When the program user right clicks the text at the top of the program window and clicks a selection in the pop-up menu, lines 39–41 of the action event handler method `actionPerformed` changes the color of the displayed text.

```
1  import javax.swing.*;
2
3  public class PopUpMenu
4  {
5      public static void main(String[] args)
6      {
7          String title = "Pop Up Menu";
8          PopUpMenuWindow window = new PopUpMenuWindow(title);
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11         window.setVisible(true);
12     }
```

Figure 12.19

The application `PopUpMenu`.

```

1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.Color;
4
5  public class PopUpMenuWindow extends JFrame implements ActionListener
6  {
7      JLabel aLabel = new JLabel("Right click this text to change its color");
8
9      JPopupMenu aMenu = new JPopupMenu();
10     JMenuItem blue = new JMenuItem("Blue");
11     JMenuItem red = new JMenuItem("Red");
12     JMenuItem green = new JMenuItem("Green");
13
14     public PopUpMenuWindow(String title)
15     {
16         super(title);
17         setSize(400, 300);
18
19         JPanel aPanel = new JPanel();
20         aMenu.add(blue);
21         aMenu.add(red);
22         aMenu.add(green);
23
24         blue.setMnemonic('B');
25         red.setMnemonic('R');
26         green.setMnemonic('G');
27
28         blue.addActionListener(this);
29         red.addActionListener(this);
30         green.addActionListener(this);
31
32         aLabel.setComponentPopupMenu(aMenu);
33         aPanel.add(aLabel);
34         add(aPanel);
35     }
36
37     public void actionPerformed(ActionEvent e)
38     {
39         if(e.getSource() == blue) aLabel.setForeground(Color.BLUE);
40         if(e.getSource() == red) aLabel.setForeground(Color.RED);
41         if(e.getSource() == green) aLabel.setForeground(Color.GREEN);
42     }
43 }

```

Figure 12.20

The class **PopUpMenuWindow**.

12.4 FILE CHOOSER AND COLOR CHOOSER DIALOG BOXES

The API Swing package provides three dialog boxes that can be used to facilitate commonly performed user tasks: specifying the path to a file to be opened or saved and specifying a color to be used in a graphics application. The `JFileChooser` class contains the methods that display file-open and file-save dialog boxes. The `JColorChooser` class contains the method that displays a dialog box containing a predefined palette of colors from which to choose and provides the ability to define a custom color.

12.4.1 File-Chooser Dialog Box

Figure 12.21a shows an Open file-chooser dialog box, and Figure 12.21b shows the Save file-chooser dialog box. Both dialog boxes are displayed by the application, `FileChoosers`, shown in Figure 12.22. The Open dialog box is displayed by line 15, which invokes the `showOpenDialog` method on the `JFileChooser` object `fc` created on line 14. Line 15 of the application does not complete execution until the user clicks the dialog box's Open or Cancel buttons or closes the dialog box. Until one of these events occurs, the user can browse the system's file structure to locate and select a file to be opened, or enter the name of the file into the File Name text field.

When a folder is selected by double clicking its name, the folder name appears in the Look In text field of the dialog box, and its subfolders are displayed. When a file is selected by clicking its name, the file name is displayed in the dialog box's File Name text field. After the Open or Cancel button is clicked, or the dialog box is closed, the `showOpenDialog` method returns an integer whose value is dependent upon which of these three events occurred. Line 15 of Figure 12.22 stores the returned integer in the variable `cancelApproveError`.

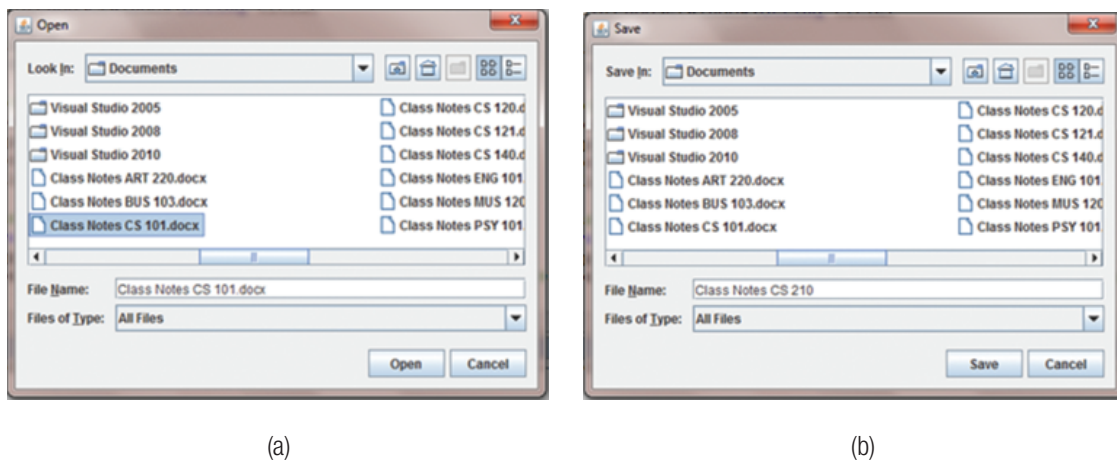


Figure 12.21

File-chooser Open and Save dialog boxes.

Lines 16 and 23 compare this integer to two static integer constants `APPROVE_OPTION` and `CANCEL_OPTION` (defined in the `JFileChooser` class) to determine if the user clicked Open (line 16) or Cancel (line 23). When the user clicks the Cancel button, line 25 reports a record of

that click to the system console. When neither button is clicked, line 29 reports that an error has occurred.

When the user clicks the Open button, line 19 places the address of the string returned from the `File` class's `getPath` method in the `String` variable `path`. This string, which contains the path to the file the user selected and the file's name, is output to the system console (lines 20–21) preceded by a line of annotation (the first two lines of output shown in Figure 12.23). Normally, the string `path` would be used to attach a `Scanner` object to the selected input file, as shown in the following code fragment:

```
File fileObject = new File(path);
Scanner fileIn = new Scanner(fileObject);
```

Lines 33–49 use a similar sequence of code to fetch a path and file name from the user in which to save information generated by the program. The only differences are that line 34 invokes the `showSaveDialog` method on the `JFileChooser` object `fc` to display the Save dialog box shown in Figure 12.21b, and the string referenced by the variable `path` on line 38 would be used to attach a `PrintWriter` object to the specified output file, as shown in the following code fragment:

```
FileWriter fileWriterObject = new FileWriter(path);
PrintWriter fileOut = new PrintWriter(fileWriterObject, false);
```

The last two lines of output shown in Figure 12.23 were produced by lines 39–40, after the user typed *Class Notes CS 210* in the text field of the dialog box shown in Figure 12.21b and clicked the Save button.

NOTE

*The default folder, shown in the **Look In** and **Save In** text fields of the Open and Save dialog boxes when they are initially displayed, is platform dependent.*

```
1  import javax.swing.*;
2  import java.io.*;
3
4  public class FileChoosers
5  {
6      public static void main(String[] args)
7      {
8          JFileChooser fc;
9          String path;
10         int cancelApproveError;
11         File file;
12
13         //Demonstrate the ***OPEN*** file dialog box
14         fc = new JFileChooser();
15         cancelApproveError = fc.showOpenDialog(null);
16         if(cancelApproveError == JFileChooser.APPROVE_OPTION) //Open clicked
17         {
18             file = fc.getSelectedFile(); //fetches file information
19             path = file.getPath(); //returns the path and file name
```



```

20     System.out.println("The path to the file to be opened is:\n" +
21         path);
22 }
23 else if(cancelApproveError == JFileChooser.CANCEL_OPTION) //canceled
24 {
25     System.out.println("The user canceled the file open operation");
26 }
27 else //an error
28 {
29     System.out.println("An error has occurred");
30 }
31
32 //Demonstrate the ***SAVE*** file dialog box
33 fc = new JFileChooser();
34 cancelApproveError = fc.showSaveDialog(null);
35 if(cancelApproveError == JFileChooser.APPROVE_OPTION) //open clicked
36 {
37     file = fc.getSelectedFile();
38     path = file.getPath();
39     System.out.println("The path to the file to be written is:\n" +
40         path);
41 }
42 else if(cancelApproveError == JFileChooser.CANCEL_OPTION) //canceled
43 {
44     System.out.println("The user canceled the file save operation");
45 }
46 else //an error
47 {
48     System.out.println("An error has occurred");
49 }
50 }
51 }

```

Figure 12.22

The application **FileChoosers**.

Console Output:

```

The path to the file to be opened is:
C:\Users\Bill\Documents\Class Notes CS 101.docx
The path to the file to be written is:
C:\Users\Bill\Documents\Class Notes CS 210

```

Figure 12.23

The output produced by the application **FileChoosers**.

12.4.2 Color-Chooser Dialog Box

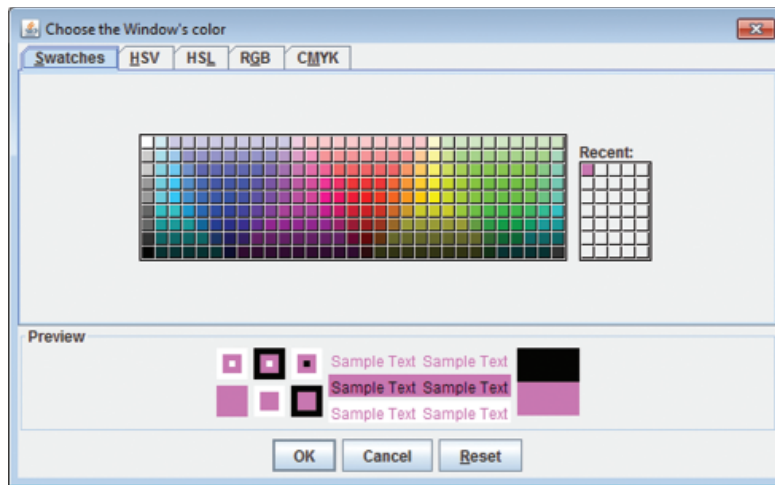
The `JColorChooser` class's static method `showDialog` is used to display a color-chooser dialog box like the one shown in Figure 12.24a. The following code fragment was used to display

it and to designate the color black as a default color choice. The second argument passed to the method is displayed at the top of the dialog box and is usually used as a user prompt.

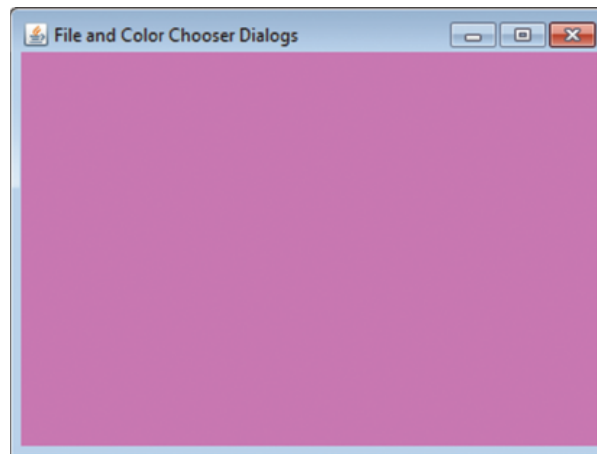
```
//Display a color chooser dialog box.
Color aColor;
aColor = JColorChooser.showDialog(null, "Choose the Window's color",
                                   Color.BLACK);
```

The user has overridden the default color choice passed to the method's third parameter by selecting the pink swatch in the middle of the box's top row of color swatches. This selection is displayed in the grid labeled *Recent* in Figure 12.24a.

The Figure 12.24b shows the window of the application `ColorChooser` whose code is shown in Figure 12.25, after the user selected the color pink from the color-chooser dialog box it displays



(a)



(b)

Figure 12.24

A `JFileChooser` dialog box and the `ColorChooser` application's program window.

and clicked OK. This application creates its window, which is an instance of a `ColorChooserWindow`, on line 8 of Figure 12.25 and makes it visible on line 10.

The constructor of the `ColorChooserWindow` class (lines 6–19 of Figure 12.26) displays a color-chooser dialog box (lines 15–16) and stores the address of the returned `Color` object that describes the color selected by the user (pink, in this case) in the variable `aColor` declared on line 12. Then, line 17 sets the background color of the `JPanel` (declared on line 11) to this color. Finally, line 18 adds the panel to the `JFrame`.

```

1  import javax.swing.*;
2
3  public class ColorChooser
4  {
5      public static void main(String[] args)
6      {
7          String title = "File and Color Chooser Dialogs";
8          ColorChooserWindow window = new ColorChooserWindow(title);
9          window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         window.setVisible(true);
11     }
12 }

```

Figure 12.25

The application `ColorChooser`.

```

1  import javax.swing.*;
2  import java.awt.Color;
3
4  public class ColorChooserWindow extends JFrame
5  {
6      public ColorChooserWindow(String title)
7      {
8          super(title);
9          setSize(400, 300);
10
11         JPanel aPanel = new JPanel();
12         Color aColor;
13
14         //Obtain the background color of the window
15         aColor = JColorChooser.showDialog(null, "Choose the Window's color",
16                                         Color.BLACK);
17         aPanel.setBackground(aColor);
18         add(aPanel);
19     }
20 }

```

Figure 12.26

The class `ColorChooserWindow`.

12.5 CHAPTER SUMMARY

The GUI components check box, radio button, combo box, and list are used to select one or more inputs from a set of valid inputs. These components are instances of the classes `JCheckBox`, `JRadioButton`, `JComboBox`, and `JList`, respectively. Ordinarily, either a set of radio buttons or a combo box is used when the choices are mutually exclusive, and a set of check boxes or a list is used when this is not the case. Radio buttons are made mutually exclusive by adding them to an instance of the `ButtonGroup` class using the class's `add` method.

When the set of input choices is large, a combo box and a list are preferred over radio buttons and check boxes because the number of items they display at one time can be specified using the methods `setMaxRowCount` for a combo box and `setVisibleCount` for a list. The method `setMaxRowCount` adds a scroll bar to a combo box, and a scroll bar can be incorporated into a list object by passing it to the constructor of a `JScrollPane` object when the scroll pane is created. This permits the user to view a large number of selection choices within a small space on the program's window.

The annotation to be displayed next to a radio button or a check box is passed to the constructor invoked to create these objects. The elements, displayed in combo boxes and lists, are placed in an array that is passed to their class's one-parameter constructor when they are created. Multiple non-sequential values in a list can be selected by clicking them while holding down the Control key (Ctrl) on the keyboard. Multiple sequential values in a list can be selected by clicking the first value in the sequence, holding down the Shift key, and clicking the last value in the sequence. The ability to select one or more values from a list is its default mode, but this can be restricted to a sequential set of values or only one value.

Normally, a set of check boxes, a set of radio buttons, a combo box, or a list is added to an instance of a `JPanel`, and the panel is then added to the window's content pane. This makes the components easier to position in the window, and the panel's border can be made visible to give the impression that the boxes or buttons it contains are part of a set. The panel's `setBorder` method and the static methods of the `BorderFactory` class can be used to display and customize a panel's border and add an informative title. A border can be placed around any component that extends the class `JComponent`, although it is most often used to put a border around a `JPanel` or a `JLabel` object.

When it is important to perform processing immediately after a check box or radio button is selected/unselected, the event handler method `itemStateChanged` (for a check box defined in the interface `ItemListener`) and `actionPerformed` (for a radio button defined in the interface `ActionListener`) is implemented and registered in the component's listener list. The event handler can invoke the `getSource` method on the argument passed to the method's parameter to determine which component was selected/unselected, and the `isSelected` method can then be invoked on the component to determine its status (selected returns `true`). The selection made in a combo box or all the selections made in a list can be determined using the methods presented in Table 12.3, which also presents a method to change the values displayed in a list (`setListData`) and to make a combo box editable (`setEditable`).

The Java API supports two types of menus, drop-down menus (`JMenuItem` instances) and pop-up menus (`JPopupMenu` instances), which are used to construct a user friendly interface that presents a group of valid input items (`JMenuItem`, `JCheckBoxMenuItem`, and `JRadioButtonMenuItem` instances) in a relatively small portion of the program's window. The annotation associated with these components is passed to their class's one-parameter constructor when they are created.

Drop-down menus are added to a menu bar (`JMenuBar` instance) whose location in the window is platform dependent. Pop-up menus are associated with other GUI components and remain invisible until the user performs a platform-dependent action (e.g., a right mouse click) on the associated component. The `add` method is invoked on a drop-down or pop-up menu object to add a menu item or a drop-down (sub) menu to them, and the `addSeparator` method is invoked on these menu objects to visually group their related elements. Hot keys can be added to menu items by invoking the `setMnemonic` method on them. When the user selects a menu item, an action event occurs. The event is serviced using the same techniques used to service action events on `JRadioButton` objects and `JButton` objects. The event handler is the method `actionPerformed` defined in the interface `ActionListener`.

In addition to these GUI components, the API Swing package also provides three dialog boxes that can be used to facilitate commonly performed user tasks: specifying the path to a file to be opened or saved and specifying a color to be used in a graphics application. The `JFileChooser` class in the Swing package provides methods to display file open and save dialog boxes, and the `JColorChooser` class provides a method to display a dialog box that contains a predefined palette of colors from which to choose a color and also provides the ability to define a custom color.

Knowledge Exercises

1. True or false:
 - a) Radio buttons are commonly used to select one or more inputs from a set of valid inputs.
 - b) Adding check boxes or radio buttons to a panel makes them easier to reposition.
 - c) Combo boxes are used to make multiple selections from a set of valid inputs.
 - d) Combo boxes or lists are used when the number of input choices is large.
 - e) A scroll bar can be associated with a list to keep the size of the component small.
 - f) A scroll bar cannot be associated with a combo box.
 - g) Elements in a list are called items, elements in a combo box are called values.
 - h) Multiple values can be selected from a list component at one time.
 - i) Multiple items can be selected from a combo box at one time.
 - j) The elements displayed in a list and or combo box are defined as an array of objects.
 - k) GUI components can be made invisible.
 - l) Combo boxes can be edited, allowing a user to type a choice into a text field.
 - m) Lists can be edited, allowing a user to type a choice into a text field.
 - n) Java supports drop-down but not pop-up menus.
 - o) Hot keys or shortcut keys can be assigned using the `setMnemonic` method.
 - p) Radio buttons and check boxes can be added to a menu.

2. Give examples of when you would use a group of check boxes and when you would use a group of radio buttons.
3. State when you would use a combo box and when you would use a list.
4. Explain how:
 - a) Radio buttons can be grouped together to make them mutually exclusive
 - b) A border is added to a GUI component such as a `JPanel`
 - c) A border's color and style can be changed from its default color and style
5. Compare and contrast the features of combo boxes and lists.
6. Explain one difference in the way that scroll bars are added to combo boxes and lists.
7. Discuss the differences between drop-down and pop-up menus.
8. What are two advantages of including menus in your applications?
9. What are mnemonics? Give an example of one.
10. Briefly explain the function of a `JFileChooser` object.
11. Briefly explain the function of a `JColorChooser` object.
12. Place the letter A, B, or C next to each of the components given below to designate the interface that defines the component's event handler method. The designations are:

A: `ItemListener`

B: `ListSelectionListener`

C: `ActionListener`

a) Check box	b) Radio button
c) Combo box	d) List
e) Menu item	
13. Place the letter A, B, or C next to each of the components given below to designate the component's event handler method. The designations are:

A: `actionPerformed`

B: `itemStateChanged`

C: `valueChanged`

a) Check box	b) Radio button
c) Combo box	d) List
e) Menu item	
14. Give the method used to:
 - a) Determine if a check box is selected
 - b) Determine if a radio button is selected
 - c) Get the item selected in a combo box
 - d) Get the index of the item selected in a combo box
 - e) Get the indices of the all the values selected in a list

- f) Get the first value selected in a list
 - g) Get all the values selected in a list
15. Give the class of each of the following components:
- a) Check box
 - b) Radio button
 - c) Combo box
 - d) List
 - e) Drop-down menu
 - f) Pop-up menu
 - g) Menu item
 - h) Menu check box
 - i) Menu radio button
16. Give the name of the method used to:
- a) Add a component to a `JPanel` object
 - b) Add a menu bar to a `JFrame` object
 - c) Add a drop-down menu to a menu bar
 - d) Associate a pop-up menu to a component
 - e) Add a hot key to a menu item
 - f) Add a check box to a menu
 - g) Add a radio button to a menu
 - h) Display a file-save dialog box
 - i) Display a choose-color dialog box

Programming Exercises

1. Design, write, and test a GUI application for the Speedy Cable Service. Include a menu to offer the user any combination of the following service options: basic, movie, sports, premium, and learning. When a user clicks the calculate button, the monthly charge for all of the services selected should be computed and output in a dialog box. (The costs are as follows: basic is \$30, movie is \$15, sports is \$20, premium is \$30, and learning is \$12). Customers can also select high or regular definition for a fee of \$10 or \$5 respectively.
2. Using a GUI design, write and test an application for a bank that offers the user the following choices: make a deposit, make a withdrawal, and check the balance of an account from a drop down menu. Dialog boxes should be used for user input and output.
3. Colorful Sports Inc. just hired you to write a pop-up menu GUI application that their customers will use to order winter clothing from a selection of three custom-colored items they sell: shirts for \$30, parkas for \$150, and gloves for \$15. The customer can select any or all of the items and can specify the size (small, medium, large, or extra large) for each item selected. Each item should have a hot key associated with it. When an item is selected, present the user with a dialog box from which he or she can select the color of the item. Provide a Place Order button that calculates and outputs the total cost to the GUI, including an 8% sales tax, the items ordered, and a swatch of the color of each item to the GUI. Include a Reset button on the GUI that clears the output and all of the selections that were made. Design the GUI.
4. Write the code and test the application described in Exercise 3.
5. Design a GUI for Sam's Sub Shop to allow users to place orders for heroes or subs. The selections should include (but are not limited to) the following items: the choice of bread (Italian, wheat, rye), one or more fillings (ham, cheese, turkey, tuna, lettuce, tomato, mayonnaise, and mustard), and one or more beverages (soda, water, and coffee). After the selections are made,

allow the user to click a button and view his or her order in a dialog box. Provide a Reset button that clears the output and all of the selections that were made.

6. Write a program to implement the design for Sam's Sub Shop in Exercise 5.
7. Design a GUI for the Tanya's Tour Trips travel agency that her customers can use to select the year, month, and day of the trip, the number of people traveling (up to four people), and a group of cities to be visited from a list of 30 cities. When the Book It button is clicked, the date of travel (mm/dd/yy) is output to the GUI along with a scrollable list of the cities to be visited. The GUI should also provide a Reset button that clears the output and all of the selections.
8. Write a program to implement the design for Tanya's Tour Trips in Exercise 7.

Enrichment

Investigate other GUI components, such as sliders, that are provided by the Java API.