CHAPTER

13

UNDERSTANDING SWING COMPONENTS

In this chapter, you will:

- Use the JFrame class
- Use additional JFrame class methods
- Use Swing event listeners
- Use JPanel class methods
- Use the JCheckBox class
- Use the ButtonGroup classes
- Create a drop-down list and combo box using the JComboBox class
- Create JScrollPanes
- Create JToolBars

Learning about inheritance has been interesting," you say to Lynn Greenbrier, "and I certainly can see how using inheritance is going to make my programming life easier. But will understanding inheritance help me create fancier applets, such as ones with frames, user lists, and choice boxes?"

"You bet it will," Lynn replies. "One reason I gave you such a thorough grounding in inheritance concepts is so it will be easier for you to learn to use JFrame-type components. All the little gadgets such as the JComboBoxes that you want to put in your JFrames are relatives, and inheritance makes it possible to use all of them. What is more important, if you have a thorough knowledge of how inheritance and components work in general, then you can adapt your knowledge to other components."

"You won't show me every component?" you ask worriedly.

"I don't have time to show you every component now," Lynn says. "Besides, there are new components that Java developers around the world are shaping right this minute."

"In other words, I can use the knowledge you give me about components, and then I can extend that knowledge to future components. That's just like inheritance," you tell Lynn. "Please explain more."

Previewing the Swing Application for Chapter 13

Event Handlers Incorporated is developing a Swing application that lets a user determine the price of an event based on several event choices. For some options, such as whether cocktails or dinner will be served, a user can select options in any combination (serve only cocktails, serve only dinner, serve both cocktails and dinner, or serve nothing). For other options, such as the dinner entrée or the entertainment, only one choice is allowed. The Chap13JDemoButtonGroup class incorporates several such devices, which you can use now.

To use the Chap13JDemoButtonGroup class:

1. Go to the command prompt for the Chapter.13 folder on your Student Disk, type **`java Chap13JDemoButtonGroup`**, and then press **[Enter]**. In the input dialog box, type **300.00** for the cost of cocktails (see Figure 13-1), and then press **[Enter]**.

Figure 13-1 Input dialog box showing the Cocktail price

2. To enter the cost of the default dinner price, type **200.00** in the input dialog (see Figure 13-2), and then press **[Enter]**.

Figure 13-2 Input dialog box showing the default Dinner price

3. To enter the event price, type **500.00** in the input dialog box (see Figure13-3), and then press **[Enter]**.

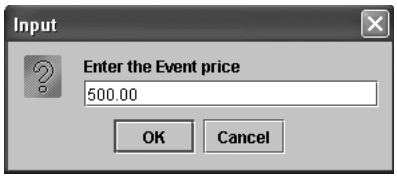

Figure 13-3 Input dialog box showing the Event price

4. The initial event cost of $500 is shown in Figure 13-4 before any boxes have been selected. Click the **Cocktails** box and **Beef** box and observe how the total price of the event changes in response to your selections. The total cost of the event, $1100 ($500 for the initial event cost, $300 for cocktails, and $300 for beef), is shown in Figure 13-5 after the cocktails and the beef are selected.
5. Close the application.

Figure 13-4 Output of Chap13JDemoButtonGroup before any boxes have been selected

Figure 13-5 Output of Chap13JDemoButtonGroup with the Cocktails and the Beef boxes selected

13

USING THE JFRAME CLASS

Computer programs are usually more user friendly (and more fun to use) when they contain graphical user interface (GUI) components such as buttons, check boxes, and menus. In Chapter 9, you learned how to add a few GUI components to a Swing applet; in this chapter, you will learn how to add several more.

You can add GUI components to either applets or applications.

You already know that you do not need to create GUI components from scratch; Java's creators packaged the Swing components to inherit from the java.awt.Container class, so you can adapt them for your purposes. You insert the import statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing GUI components and their methods.

Components are also called widgets, which stands for windows gadgets.

When you use components in a Java Swing program, you usually place them in containers. A **container** is a type of component that holds other components so you can treat a group of several components as a single entity. Usually, a container takes the form of a window that you can drag, resize, minimize, restore, and close. Containers are defined in the Container class.

As you know, all Java classes are subclasses; they all descend from the Object class. The Component class is a child of the Object class, and the Container class is a child of the Component class. Therefore, every Container Object "is a" Component, and every Component Object (including every Container) "is an" Object.

The Container class is also a parent class. Its child is the Window class. Similarly, the Frame class is a subclass of the Window class, and the JFrame class is a subclass of the Frame class, as shown in Figure 13-6.

```
java.lang.Object
  |--java.awt.Component
        |--java.awt.Container
              |--java.awt.Window
                    |--java.awt.Frame
                          |--javax.swing.JFrame
```

Figure 13-6 Relationship of the JFrame class to the Object, Component, Container, Window, and Frame superclass

Recall that the Object class is defined in the java.lang package, which is imported automatically every time you write a Java program.

The Component class is an abstract class. You learned in Chapter 12 that when you create an abstract class, you cannot create any concrete instances; instead, you create subclasses from which you create concrete instances. All GUI components, such as the buttons, text fields, and other objects with which the user interacts, are actually subclasses or extensions of the Component class. Likewise, the Container class, which descends from the Component class, is itself an abstract class. Therefore, there are no "plain" Containers; every concrete Container object is a member of a subclass of Container. The Window class, which inherits from Container, is not abstract; you can instantiate a Window object. However, Java programmers rarely use Window objects because the Window subclass Frame allows you to create more useful objects. Window objects do not have title bars or borders, but a Frame object does. As Figure 13-6 shows a JFrame"is a" Frame as well as

a Window, a Container, a Component, and an Object, and therefore inherits all the methods of the parents.

You usually create a JFrame so that you can place other objects within it for display. The JFrame class has four constructors:

- JFrame() constructs a new frame that is initially invisible.
- JFrame(GraphicsConfiguration gc) creates a JFrame in the specified GraphicsConfiguration of a screen device and a blank title.
- JFrame(String title) creates a new, initially invisible JFrame with the specified title.
- JFrame(String title, GraphicsConfiguration gc) creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen.

JFrame objects constructed with the no-argument constructor are untitled. For example, the following two statements construct two JFrames—one with the title "Hello", and another JFrame with no title:

```
JFrame firstFrame = new JFrame("Hello");
JFrame secondFrame = new JFrame();
```

Next you will create a JFrame object that appears on the screen.

To create a JFrame object:

1. Open a new file in your text editor.
2. Type the following statement to import the java.swing classes: **`import javax.swing.*;`**.
3. On the next lines, type the following class header for the JDemoFrame class and its opening curly brace:

   ```
   public class JDemoFrame
   {
   ```

4. On the next lines, type the following main() method header and its opening curly brace:

   ```
   public static void main(String[] args)
   {
   ```

5. Within the body of the main() method, enter the following code to declare a JFrame with a title, set the JFrame's size, and make the JFrame visible. If you neglect to set a JFrame's size, you will see only the title bar of the JFrame. If you neglect to make the JFrame visible, you will not see anything at all. Add two closing curly braces—one for the main() method and one for the JDemoFrame class.

   ```
   JFrame aFrame = new JFrame("This is a frame");
   aFrame.setSize(200,100);
   ```

```
    aFrame.setVisible(true);
  }
}
```

6. Save the file as **JDemoFrame.java** in the Chapter.13 folder on your Student Disk. Compile the class using the **javac** command, and then run the program using the **java** command. The output looks like Figure 13-7.

Figure 13-7 Output of the JDemoFrame program

The term frame means a generic GUI window that has a border and a title and may include buttons. You can create forms using many programming languages as well as by using Java.

The JFrame shown in Figure 13-7 resembles frames that you have seen when using different GUI programs. One reason to use similar frame objects in your programs is your program's user is already familiar with the frame environment. Users expect to see a title bar at the top of a frame that contains information (such as "This is a frame"). Users also expect to see Minimize, Maximize or Restore, and Close buttons in the frame's upper-right corner. Most users assume that they can change a frame's size by dragging its border, or reposition the frame on their screen by dragging the frame's title bar to a new location. Next you will confirm that the frame you just created has these capabilities.

To confirm that the JFrame you created has Minimize, Maximize, Restore, and dragging capabilities:

1. Run the **JDemoFrame** program again, if necessary. Click the JFrame's **Minimize** button. The JFrame minimizes to an icon on the Windows taskbar.
2. Click the JFrame's **icon** on the taskbar. The JFrame returns to its previous size.
3. Click the JFrame's **Maximize** button. The JFrame fills the screen.
4. Click the JFrame's **Restore** button to return the JFrame to its original size.
5. Position your mouse pointer on the JFrame's title bar, and then drag the JFrame to a new position on your screen.
6. Click the JFrame's **Close** button. The JFrame disappears because the default behavior is to simply hide the JFrame when the user closes the window. The cursor is left blinking in the command window until the application is closed. To close the application you must press **[Ctrl]+C**.

In Chapter 6, you learned to press [Ctrl]+C to stop a program that contains an infinite loop.

Using Additional JFrame Class Methods

When you extend the JFrame class, you inherit several useful methods. Table 13-1 lists the method header and purpose of several methods available to the JFrame class.

Table 13-1 Useful methods of the JFrame class

Method	Purpose
`void setTitle(String)`	Sets a JFrame's title
`void setSize(int, int)`	Sets a JFrame's size in pixels with the width and height as arguments
`void setSize(Dimension)`	Sets a JFrame's size using a Dimension class object by calling the Dimension(int, int) constructor that creates the object representing the specified width and height arguments
`String getTitle()`	Returns a JFrame's title
`void setResizable(boolean resizable)`	Sets the JFrame to be resizable by passing `true`, or sets the JFrame not to be resizable by passing `false` to the method
`boolean isResizable()`	Returns `true` or `false` to indicate whether the Frame is resizable
`void setVisible(boolean)`	Sets a JFrame visible using the Boolean argument `true` and invisible using the Boolean argument `false`
`void setBounds (int, int, int, int)`	Overrides the default behavior for the JFrame to be positioned in the upper-left corner of the computer's desktop. The first two arguments are the x and y position of the JFrame's upper-left corner on the desktop. The last two arguments set the width and height.

The syntax to use any of these methods is to use a JFrame object, a dot, and the method name. If you use any of these methods within a JFrame's class, then the method call to set the title is `this.setTitle("This is the title");`, or more simply, `setTitle("This is the title");`.

Earlier in this chapter you learned that when an application using a JFrame is closed, the normal behavior is for the application to keep running. To create a JFrame that ends the program when the user clicks the Close button, you can call the JFrame's setDefaultCloseOperation() method with the class variable EXIT_ON_CLOSE as an argument. There are four class variables that provide flexibility in handling the Close

operation. These class variables and their ensuing actions that can be passed as an argument to the setDefaultCloseOperation() method include:

- EXIT_ON_CLOSE exits the program when the JFrame is closed.
- DISPOSE_ON_CLOSE closes the frame, disposes of the JFrame object, and keeps running the application.
- DO_NOTHING_ON_CLOSE keeps the JFrame and continues running.
- HIDE_ON_CLOSE closes the JFrame and continues running.

When a JFrame serves as a Swing application's main user interface, the normal behavior when a JFrame is closed is for the application to keep running. To exit a program when the JFrame is closed, add the following statement to the JFrame's constructor method: `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`. For example, to set a JFrame program named JDefault to close when the JFrame's Close button is clicked, you can write the following constructor code:

```
public JDefault()
{
  super("JDefault Example")
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//other statements
}
```

Using Swing Event Listeners

Classes that respond to user events must implement an interface that deals with the events. These interfaces are called **event listeners**. Each of these listeners can handle a specific event type, and a class can implement as many event listeners as it needs. Table 13-2 lists event listeners and the types of events for which they are used.

Table 13-2 Alphabetical listing of event listeners

Listener	Type of events	Example
ActionListener	Action events	Button clicks
AdjustmentListener	Adjustment events	Scrollbar moves
FocusListener	Keyboard focus events	Textfield gains or loses focus
ItemListener	Item events	Check box changes
KeyListener	Keyboard events	Text is entered from keyboard
MouseListener	Mouse events	Mouse clicks
MouseMotionListener	Mouse movement events	All mouse movements
WindowListener	Window events	Window closes
Change Listener	Slider events	Slider moves

In Table 13-3, each component that is created is associated with one of the methods to associate a listener with it:

Table 13-3 Swing components and their associated listeners

Components	Associated listeners
JButton, JCheckBox, JComboBox, JToolbar, JTextField, and JRadioButton	addActionListener()
JScrollBar	addAdjustmentListener()
all Swing Components	addFocusListener(), addKeyListener(), addMouseListener(), and addMouseMotionListener()
JButton, JCheckBox, JComboBox and JRadioButton	addItemListener()
all JWindow and JFrame Components	addWindowListener()
JSlider	addChangeEvent()

When a user event takes place, the appropriate method is called automatically by the system. For example, when an event occurs in a program that implements ActionListener, all classes that implement ActionListener must use a method with a structure similar to the following:

```
public void actionPerformed(ActionEvent event)
{
   //method to handle the event goes here
}
```


Failure to include the actionPerfomed() method will result in a compilation error. If more than one component has an action event listener, you must figure out which component was used and code accordingly in the program. In the example above, an ActionEvent object is sent as an argument when the method is called. ActionEvent is part of the java.awt.event package, and a subclass of the EventObject class.

This ActionEvent object that is an argument to actionPerformed() contains information about the object that caused the event—for example, perhaps a button click or a scrollbar movement generated the event. You can use several methods to discover the details about the event-generating object. For example, the getSource() method returns the identity of the component where the event occurred. Consider the following code example:

```
public void actionPerformed(ActionEvent event)
{
   Object source = event.getSource();
   if(source == answerButton)
   //take some action
   else
   //take some other action
}
```

In this example, the `if` statement tests if the source is `answerButton` and if `true` takes some action. An alternate method is to use the `instanceof` keyword to check what kind of object generated the event:

```
{
   Object source = event.getSource();
   if(source instanceof JButton)
   //take some action
   else
   //take some other action
}
```

In this example, if the source of the event is a JButton, the `if` statement evaluates to `true` and some action is taken.

You first learned the `instanceof` keyword in Chapter 9 when you read about action events of Swing applets.

Using JPanel Class Methods

Because it allows other components to be added directly to the container, the JPanel, found in the JPanel class, is the simplest Swing container. The structure of the JPanel class is shown in Figure 13-8.

```
java.lang.Object
  |--java.awt.Component
        |--java.awt.Container
              |--javax.swing.JComponent
                    |--javax.swing.JPanel
```

Figure 13-8 Structure of the JPanel class

To add a component to a JPanel, you call the Container's add() method, using the component as the argument. For example, the following code creates a JButton and adds it to a JPanel:

```
JButton exit = new JButton("Exit");
JPanel panel = new JPanel();
panel.add(exit);
```

Some Swing containers such as the JFrame and JApplet do not allow components to be added directly to the container. These Swing containers require that components added to containers must be broken down into panes.

Components are added to a container's content pane using the following steps:

1. Create a JPanel object.
2. Add components to the JPanel using the add() method.
3. Call the setContentPane() method with the panel object created to set the application's content pane.

For example, the following code creates a JPanel object named pane and a JButton object named button. Then the button object is added to the pane object, using the add() method with button as the argument. The pane object is set as the content pane, using the setContentPane() method with pane as the argument.

```
JPanel pane = new JPanel();
JButton button = JButton("Exit");
pane.add(button);
setContentPane(pane);
```

Swing components have a default size for the objects that are created. A component's size can be changed using the component's setPreferredSize() method. For example, a JButton's preferred size can be changed as follows:

```
Dimension buttonSize = new Dimension(200, 20);
JButton bigButton= new JButton();
bigButton.setPreferredSize(buttonSize);
```

In the example, the preferred size of the JButton named bigButton is changed to a width of 200 pixels and a height of 20 pixels using a Dimension object named buttonSize.

Next you will create a Swing application that displays a JFrame that uses a JPanel and some JButtons. This exercise will show you that you can add JPanels to a Swing application just as easily as you can add them to a Swing applet; it will also demonstrate some JComponent class methods and associated event methods. Within the Swing application, you create a JFrame and a JPanel containing three JButtons that change captions when the user clicks them.

To create a JFrame that displays three JButtons within a JPanel:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application. The import statements used before the class header definition make the Swing components, the AWT components, and the event listener available. Note that the JChangeMessage class `implements ActionListener`. Create a Dimension object to hold the width and height dimensions. Create three JButtons with captions of "North", "Center", and "South". Then create a BorderLayout object for the component layout.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JChangeMessage extends JFrame implements
   ActionListener
```

```
{
    Dimension size = new Dimension(250,20);
    JButton b1 = new JButton("North");
    JButton b2 = new JButton("Center");
    JButton b3 = new JButton("South");
    BorderLayout aBorder = new BorderLayout();
```

2. In the JChangeMessage constructor method, set the JFrame title to "Change Message" and the default close operation to `EXIT_ON_CLOSE`. Set the preferred size of each JButton by calling the setPreferredMethod() and using the Dimension object named size as the argument. Add an ActionListener for each JButton using the keyword `this` to represent the JFrame.

```
public JChangeMessage()
{
  super("Change Message");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  b1.setPreferredSize(size);
  b2.setPreferredSize(size);
  b3.setPreferredSize(size);
  b1.addActionListener(this);
  b2.addActionListener(this);
  b3.addActionListener(this);
```

3. Create a JPanel object named pane. Set the layout for pane to BorderLayout, with the setLayout() method using the aBorder object as an argument. Add three JButtons to the JPanel named pane using the layout locations "North", "Center", and "South". Then call the setContentPane() method with pane as the argument to make it the content pane.

```
    JPanel pane = new JPanel();
    pane.setLayout(aBorder);
    pane.add("North", b1);
    pane.add("Center", b2);
    pane.add("South", b3);
    setContentPane(pane);
}
```

4. Add the following main() method that creates a new JFrame named aFrame, sizes it using the setSize() method, and sets its visible property to `true`.

```
public static void main(String[] args)
{
    JFrame aFrame = new JChangeMessage();
    aFrame.setSize(275,100);
    aFrame.setVisible(true);
}
```

5. Enter the following actionPerformed() method which will execute when the user clicks a JButton. Notice that each JButton has its own listener. To determine

which JButton is clicked, an Object source is created by calling the ActionEvent's getSource() method. An **if..else** structure is used to determine which JButton is clicked. If the source of the JButton that sent the event is the b1 JButton, the caption of b1 is set to "Event Handlers Incorporated" using the setText() method. The same procedure is applied to each of the remaining JButtons. Add a closing curly brace to end the ChangeMessage class.

```
public void actionPerformed(ActionEvent e)
  {
    Object source = e.getSource();
    if (source == b1)
        b1.setText("Event Handlers Incorporated");
    else if (source == b2)
        b2.setText("Plan With Us");
    else if (source == b3)
        b3.setText("You just relax. We'll manage the
        fuss.");
  }
}
```

6. Save the file as **JChangeMessage.java** in the Chapter.13 folder on your Student Disk. Compile the file using the **javac** command. The output is shown in Figure 13-9 before any JButtons are clicked, and again in Figure 13-10 after all the JButtons are clicked.

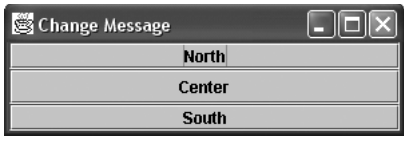

Figure 13-9 JChangeMessage program before any JButtons are clicked

Figure 13-10 JChangeMessage program after all JButtons are clicked

USING THE JCHECKBOX CLASS

A JCheckBox consists of a JLabel positioned beside a square; you can click the square to display or remove a check mark. (Usually you use a JCheckBox to allow the user to turn an option on or off.) The structure of the JCheckBox class is shown in Figure 13-11.

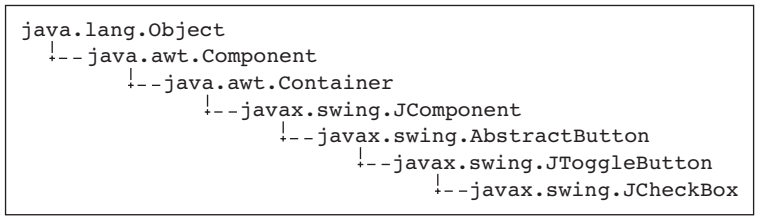

Figure 13-11 Structure of the JCheckBox class

CheckBox methods are listed in Table 13-4.

Table 13-4 JCheckBox methods

Method	Purpose
`void setLabel (String label)`	Sets the label for the JCheckBox
`String getLabel()`	Returns the JCheckBox label
`void setState (boolean condition)`	Sets the JCheckBox state to `true` for checked or `false` for unchecked
`boolean getState()`	Gets the current state (checked or unchecked) of the JCheckBox

When you construct a JCheckBox, you can choose whether to assign it a label. The following statements create two JCheckBox objects—one with a label and one without a label:

```
JCheckBox boxOne = new JCheckBox();
JCheckBox boxTwo = new JCheckBox("Click here please");
```

If you do not initialize a JCheckBox with a label, or if you want to change the label later, you can use the setLabel() method, as in `boxOne.setLabel("Check this box now");`. You can set the state of a JCheckBox with the setState() method; for example, use `boxOne.setState(false);` to insure that boxOne is unchecked. The getState() method is most useful in Boolean expressions, as in `if(boxTwo.getState()) ++votes;`, which adds one to a votes variable if boxTwo is currently checked.

Using a JCheckBox object requires using a new interface, ItemListener. Whereas ActionListener provides for mouse clicks and requires that you write an actionPerformed() method, ItemListener provides for objects whose states change from `true` to `false` and requires that you write an itemStateChanged() method. When a check box's state is changed from checked to unchecked or from unchecked to checked, the code in the itemStateChanged() method executes.

You can call the getItem() method to determine the identity of the item that generated an event. To determine whether that item was selected or deselected you use the getChange() method. This method returns an integer that will be equal to either the class variable ItemEvent.SELECTED or ItemEvent.DESELECTED. For example, in the following

itemStateChanged() method code, the getItem() method is called and returns the object named source. Then source is tested in an `if` statement to determine if it is equal to another JCheckBox object named checkBox. If the two objects are equal, then the getStateChange() method is called and returns an integer value that is assigned to the integer named select. The value of the select is compared to the class field ItemEvent.SELECTED, and if they are equal, a set of program statements executes. If they are not equal, the program statements following the `else` execute.

```
public void itemStateChanged(ItemEvent e)
  {
    Object source = e.getItem();
    if (source == checkBox)
    {
      int select = e.getStateChange();
      if(select == ItemEvent.SELECTED)
      //some statements
      else
      //other statements
    }
  }
```

Next you will create an interactive program that Event Handlers Incorporated clients can use to determine an event's price. The base price of an event is $500; serving cocktails adds $300, and serving dinner adds $200. The user can check and uncheck the cocktail and dinner check boxes to recalculate the event price.

To write a Swing application that includes two JCheckBox objects:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application that demonstrates the use of a JCheckBox. Note that the JDemoCheckBox class `implements ItemListener`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoCheckBox extends JFrame implements
   ItemListener
{
```

2. Create a FlowLayout object named flow to be used as an argument to set the layout of the program to FlowLayout. Create two JCheckBoxes named cocktailBox and dinnerBox, both unchecked, by setting their second argument to `false`. Create two JLabels to hold the headings "Event Handlers Incorporated" and "Event Price Estimate". Create a JTextField object named totPrice and a String variable named output. The JTextField is used to display the total price calculated for the event. The total price of the event is calculated from the prices of the individual event items selected and stored in the String variable named output.

```
FlowLayout flow = new FlowLayout();
JCheckBox cocktailBox = new JCheckBox("Cocktails",
   false);
```

```
JCheckBox dinnerBox = new JCheckBox("Dinner", false);
JLabel aEvent = new JLabel
   ("Event Handlers Incorporated");
JLabel ePrice = new JLabel("Event Price estimate");
JTextArea totPrice = new JTextArea(1,10);
String output;
```

3. Using the JOptionPane showInputDialog()method, accept keyboard input for the price of cocktails, dinner, and the event. Use the parseDouble() method to change the input String objects to the doubles named cocktailPrice, dinnerPrice, and totalPrice.

```
String firstPrice = JOptionPane.showInputDialog
   ("Enter the Cocktail price");
String secondPrice = JOptionPane.showInputDialog
   ("Enter the Dinner price");
String thirdPrice = JOptionPane.showInputDialog
   ("Enter the Event price");
double cocktailPrice = Double.parseDouble(firstPrice);
double dinnerPrice = Double.parseDouble(secondPrice);
double totalPrice = Double.parseDouble(thirdPrice);
```

Figure 13-12 shows what the last dialog box will look like when the program is complete.

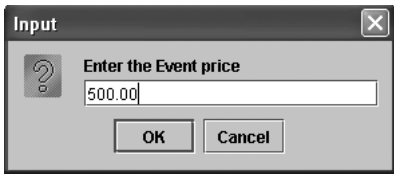

Figure 13-12 Input dialog box for the price of the event

4. Create the constructor method statement `public JDemoCheckBox()`. Add an opening curly brace, press **[Enter]**, and set the JFrame's title to "Check Box". Set the value of the JFrame's setDefaultCloseOperation to EXIT_ON_CLOSE so that the JFrame will close when the Close button is clicked. Add a JPanel named pane that will act as the content pane, and set the layout of the JPanel to FlowLayout. Add the cocktailBox and dinnerBox to the JPanel. Add the two JLabels and the JTextField to the JPanel. Note that you must add these components to the JPanel in the top-to-bottom and left-to-right order in which they are to appear. Use the setText() method to set the initial text of totPrice to the String variable thirdPrice which holds the event price captured from keyboard input. Register the cocktailBox and dinnerBox by adding an ItemListener for each, and then use the setContentPane() method to set the JPanel as the content pane for the program.

```
public JDemoCheckBox()
{
  super("Check Box");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel pane = new JPanel();
  pane.setLayout(flow);
  pane.add(cocktailBox);
  pane.add(dinnerBox);
  pane.add(aEvent);
  pane.add(ePrice);
  pane.add(totPrice);
  totPrice.setText(thirdPrice);
  cocktailBox.addItemListener(this);
  dinnerBox.addItemListener(this);
  setContentPane(pane);
}
```

5. Add the following main() method that creates a new JFrame named aFrame, sizes it using the setSize() method, and sets its visible property to `true`:

```
public static void main(String[] args)
{
 JFrame aFrame = new JDemoCheckBox();
 aFrame.setSize(200,150);
 aFrame.setVisible(true);
}
```

6. Enter the following itemStateChanged() method, which executes when the user changes the status of one of the two JCheckBoxes that are registered as ItemListeners. The base price of the event is set at $500. If the cocktail JCheckBox is checked, then the program adds the cocktail price ($300) to the event total price. If the dinner JCheckBox is checked, the program adds the dinner price ($200) to the event total price. If either of the JCheckBoxes is subsequently unchecked, the appropriate prices are subtracted from the total price. Note that the source of an ItemEvent is determined using the getItem() method. Note also that whether or not the ItemEvent object is checked is determined using the getStateChange() method. Finally, observe that the variable output holds the total price as a string. This is accomplished by assigning to output the result of concatenating an empty string " " + totalPrice which forces the result to yield a string representation.

```
public void itemStateChanged(ItemEvent check)
{
  Object source = check.getItem();
  if (source == cocktailBox)
  {
    int select = check.getStateChange();
    if(select == ItemEvent.SELECTED)
    {
```

```
            totalPrice = totalPrice + cocktailPrice;
            output = " " + totalPrice;
            totPrice.setText(output);
          }
          else if(select == ItemEvent.DESELECTED)
          {
            totalPrice = totalPrice - cocktailPrice;
            output = " " + totalPrice;
            totPrice.setText(output);
          }
        }
        if(source == dinnerBox)
        {
          int select = check.getStateChange();
          if(select == ItemEvent.SELECTED)
          {
            totalPrice = totalPrice + dinnerPrice;
            output = " " + totalPrice;
            totPrice.setText(output);
          }
          else if(select == ItemEvent.DESELECTED)
          {
            totalPrice = totalPrice - dinnerPrice;
            output = " " + totalPrice;
            totPrice.setText(output);
          }
        }
      }
    }
```

7. Save the file as **JDemoCheckBox.java** in the Chapter.13 folder on your Student Disk. Compile the file using the **javac** command. Run the program with **java JDemoCheckBox**. The initial event cost of $500 is shown in Figure 13-13 before any boxes have been selected. Click the boxes of your choice and observe how the total price of the event changes in response to your selections. The total cost of the event, $1000, is shown in Figure 13-14, with cocktails and dinner selected.

Figure 13-13 Output of JDemoCheckBox before any boxes have been selected

Figure 13-14 Output of JDemoCheckBox with both the Cocktails and the Dinner boxes selected

USING THE BUTTONGROUP CLASSES

You can group several JCheckBoxes so a user can select only one at a time. When you group JCheckBox objects, all other JCheckBoxes are automatically turned off when the user selects any one check box.

A group of JCheckBoxes is similar to a set of radio buttons that you can create using the JRadioButton class. However, when you place check boxes in a group, any number can be selected. A group of radio buttons can have, at most, one button selected at a time.

To organize check boxes into a group that allows only one at a time to be selected you must create a ButtonGroup class object. You can either create a ButtonGroup and then create the individual JCheckBoxes, or you can create the JCheckBoxes and then create the ButtonGroup. The structure of the ButtonGroup class is shown in Figure 13-15.


```
java.lang.Object
  |
  +--java.swing.ButtonGroup
```

Figure 13-15 Structure of the ButtonGroup class

To create a ButtonGroup, and then add a JCheckBox:

1. Create a ButtonGroup such as `ButtonGroup aGroup = new ButtonGroup();`.
2. Create a JCheckBox such as `JCheckBox aBox = new CheckBox();`.
3. Add `JCheckBox aBox` to `ButtonGroup aGroup` as `aGroup.add(aBox);`.

For example, if you define a ButtonGroup as `ButtonGroup favoriteStoogeGroup = new ButtonGroup();`, then you can assign an unselected JCheckBox to the group, and add the JCheckBox object to the ButtonGroup using the following code:

```
JCheckBox larryBox = new JCheckBox("Larry", false);
favoriteStoogeGroup.add(larryBox);
```

You can the use the following statement to assign `JCheckBox moeBox = new JCheckBox("Moe", true);` to the ButtonGroup favoriteStoogeGroup:

`favoriteStoogeGroup.add(moeBox);`.

If you assign the `true` state to multiple JCheckBoxes within a group, each new `true` assignment negates the previous one because only one box can be selected within a group.

You can set one of the JCheckBoxes within a group to "on" by clicking it with the mouse, or you can select a JCheckBox within a ButtonGroup with a statement such as `favoriteStoogeGroup.setSelected(larryBox);`. You can determine which, if any, of the JCheckBoxes in a ButtonGroup are selected with the isSelected() method. For example, the statement: `if(favoriteStoogeGroup.isSelected());` evaluates to `true` if the favoriteStoogeGroup is selected.

Each JCheckBox object has access to every JCheckBox class method regardless of whether the JCheckBox is part of a ButtonGroup.

Next you will add a ButtonGroup to the program that determines the price of an event for Event Handlers Incorporated. If the user wants to serve dinner at an event, the price varies based on the selected menu. Because the user can choose only one entrée (chicken, beef, or fish), it is appropriate to select the entrée using a ButtonGroup.

To add a ButtonGroup to the Event Handlers pricing program:

1. In your text editor, open the **JDemoCheckBox.java** file from the Chapter.13 folder on your Student Disk, and then save it as **JDemoButtonGroup.java**.
2. Delete the old class header and type the new class header: `public class JDemoButtonGroup extends JFrame implements ItemListener`.
3. Position your insertion point at the end of the statement `JCheckBox dinnerBox = new JCheckBox("Dinner", false);`, press **[Enter]** to open up a blank line. To add new JCheckBoxes for the remaining two entrées, beefBox and fishBox, initializing each entrée box to be unchecked add the following statements:

```
JCheckBox beefBox = new JCheckBox("Beef", false);
JCheckBox fishBox = new JCheckBox("Fish", false);
```

4. Position your insertion point at the end of the statement `double totalPrice = Double.parseDouble(thirdPrice);`, and then press **[Enter]**. Then add the following two new variables for the dinner price if the selected entrée is either beef or fish: `int beefPrice = 300, fishPrice = 500;`.

5. Delete the old JDemoCheckBox() constructor method, and then type the new constructor method as **`JDemoButtonGroup()`**. Change the title of the JFrame by changing the argument of the super() method to "Button Group": **`super("Button Group");`**

6. Position the insertion point in the JDemoButtonGroup() method at the end of the line `pane.setLayout(flow);`, and then press **[Enter]**. Type the following code to add a ButtonGroup to hold the three dinner options. Then add the dinnerBox (default dinner entrée), beefBox, and fishBox to the new dinnerGroup.

```
ButtonGroup dinnerGroup = new ButtonGroup();
dinnerGroup.add(dinnerBox);
dinnerGroup.add(beefBox);
dinnerGroup.add(fishBox);
```

7. Position your insertion point to the right of the statement `pane.add(dinnerBox);`, and then press **[Enter]**. Add the beefBox and fishBox to the pane as follows:

```
pane.add(beefBox);
pane.add(fishBox);
```

8. Position your insertion point to the right of the statement `dinnerBox.addItemListener(this);`, and then press **[Enter]** to start a new line. Enter the following code to register a listener for the beefBox and fishBox. It is necessary to register listeners for each of the JCheckBoxes, rather than a listener for the ButtonGroup.

```
beefBox.addItemListener(this);
fishBox.addItemListener(this);
```

9. Within the main() method, delete the statement `JFrame aFrame = new JDemoCheckBox();`, and then add **`JFrame aFrame = new JDemoButtonGroup();`**. Delete the statement **`aFrame.setSize(200,150);`**, and then add **`aFrame.setSize(300,150);`**.

10. Place your insertion point at the end of the closing brace of the `if(source == dinnerBox)` statement, and then press **[Enter]**. Add the following changes to the itemStateChanged() methods, which execute when the user changes the state of beefBox or fishBox:

```
if (source == beefBox)
{
     int select = check.getStateChange();
     if(select == ItemEvent.SELECTED)
     {
        totalPrice = totalPrice + beefPrice;
        output = " " + totalPrice;
        totPrice.setText(output);
     }
```

```
            else if(select == ItemEvent.DESELECTED)
            {
                totalPrice = totalPrice - beefPrice;
                output = " " + totalPrice;
                totPrice.setText(output);
            }
        }
        if (source == fishBox)
        {
            int select = check.getStateChange();
            if(select == ItemEvent.SELECTED)
            {
                 totalPrice = totalPrice + fishPrice;
                 output = " " + totalPrice;
                 totPrice.setText(output);
            }
            else if(select == ItemEvent.DESELECTED)
            {
                totalPrice = totalPrice - fishPrice;
                output = " " + totalPrice;
                totPrice.setText(output);
            }
        }
```

11. Save the file in the Chapter.13 folder on your Student Disk. Compile the program using the **javac** command, and then run it. The output should look similar to Figure 13-16 if you selected Cocktails and Beef. Note that selecting Beef added an additional $300 to the event price.

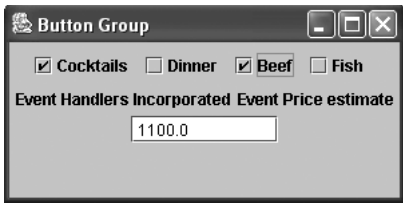

Figure 13-16 Output of the JDemoButtonGroup Swing application

Creating a Drop-Down List and Combo Box Using the JComboBox Class

The Swing package contains a single JComboBox class for picking items from a list or entering text into a field. Another option is a drop-down list, also called a choice list, which is a component that enables a single item to be chosen from a list of items. This list is usually configured to appear when the user clicks the component. A drop-down

list can also be configured to be a combo box. If the setEditable() method of the component is called with the argument `true`, it becomes a combo box that the user can use to enter text into a field.

The default behavior of the JComboBox class is to display an option; by clicking the JComboBox object, a drop-down menu that contains a list of items appears. When the user selects an item from the drop-down list, the selected item replaces the original option in the display. The structure of the JComboBox class is shown in Figure 13-17.

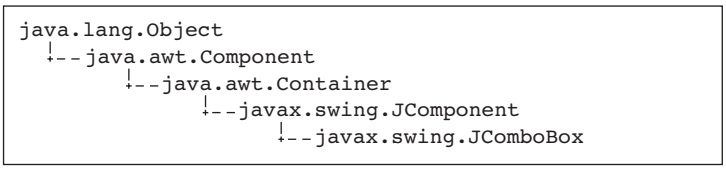

Figure 13-17 Structure of the JComboBox class

You can build a JComboBox by using a constructor with no arguments, and then adding items to the list with the addItem() method. For example, the following statements create a JComboBox object with three options:

```
JComboBox majorChoice = new JComboBox();
majorChoice.addItem("English");
majorChoice.addItem("Math");
majorChoice.addItem("Sociology");
```

Table 13-5 lists the methods you can use with a JComboBox object. Using the select() method, you choose one of the items in a JComboBox to be the initially selected item, as in `majorChoice.select("Math");`. You can extract the text of a JComboBox object and assign it to a String variable, as in `String myMajor = majorChoice.getSelectedItem();` with the getSelectedItem() method.

Table 13-5 JComboBox class methods

Method	Purpose
`String getItemAt(int)`	Returns the text of the list item at the index position specified by the integer argument. The first item of a choice list is at index position zero.
`String getSelectedItem()`	Returns the text of the currently selected item
`int getItemCount()`	Returns the number of items in the list
`int getSelectedIndex()`	Returns the item in the list that matches the given item
`void setSelectedIndex(int)`	Selects the item at the position indicated (by the integer argument)
`void setSelectedItem(Object)`	Selects the specified object in the list
`void setMaximumRowCount(int)`	Sets the maximum number of combo box rows that are displayed at one time

You can also treat the items in a JComboBox object as a zero-based array. For example, you can use the getSelectedIndex() method to determine the list position of the currently selected item. Then you can use the index to access corresponding information. For example, if a JComboBox named historyChoice has been filled with a list of historical events, such as "Declaration of Independence," "Pearl Harbor," and "Man walks on moon," then after a user chooses one of the items, you can code `int pos = historyChoice.getSelectedIndex();`. Now the variable pos holds the position of the selected item, and you can use the pos variable to access an array of dates so you can display the appropriate one. For example, if `int[] dates = {1776, 1941, 1968};`, then `dates[pos]` holds the year for the selected historical event.

Next you will create a Swing application for Event Handlers Incorporated that allows the user to choose a party favor and displays the favor price. The party favor types are none ($0), Hats ($725), Streamers ($325), Noise Makers ($125), or Balloons ($135).

To write a drop-down list using a JComboBox:

1. Open a new file in your text editor, and then add the necessary import statements followed by the first few lines of the JDemoList program. The JDemoList program uses the JComboBox to create a drop-down list. Note that JDemoList `implements ItemListener`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoList extends JFrame implements
   ItemListener
{
```

2. Create a FlowLayout object named flow to be used as an argument to set the layout of the program to FlowLayout. Create a JComboBox object named favorBox, and a JLabel object named favorList to hold the list options. Add another JLabel to hold a heading, "Event Handlers Incorporated", and a JTextField for displaying the output generated from selecting a party favor from the list.

```
FlowLayout flow = new FlowLayout();
JComboBox favorBox = new JComboBox();
JLabel favorList = new JLabel("Favor List");
JLabel aEvent = new
   JLabel("Event Handlers     Incorporated");
JTextField totPrice = new JTextField(10);
```

3. Enter the following code to create an array of integers to hold the five prices for the five party favor types. Add an integer variable to hold the price of the selected favor, a String variable to hold the output from the party favor selection, and an int variable to hold the index number of the party favor items in the party favor list.

```
int[] favorPrice = {0, 725, 325, 125, 135};
int totalPrice = 0;
```

```
String output;
int FavorNum;
```

4. Add the JDemoList() constructor, set the title of the JFrame to "JDemoList", and then set the value of the JFrame's setDefaultCloseOperation to EXIT_ON_CLOSE so that the JFrame will close when the Close button is clicked.

```
public JDemoList()
{
    super("JDemoList");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

5. Enter the text to create a JPanel named pane, and set the layout for the pane using the FlowLayout object named flow. Add the favorList object to the pane, register the ItemListener for the favorBox object, and add the names of five items to the JComboBox list:

```
    JPanel pane = new JPanel();
    pane.setLayout(flow);
    pane.add(favorList);
    favorBox.addItemListener(this);
    favorBox.addItem("None");
    favorBox.addItem("Hats");
    favorBox.addItem("Streamers");
    favorBox.addItem("Noise Makers");
    favorBox.addItem("Balloons");
```

6. Enter the following add() methods to add favorBox, aEvent, and totPrice to the pane object. Set the content pane for the application as the pane object, and then type the closing brace for the JDemoList() constructor.

```
    pane.add(favorBox);
    pane.add(aEvent);
    pane.add(totPrice);
    setContentPane(pane);
}
```

7. Add the following main() method that creates a new JDemoList object named frame, sizes it using the setSize() method, and sets its visible property to `true`:

```
public static void main(String[] arguments)
    {
        JDemoList frame = new JDemoList();
        frame.setSize(200,150);
        frame.setVisible(true);
    }
```

8. Enter the following itemStateChanged() method, which determines the index of the selected party favor type and prints the correct price based on the index. When the user clicks the JComboBox option, a drop-down list appears. When an item is selected, and the getSource() method is used to

determine the item selected, the getSelectedIndex() method identifies the index position of the selected item. The index position is assigned to the integer variable favorNum, and the price of the item selected is assigned to the totPrice variable.

```
public void itemStateChanged(ItemEvent list)
{
    Object source = list.getSource();
    if (source == favorBox)
    {
     int favorNum = favorBox.getSelectedIndex();
     totalPrice = favorPrice[favorNum];
     output = "Favor Price $" + totalPrice;
     totPrice.setText(output);
    }
}
```

9. Add the closing curly brace for the class.
10. Save the file as **JDemoList.java** in the Chapter.13 folder on your Student Disk, compile it using the **javac** command, and run it with the command **java JDemoList**. If you selected the Streamers party favor from the drop-down list, your output should look like Figure 13-18. Make another choice from the list, and then observe the new favor price.

Figure 13-18 JDemoList output with Streamers selected

It is possible to create a combo box from a drop-down list. If the JComboBox component object's method setEditable() is called with `true` as an argument, the user can make a choice by entering the text of a list item in a JTextField and pressing [Enter], rather than using the drop-down list to make a selection. Next you will modify the JDemoList program so the user can make a choice by entering the text of a list item.

To allow the user to make a choice by entering a list item's name:

1. Open the **JDemoList** program if necessary, and then change the class name to **JDemoBox**.
2. Delete the JDemoList() constructor. Add the new constructor by typing: **public JDemoBox()**. Change the title of the JFrame by changing the argument of the super() method to **super("JDemoBox");**.

3. Position your insertion point at the end of the statement `pane.add (favorBox);`, and then press **[Enter]** to start a new line. Add the statement to set the favorBox object's setEditable() method to `true` by typing: `favorBox.setEditable(true);`. This statement causes the JComboBox to behave as a combo box and allow the user to enter text.
4. Delete the statement `JDemoList frame = new JDemoList();` in the main() method. Add the new statement that creates a JFrame named frame by typing: `JDemoBox frame = new JDemoBox();`.
5. Save the file as **JDemoBox.java** in the Chapter.13 folder on your Student Disk. Compile it using the `javac` command, and then run the class with the command `java JDemoBox`. Your output should look like Figure 13-19. Select the default selection ("None") and press **[Del]** to delete it. Type **Balloons** (be sure to type a capital B) in the text box. Press **[Enter]**. Press **[Enter]** and notice the Favor Price is now included, as shown in Figure 13-20.

Figure 13-19 Output of JDemoBox program prior to Favor selection

Figure 13-20 Output of JDemoBox program after [Enter] is pressed

CREATING JSCROLLPANES

When components in a Swing GUI are bigger than the area available to display them, you can add a scroll pane container. It is common to add a JTextArea component to a scroll pane container because it allows you to use both multiple rows and columns. The JScrollPane class is a container whose methods can be used to hold any component that can be scrolled. Figure 13-21 displays the structure of the JScrollPane class.

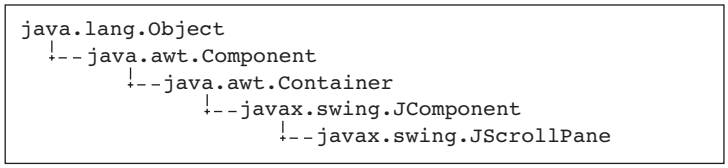

Figure 13-21 Structure of the JScrollPane class

The JScrollPane constructor takes one of four forms:

- JScrollPane() creates an empty JScrollPane where both horizontal and vertical scrollbars appear when needed.
- JScrollPane(Component) creates a JScrollPane that displays the contents of the specified component.
- JScrollPane(Component, int, int) creates a JScrollPane that displays the specified component, vertical scrollbar, and horizontal scrollbar.
- JScrollPane(int, int) creates a scroll pane with specified vertical and horizontal scrollbars.

A simple scroll pane can be created with the JScrollPane() constructor as follows:

```
JScrollPane scroll = JScrollPane();
```

The horizontal and vertical scrollbars will appear if they are needed. User control of the horizontal and vertical scrollbar configuration is achieved by using class variables of the ScrollPaneConstants class. Each of the following constants can be used for the horizontal and vertical scrollbar:

- HORIZONTAL_SCROLLBAR_AS_NEEDED
- HORIZONTAL_SCROLLBAR_ALWAYS
- HORIZONTAL_SCROLLBAR_NEVER
- VERTICAL_SCROLLBAR_AS_NEEDED
- VERTICAL_SCROLLBAR_ALWAYS
- VERTICAL_SCROLLBAR_NEVER

The following code creates a scroll pane with a vertical scrollbar and no horizontal scrollbar:

```
JScrollPane scroll = new JScrollPane(area,
  VERTICAL_SCROLLBAR_ALWAYS,
  HORIZONTAL_SCROLLBAR_NEVER);
```

A JTextArea can be created and added as a component to the scroll pane using the following code:

```
JTextArea area = new JTextArea(10,40);
JScrollPane scroll = new JScrollPane(area);
```

Notice that first a JTextArea object area is created. Then the JScrollPane(component) constructor is called with the JTextArea object named area as an argument, and the constructor creates a JScrollPane object named scroll. Text that is subsequently added to the JTextArea is viewable within the JScrollPane object created. When the added text fills more than the allotted 10 rows or 40 columns, a subsequent vertical (more than 10 rows) or horizontal (more than 40 columns) scrollbar appears automatically.

CREATING JTOOLBARS

A Swing GUI can be designed so that a user can move a toolbar from one section of a graphical user interface to another section. This type of a toolbar is called a **dockable** toolbar; the process that allows you to move and then attach the toolbar is called **docking**. A toolbar is created in Swing with the JToolBar class. The structure of the JToolBar class is shown in Figure 13-22.

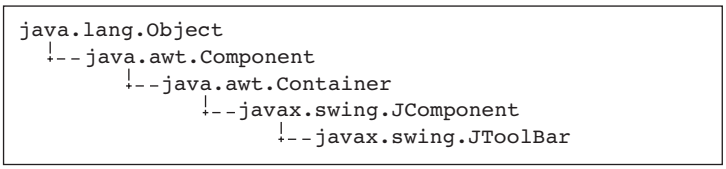

Figure 13-22 Structure of the JToolBar class

The most common toolbar is a container that groups several components, usually JButtons, into a row or column. Constructor methods for the JToolBar class include the following:

- JToolBar() creates a new toolbar that will line up components in a horizontal direction.
- JToolBar(int) creates a new toolbar with a specified orientation of horizontal or vertical.

You can use the HORIZONTAL and VERTICAL constants from the SwingConstants class to explicitly set the orientation. For example, the following statement creates a toolbar that will line up components vertically:

```
JToolBar tbar = new JToolBar(SwingConstants.VERTICAL);
```

After you create a toolbar, you can add components to it using the toolbar's add() method. Continuing the previous example, the statements `JButton b1 = new JButton();`, followed by the statement `tbar.add(b1);` add the JButton named b1 to an existing JToolBar named tbar.

To make a JToolBar **dockable**, that is, attached to the edge of a screen much like a boat is tied to a dock, the JToolBar component must use the layout manager BorderLayout.

The container should use only two of the possible north, east, south, west, and center areas. One of the two areas must be the center area.

Chapter 14 covers the different layout managers in more detail. Recall that both the FlowLayout and the BorderLayout managers have been used in this chapter and in Chapter 9.

It is common practice to use toolbar components that are labels and buttons with images. The images are created using the ImageIcon class. The ImageIcon class was introduced in Chapter 9 in conjunction with the creation of JApplets. A JButton can have both an icon and a text label. For example the following statements demonstrate how to create an ImageIcon object, and then create a JButton object with both an icon and a text label:

```
ImageIcon picture = new ImageIcon("picture.gif");
JButton both = new JButton("My Text", picture);
```

The JButton will appear with the text "My Text" and the graphic named "picture.gif".

Next you will combine the scroll pane and text area objects (described in the previous section) with a toolbar to create a dockable toolbar Swing application.

Create a dockable toolbar application with a text area:

1. Open a new file in your text editor, and then type the following first few lines of a Swing application that demonstrates the creation of a dockable toolbar. Note that the class `implements ActionListener` will be included.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoToolBar extends JFrame implements
   ActionListener
{
```

2. Create the necessary BorderLayout object required by a dockable toolbar. Create a JTextArea with space for eight lines of thirty characters each. Create a scroll pane object named scroll with three arguments—the first argument is the text area Component Object named edit, the second and third arguments are the scroll pane constants for including a vertical scrollbar and no horizontal scrollbar. The scroll pane will have a vertical scrollbar that allows the text area to hold rows of text larger than its row size.

```
BorderLayout bord = new BorderLayout();
JTextArea edit = new JTextArea(8,30);
JScrollPane scroll = new JScrollPane(edit,
  ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

3. Create a JPanel object named pane to be used as the content pane. The JPanel will hold the completed toolbar and scroll pane. Next create three ImageIcon objects to hold the three images to be displayed on the buttons. Create three JButtons with constructors that take both a text object and an ImageIcon object as arguments. Then create a JToolBar() object named bar to hold the three JButton objects.

```
JPanel pane = new JPanel();
ImageIcon image1 = new ImageIcon("dining.gif");
ImageIcon image2 = new ImageIcon("mail.gif");
ImageIcon image3 = new ImageIcon("phone.gif");
JButton b1 = new JButton("Dining", image1);
JButton b2 = new JButton("Mail", image2);
JButton b3 = new JButton("Phone", image3);
JToolBar bar = new JToolBar();
```

4. Type the JDemoToolBar() constructor and add an opening curly brace. Use the super() method with the argument "Event Handlers Toolbar" to set the title of the JFrame. Use the setDefaultCloseOperation() method to cause the JFrame to close when the Close button is clicked. Add each of the three JButtons to the toolbar using the toolbar's add() method.

```
public JDemoToolBar()
  {
    super("Event Handlers Toolbar");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    bar.add(b1);
    bar.add(b2);
    bar.add(b3);
```

5. Register each JButton with an ActionListener and set the pane object as the content pane. Then add the closing curly brace to the JDemoToolBar() constructor.

```
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
    pane.setLayout(bord);
    pane.add(bar);
    pane.add(scroll);
    pane.add(bar,BorderLayout.NORTH);
    setContentPane(pane);
  }
```

6. Add the following main() method that creates a new JFrame named tFrame, sizes it using the setSize() method, and then sets its visible property to `true`:

```
public static void main(String[] arguments)
{
 JFrame tFrame = new JDemoToolBar();
 tFrame.setSize(400,200);
```

```
    tFrame.setVisible(true);
  }
```

7. Enter the following actionPerformed() method, which executes when the user clicks one of the three JButtons that are registered as ActionListeners. When the b1 JButton is clicked, a list of corporate event choices appears in the scroll pane text area. When the b2 JButton is clicked, the Event Handlers Incorporated name and address are added to the existing scroll pane text area. When the b3 JButton is clicked, the Event Handlers Incorporated name and phone numbers are appended to the existing scroll pane text area. Note also the use of the newline character "\n" to format a new line when it is contained in the string text. At the end of the actionPerformed() method add a closing curly brace for the class.

```
public void actionPerformed(ActionEvent event)
{
      Object source = event.getSource();
      if (source == b1)
      {
        edit.append("\nOur Event Choices are:\n"
        + "Corporate, Private, and Nonprofit\n"
        + "with cocktails, dinner, and party favors\n");
      }
      else if (source == b2)
      {
        edit.append("\nOur address is:\n"
        + "Event Handlers Incorporated\n"
        + "8900 U.S. Hwy 14\n"
        + "Crystal Lake, IL 60014\n");
      }
      else if (source == b3)
      {
        edit.append("\nOur Telephone numbers are:\n"
        + "1-800-656-4576\n"
        + "575-656-5879\n");
      }
    }
}
```

8. Save the file as **JDemoToolBar.java** in the Chapter.13 folder on your Student Disk. Compile the file using the **javac** command. Run the program with **java JDemoToolBar**. The output of the Swing application before any toolbar buttons are clicked is shown in Figure 13-23.

9. Click the **waiter** icon on the toolbar. A list of corporate event choices appears in the scroll pane text area, as shown in Figure 13-24.

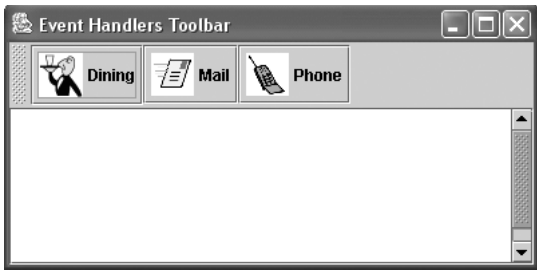

Figure 13-23 Output of the Swing application before any toolbar buttons are clicked

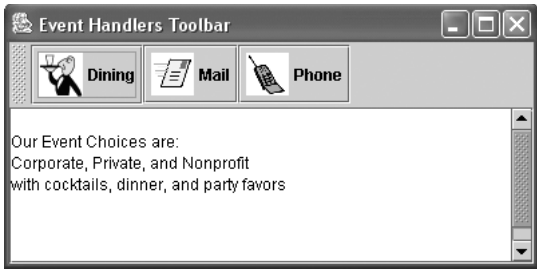

Figure 13-24 Output of a list of corporate event choices

10. Click the **letter** icon on the toolbar. The Event Handlers Incorporated name and address are appended to the existing scroll pane text area. Your output should look like Figure 13-25.

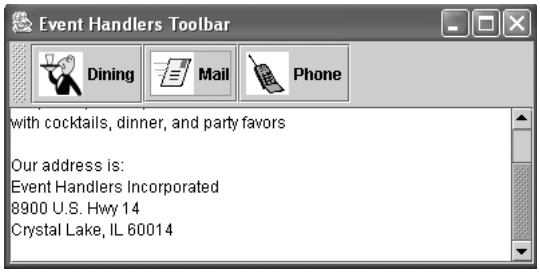

Figure 13-25 Output of Event Handlers Incorporated name and address

11. Click the **cell phone** icon on the toolbar. The Event Handlers Incorporated telephone numbers are added to the existing scroll pane text area, as shown in Figure 13-26.

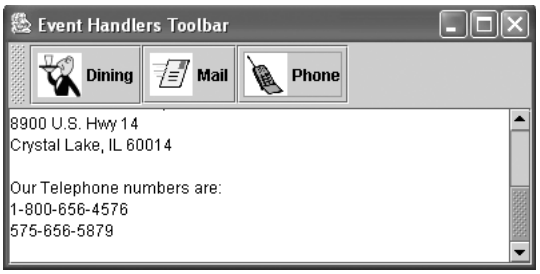

Figure 13-26 Output of Event Handlers Incorporated telephone numbers

12. Grab the toolbar by its handle (the dotted area to the left of buttons), and drag the toolbar to the bottom of the application window. When you release the toolbar, the application is rearranged using the BorderLayout manager. The new position of the toolbar is shown in Figure 13-27. You can even drag the toolbar entirely outside the application (not shown).

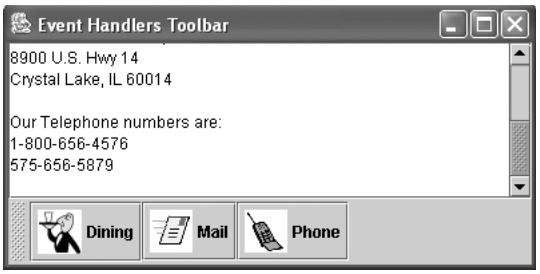

Figure 13-27 Output of JToolBar at the new border layout position

CHAPTER SUMMARY

- You insert the import statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing GUI components and their methods.
- Within the awt package, components are defined in the Component class. When you use components in a Java program, you usually place them in containers. A Container "is a" component that holds other components so you can treat a group of several components as a single entity. Containers are defined in the Container class.
- The JFrame class "is a" subclass of the awt Component class. The JFrame is the best container option for hosting Java applications. Used in conjunction with the Component class, the setSize() method allows you to set the physical size of a JFrame and the setVisible() method makes the JFrame component visible or invisible to the user. You usually create a JFrame so you can place other objects within it for display using a JPanel.

- Both the Container and Component classes are abstract classes. All of the GUI components, such as buttons, text fields, and other objects with which the user interacts, are subclasses or extensions of the Component class.
- A JFrame's action in response to a user clicking the Close button is set by passing an argument to the setDefaultCloseOperation() method placed inside the JFrame constructor method. The most common action is to close the application using the argument JFrame.EXIT_ON_CLOSE.
- Classes that respond to user events must implement an interface that deals with the events. These interfaces are called event listeners. Each of these listeners can handle a specific event type, and a class can implement as many event listeners as it needs.
- The simplest Swing container is the JPanel found in the JPanel class. To add a component to the JPanel you call the container's add() method, using the component as the argument.
- Using a JCheckBox Object requires using the interface, ItemListener. Whereas the interface ActionListener provides for mouse clicks and requires you to write an actionPerformed() method, ItemListener provides for objects whose states change from `true` to `false` and requires you to write an itemStateChanged() method.
- Since every event handling method is sent an event object of some type, the objects getSource() method can be used to determine the component that sent the event.
- A JCheckBox consists of a label positioned beside a square; you can click the square to display or remove a check mark. JCheckBox methods include those that set or get the JCheckBox's label, and set or get the JCheckBox's state of checked or unchecked.
- By using the ButtonGroup class, you can group several JCheckBoxes so that the user can select only one at a time. The methods you use with the ButtonGroup class include those that set a JCheckBox in the group to `true` and that return the currently selected JCheckBox.
- A JComboBox Object, created as a drop-down list, displays an option; clicking the option displays a menu that contains other options. When the user selects an item in the menu, the selected item replaces the original item in the display. The JComboBox methods include methods that add an item to the list, methods that select or return an item based on its name or position in the menu of the list, and a method that changes the JComboBox Object to a combo box that allows the user to enter text to select an item.
- When components in a Swing GUI are larger than the area available to display them, you can create a scroll pane container from the JScrollPane class. The JScrollPane class is a container whose methods can be used to hold any component that can be scrolled. It is common to add a JTextArea component to a scroll pane container because it allows you to use both multiple rows and columns.
- A Swing GUI can be designed so that a user can move a toolbar from one section of a graphical user interface to another section. This type of a toolbar is called a dockable toolbar, and the process is called docking. A toolbar is created in Swing with the JToolBar class and can display both text and images in the toolbar menu.

Review Questions

1. The generic name for the component type that holds other components so you can treat a group of several components as a single entity is _____.
   a. frame
   b. window
   c. container
   d. receptacle
2. To cause a JFrame to close when the Close button is clicked, you can call a JFrame's setDefaultCloseOperation() method with _____ as an argument:
   a. EXIT_ON_CLOSE
   b. DISPOSE_ON_CLOSE
   c. DO_NOTHING_ON_CLOSE
   d. HIDE_ON_CLOSE
3. A drop-down list can also be configured to be a combo box if the component's _____ method is called with the argument `true`.
   a. setSize()
   b. setVisible()
   c. setEditable()
   d. setComponent()
4. If you use an argument with a JFrame constructor, the argument represents the JFrame's _____.
   a. title
   b. size
   c. color
   d. position
5. Within an event-driven program, an object that is interested in an event is a _____.
   a. source
   b. Component
   c. Container
   d. listener
6. Which of the following statement(s) is true concerning event listeners?
   a. Each listener can handle a specific event type.
   b. A class can implement as many event listeners as it needs.

c. both a and b

d. none of the above

7. The statement _____ adds a JCheckBox object aBox to the ButtonGroup object aGroup.

   a. `aGroup.add(aBox);`

   b. `aBox.add(aGroup);`

   c. `add(aBox);`

   d. You cannot add a JCheckBox object to a ButtonGroup object.

8. A JFrame is _____.

   a. a Container

   b. a Component

   c. a Frame

   d. all of the above

9. Component classes include all of the following classes, except _____.

   a. Object

   b. JLabel

   c. ButtonGroup

   d. JCheckBox

10. Component methods include _____.

    a. getName()

    b. setComponent()

    c. isVisible()

    d. deleteComponent()

11. The method that changes the text displayed on a JButton is _____.

    a. setText()

    b. setLabel()

    c. setButton()

    d. setName()

12. Which of the following constant(s) can be used to determine whether a JCheckBox is selected or deselected?

    a. SELECTED

    b. DESELECTED

    c. ItemEvent.SELECTED

    d. both a and b

13. Which of the following statements initializes a JCheckBox with the label "Choose Me"?
    a. `JCheckBox ChooseMe = new JCheckBox();`
    b. `JCheckBox aBox = new JChooseMe();`
    c. `JCheckBox aBox = new JCheckBox("Choose Me");`
    d. `JCheckBox aBox = "Choose Me";`
14. ItemListener is a(n) _____.
    a. method
    b. Container
    c. Component
    d. interface
15. When you use ItemListener, you must write a method named _____.
    a. itemMethod()
    b. itemStateChanged()
    c. actionPerformed()
    d. listenerActivated()
16. When you group JCheckBox objects within a ButtonGroup, all other JCheckBoxes are automatically _____ when the user selects a JCheckBox.
    a. turned off
    b. turned on
    c. disabled
    d. enabled
17. Which of the following statements is true?
    a. You can create a ButtonGroup, and then create the individual JCheckBoxes.
    b. You can create JCheckBoxes, and then create their ButtonGroup.
    c. Both of the above are true.
    d. None of the above are true.
18. The getSelectedIndex() method returns _____.
    a. void
    b. a Boolean value
    c. an `int`
    d. a ButtonGroup object

19. Clicking a JComboBox object results in _____.
    a. a check mark appearing on the screen
    b. an item being dimmed
    c. the display of a menu
    d. a JButton becoming disabled
20. To make a JToolBar dockable, the JToolBar component must use a _____ layout manager.
    a. FlowLayout
    b. BorderLayout
    c. GridLayout
    d. CardLayout

EXERCISES

1. Write a program that displays a JFrame that contains the words to any well-known nursery rhyme. Save the program as **JNurseryRhyme.java** in the Chapter.13 folder of your Student Disk.
2. Create a Swing application with a JFrame that holds five labels describing reasons a customer might not buy your product (for example, "Too expensive"). Every time the user clicks a JButton, remove one of the negative reasons. Save the program as **JDemoResistance.java** in the Chapter.13 folder of your Student Disk.
3. Write a Swing application for a construction company to handle a customer's order to build a new home. Use separate ButtonGroups to allow the customer to select one of four models (the Aspen, $100,000; the Brittany, $120,000; the Colonial, $180,000; or the Dartmoor, $250,000), the number of bedrooms (two, three, or four; each bedroom adds $10,500), and a garage (no, one-, two-, or three-car; each car adds $7,775). Save the program as **JMyNewHome.java** in the Chapter.13 folder of your Student Disk.
4. a. Write a Swing application for a video store. Place the names of 10 of your favorite movies in a drop-down list. Let the user select the movie that he or she wants to rent. Charge $2.00 for most movies, and $2.50 for your personal favorite movie. Display the total rental fee. Save the program as **JVideo.java** in the Chapter.13 folder on your Student Disk.
   b. Change the drop-down list in the JVideo class to a combo box. Type the name of the movie you wish to rent. Save the program as **JVideo2.java** in the Chapter.13 folder on your Student Disk.
5. Design an order form Swing application for a pizzeria. The user makes a choice from drop-down lists, and the application displays the price. The user can choose a pizza size of small ($7), medium ($9), large ($11), or extra-large ($14), and any number of toppings. There is no additional charge for cheese, but all other toppings

add $1 each to the base price. You must offer a choice of at least five different toppings. Save the program as **JPizzaria.java** in the Chapter.13 folder on your Student Disk.

6. Write a program that allows the user to choose basketball team names that represent the team the user wants to win the NCAA. Put at least five team names in a drop-down list, allow the user to select a team, and then display the selected team. Save the program as **JBasketball.java** in the Chapter.13 folder on your Student Disk.
7. Write a program that allows the user to choose insurance options in JCheckBoxes. Use a ButtonGroup group for HMO (health maintenance organization) and PPO (preferred provider organization) options; the user can only select one option. Use regular JCheckBoxes for dental and vision options; the user can select one option. Save the program as **JInsurance.java** in the Chapter.13 folder on your Student Disk.
8. Write a program that allows the user to select options for a dormitory room. Use JCheckBoxes for the options, such as private room, Internet connection, cable TV connection, microwave, and refrigerator. Display the names of the options checked in a common text area. Save the program as **JDormRoom.java** in the Chapter.13 folder on your Student Disk.
9. Create a Swing applet with a JPanel that holds a JLabel. The JLabel hosts an icon that is larger than the Swing Applet. Create a JScrollPane to hold the JPanel so when the Swing applet is displayed the user can use both horizontal and vertical scrollbars to view the entire picture. If you wish, you can use some appropriate images found in the Chapter.13 folder of your Student Disk. Save the program as **JScrollApplet.java** in the Chapter.13 folder of your Student Disk.
10. Create a JFrame with a JPanel that holds a JLabel. The JLabel hosts an icon that is larger than the JFrame. Create a JScrollPane to hold the JPanel so when the JFrame is displayed you can use both horizontal and vertical scrollbars to view the entire picture. If you wish, you can use some appropriate images found in the Chapter.13 folder of your Student Disk. Save the program as **JScrollPicture.java** in the Chapter.13 folder of your Student Disk.
11. Create a JToolBar with both text and icons on the menu bar. When the user clicks a menu button, display another icon that has the same theme as the menu button in the scroll pane. For example, if you use a patriotic icon on the menu bar you could display a flag in the scroll pane. If you wish, you can use some appropriate images found in theChapter.13 of your Student Disk. Use at least three menu bar items. Save the program as **JImageBar.java** in the Chapter.13 folder of your Student Disk.

12. Each of the following files in the Chapter.13 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugThirteen1.java will become FixDebugThirteen1.java.
    a. DebugThirteen1.java
    b. DebugThirteen2.java
    c. DebugThirteen3.java
    d. DebugThirteen4.java

CASE PROJECT

The WebBuy Company has asked you to write a Swing application that will allow a user to compose three parts of an e-mail message. The three parts of a complete e-mail message include the "To:", "Subject:", and "Message:" text. The "To:" and "Subject:" text areas should each allow for one line. The "Message:" area should allow scrolling, if necessary, to accommodate a long message. A button is required to send the e-mail message. When the message is completed and the Send button is clicked, the program appends "E-mail has been sent!" on a new line in the message area. Save the program as **JEmail.java** in the Chapter.13 folder on your Student Disk.