CHAPTER 24

Advanced Swing GUI Applications

TOPICS

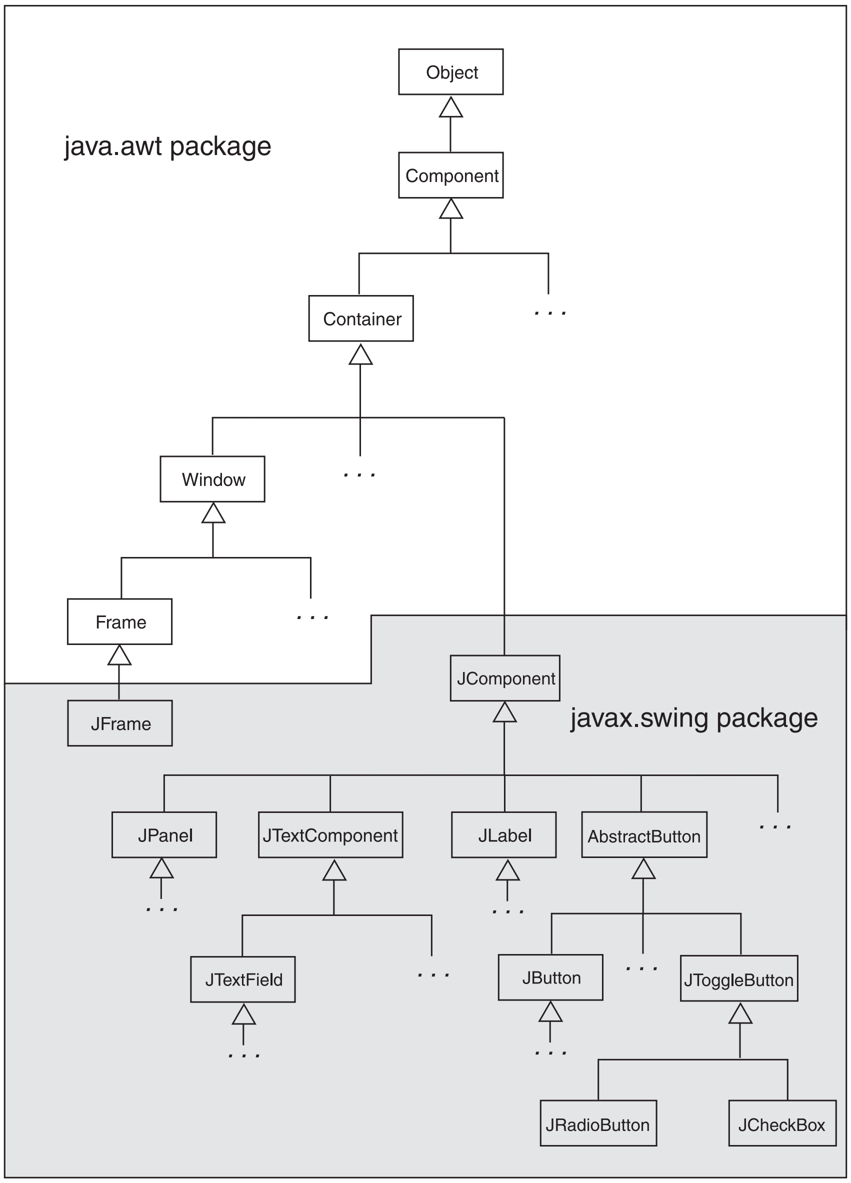
NOTE: This chapter discusses GUI development using the Swing classes. Oracle has announced that JavaFX is replacing Swing as the standard GUI library for Java. Swing will remain part of the Java API for the foreseeable future, however, so we are providing this chapter for you to use as you make the transition from Swing to JavaFX. To learn about JavaFX, see Chapters 12, 13, and 14.

24.1 The Swing and AWT Class Hierarchy

Now that you have used some of the fundamental GUI components, let's look at how they fit into the class hierarchy. Figure 24-1 shows the parts of the Swing and AWT class hierarchy that contain the `JFrame`, `JPanel`, `JLabel`, `JTextField`, `JButton`, `JRadioButton`, and `JCheckBox` classes. Because of the inheritance relationships that exist, there are many other classes in the figure as well.

The classes that are in the unshaded top part of the figure are AWT classes and are in the `java.awt` package. The classes that are in the shaded bottom part of the figure are Swing classes and are in the `javax.swing` package. Notice that all of the components we have dealt with ultimately inherit from the `Component` class.

Figure 24-1 Part of the Swing and AWT class hierarchy (Oracle Corporate Counsel)



24.2 Read-Only Text Fields

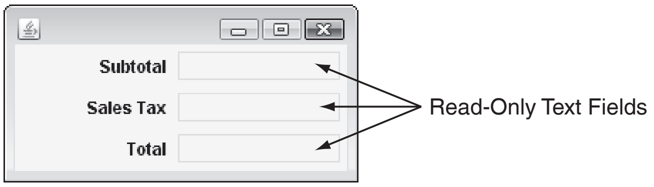
CONCEPT: A read-only text field displays text that can be changed by code in the application, but cannot be edited by the user.

A read-only text field is not a new component, but a different way to use the `JTextField` component. The `JTextField` component has a method named `setEditable`, which has the following general format:

```
setEditable(boolean editable)
```

You pass a `boolean` argument to this method. By default a text field is editable, which means that the user can enter data into it. If you call the `setEditable` method and pass `false` as the argument, then the text field becomes read-only. This means it is not editable by the user. Figure 24-2 shows a window that has three read-only text fields.

Figure 24-2 A window with three read-only text fields (Oracle Corporate Counsel)



The following code could be used to create the read-only text fields shown in the figure:

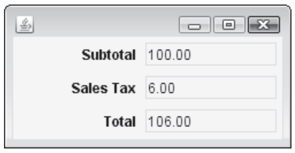
```
// Create a read-only text field for the subtotal.
JTextField subtotalField = new JTextField(10);
subtotalField.setEditable(false);

// Create a read-only text field for the sales tax.
JTextField taxField = new JTextField(10);
taxField.setEditable(false);

// Create a read-only text field for the total.
JTextField totalField = new JTextField(10);
totalField.setEditable(false);
```

A read-only text field looks like a label with a border drawn around it. You can use the `setText` method to display data inside it. Here is an example:

```
subtotalField.setText("100.00");
taxField.setText("6.00");
totalField.setText("106.00");
```

This code causes the text fields to appear as shown in Figure 24-3.

Figure 24-3 Read-only text fields with data displayed (Oracle Corporate Counsel)



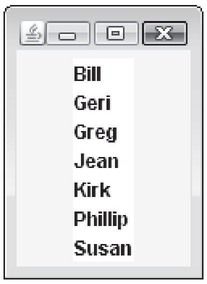
24.3 Lists

CONCEPT: A list component displays a list of items and allows the user to select an item from the list.

VideoNote The JList Component

A list is a component that displays a list of items and also allows the user to select one or more items from the list. Java provides the `JList` component for creating lists. Figure 24-4 shows an example. The `JList` component in the figure shows a list of names. At runtime, the user may select an item in the list, which causes the item to appear highlighted. In the figure, the first name is selected.

Figure 24-4 A `JList` component (Oracle Corporate Counsel)


Bill
Geri
Greg
Jean
Kirk
Phillip
Susan

When you create an instance of the `JList` class, you pass an array of objects to the constructor. Here is the general format of the constructor call:

```
JList (Object[] array)
```

The `JList` component uses the array to create the list of items. In this text we always pass an array of `String` objects to the `JList` constructor. For example, the list component shown in Figure 24-4 could be created with the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

Selection Modes

The `JList` component can operate in any of the following selection modes:

- Single Selection Mode. In this mode only one item can be selected at a time. When an item is selected, any other item that is currently selected is deselected.
- Single Interval Selection Mode. In this mode multiple items can be selected, but they must be in a single interval. An interval is a set of contiguous items.
- Multiple Interval Selection Mode. In this mode multiple items may be selected with no restrictions. This is the default selection mode.

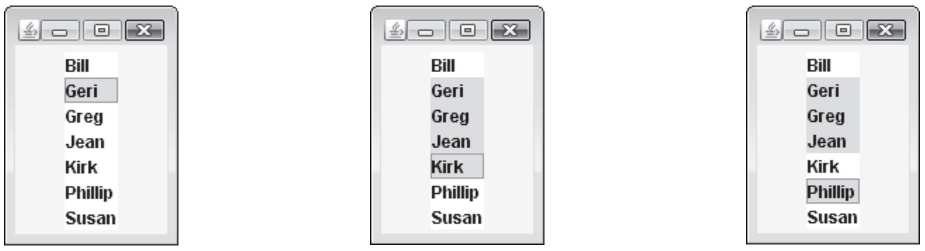
Figure 24-5 shows an example of a list in each type of selection mode.

Figure 24-5 Selection modes (Oracle Corporate Counsel)

Single selection mode allows only one item to be selected at a time.

Single interval selection mode allows a single interval of contiguous items to be selected.

Multiple interval selection mode allows multiple items to be selected with no restrictions.



The default mode is multiple interval selection. To keep our applications simple, we will use single selection mode for now. You change a `JList` component's selection mode with the `setSelectionMode` method. The method accepts an `int` argument that determines the selection mode.

The `ListSelectionModel` class, which is in the `javax.swing` package, provides the following constants that you can use as arguments to the `setSelectionMode` method:

- `ListSelectionModel.SINGLE_SELECTION`
- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
- `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`

Assuming that `nameList` references a `JList` component, the following statement sets the component to single selection mode:

```
nameList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Responding to List Events

When an item in a `JList` object is selected it generates a list selection event. You handle list selection events with a list selection listener class, which must meet the following requirements:

- It must implement the `ListSelectionListener` interface.
- It must have a method named `valueChanged`. This method must take an argument of the `ListSelectionEvent` type.

NOTE: The `ListSelectionListener` interface is in the `javax.swing.event` package, so you must have an `import` statement for that package in your source code.

Once you have written a list selection listener class, you create an object of that class and then pass it as an argument to the `JList` component's `addListSelectionListener` method. When the `JList` component generates an event, it automatically executes the `valueChanged` method of the list selection listener object, passing the event object as an argument. You will see an example in a moment.

Retrieving the Selected Item

You may use either the getSelectedValue method or the getSelectedIndex method to determine which item in a list is currently selected. The getSelectedValue method returns a reference to the item that is currently selected. For example, assume that nameList references the JList component shown earlier in Figure 24-4. The following code retrieves a reference to the name that is currently selected and assigns it to the selectedName variable:

```
String selectedName;
selectedName = (String) nameList.getSelectedValue();
```

Note that the return value of the getSelectedValue method is an Object reference. In this code we had to cast the return value to the String type in order to store it in the selectedName variable. If no item in the list is selected, the method returns null.

The getSelectedIndex method returns the index of the selected item, or −1 if no item is selected. Internally, the items that are stored in a list are numbered. Each item's number is called its index. The first item (which is the item stored at the top of the list) has the index 0, the second item has the index 1, and so forth. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the nameList component shown in Figure 24-4:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

Because the names array holds the values displayed in the namesList component, the following code could be used to determine the selected item:

```
int index;
String selectedName;
index = nameList.getSelectedIndex();
if (index != –1)
   selectedName = names[index];
```

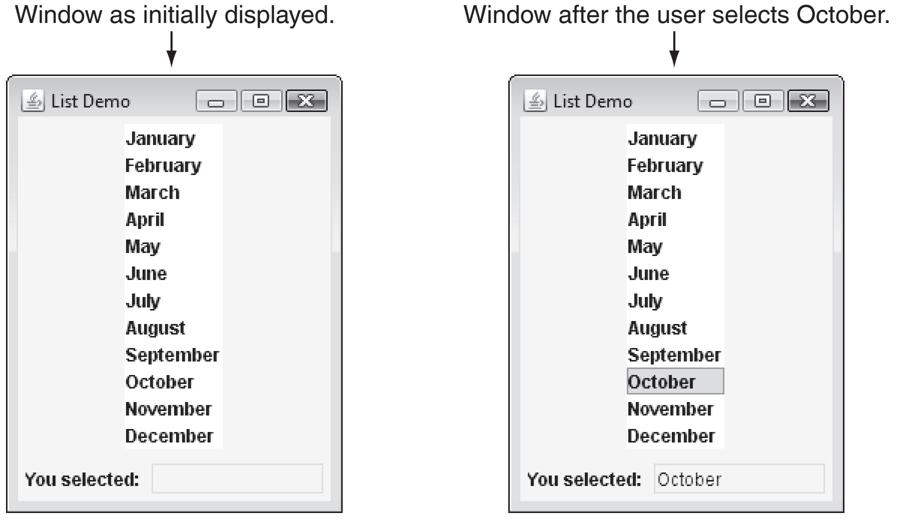
The ListWindow class shown in Code Listing 24-1 demonstrates the concepts we have discussed so far. It uses a JList component with a list selection listener. When an item is selected from the list, it is displayed in a read-only text field. The main method creates an instance of the ListWindow class, which displays the window shown on the left in Figure 24-6. After the user selects October from the list, the window appears as that shown on the right in the figure.

Code Listing 24-1 (ListWindow.java)

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

/**
   This class demonstrates the List Component.
*/
```

```

public class ListWindow extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
                           "March", "April", "May", "June", "July",
                           "August", "September", "October", "November",
                           "December" };

   /**
      Constructor
   */

   public ListWindow()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                 new ListListener());

      // Add the list to the panel.
      monthPanel.add(monthList);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
   */

   private class ListListener
                    implements ListSelectionListener
   {
      public void valueChanged(ListSelectionEvent e)
      {
         // Get the selected month.
         String selection =
               (String) monthList.getSelectedValue();

         // Put the selected month in the text field.
         selectedMonth.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ListWindow class which causes it to display
      its window.
   */

     public static void main(String[] args)
     {
          new ListWindow();
     }
}
```

Figure 24-6 Window displayed by the `ListWindow` class (Oracle Corporate Counsel)



You selected: October

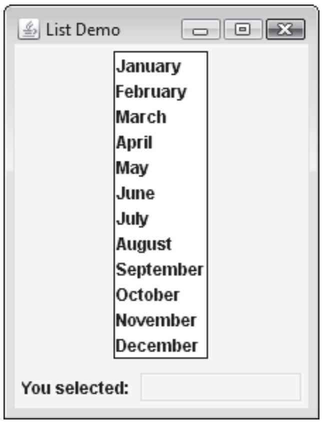
Placing a Border around a List

As with other components, you can use the `setBorder` method, which was discussed in Chapter 23, to draw a border around a `JList`. For example the following statement can be used to draw a black 1-pixel thick line border around the `monthList` component:

```
monthList.setBorder(BorderFactory.createLineBorder(Color.BLACK, 1));
```

This code will cause the list to appear as shown in Figure 24-7.

Figure 24-7 List with a line border (Oracle Corporate Counsel)



Adding a Scroll Bar to a List

By default, a list component is large enough to display all of the items it contains. Sometimes a list component contains too many items to be displayed at once, however. Most GUI applications display a scroll bar on list components that contain a large number of items. The user simply uses the scroll bar to scroll through the list of items.

List components do not automatically display a scroll bar. To display a scroll bar on a list component, you must follow the following general steps:

1. Set the number of visible rows for the list component.
2. Create a scroll pane object and add the list component to it.
3. Add the scroll pane object to any other containers, such as panels.

Let's take a closer look at how these steps can be used to apply a scroll bar to the list component created in the following code:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JList nameList = new JList(names);
```

First, we establish the size of the list component with the `JList` class's `setVisibleRowCount` method. The following statement sets the number of visible rows in the `nameList` component to three:

```
nameList.setVisibleRowCount(3);
```

This statement causes the nameList component to display only three items at a time.

Next, we create a scroll pane object and add the list component to it. A scroll pane object is a container that displays scroll bars on any component it contains. In Java we use the JScrollPane class to create a scroll pane object. We pass the object that we wish to add to the scroll pane as an argument to the JScrollPane constructor. The following statement demonstrates:

```
JScrollPane scrollPane = new JScrollPane(nameList);
```

This statement creates a JScrollPane object and adds the nameList component to it.

Next, we add the scroll pane object to any other containers that are necessary for our GUI. For example, the following code adds the scroll pane to a JPanel, which is then added to the JFrame object's content pane:

```
// Create a panel and add the scroll pane to it.
JPanel panel = new JPanel();
panel.add(scrollPane);

// Add the panel to this JFrame object's contentPane.
add(panel);
```

When the list component is displayed, it will appear as shown in Figure 24-8.

Although the list component displays only three items at a time, the user can scroll through all of the items it contains.

The ListWindowWithScroll class shown in Code Listing 24-2 is a modification of the ListWindow class. In this class, the monthList component shows only six items at a time, but displays a scroll bar. The code shown in bold is the new lines that are used to add the scroll bar to the list. The main method creates an instance of the class, which displays the window shown in Figure 24-9.

Figure 24-8 List component with a scroll bar (Oracle Corporate Counsel)

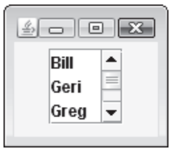
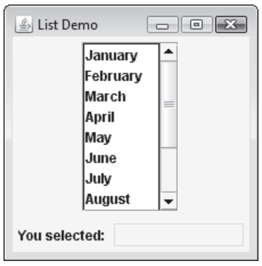


Figure 24-9 List component with scroll bars (Oracle Corporate Counsel)



Code Listing 24-2 (`ListWindowWithScroll.java`)

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

/**
   This class demonstrates the List Component.
*/

public class ListWindowWithScroll extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JList monthList;             // The months
   private JScrollPane scrollPane;      // A scroll pane
   private JTextField selectedMonth;    // The selected month
   private JLabel label;                // A message

   // The following array holds the values that will
   // be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public ListWindowWithScroll()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the month and selectedMonth panels.
      buildMonthPanel();
      buildSelectedMonthPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.CENTER);
      add(selectedMonthPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing
      the names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to single selection.
      monthList.setSelectionMode(
                 ListSelectionModel.SINGLE_SELECTION);

      // Register the list selection listener.
      monthList.addListSelectionListener(
                                  new ListListener());

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane);
   }

   /**
      The buildSelectedMonthPanel method adds an
      uneditable text field to a panel.
   */

   private void buildSelectedMonthPanel()
   {
      // Create a panel to hold the text field.
      selectedMonthPanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the text field.
      selectedMonth = new JTextField(10);

      // Make the text field uneditable.
      selectedMonth.setEditable(false);

      // Add the label and text field to the panel.
      selectedMonthPanel.add(label);
      selectedMonthPanel.add(selectedMonth);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the list.
   */

   private class ListListener
                     implements ListSelectionListener
   {
      public void valueChanged(ListSelectionEvent e)
      {
         // Get the selected month.
         String selection =
                (String) monthList.getSelectedValue();

         // Put the selected month in the text field.
         selectedMonth.setText(selection);
      }
   }

   /**
      The main method creates an instance of the
      ListWindowWithScroll class which causes it
      to display its window.
   */

     public static void main(String[] args)
     {
        new ListWindowWithScroll();
     }
}
```

NOTE: By default, when a JList component is added to a JScrollPane object, the scroll bar is only displayed when there are more items in the list than there are visible rows.

NOTE: When a JList component is added to a JScrollPane object, a border will automatically appear around the list.

Adding Items to an Existing JList Component

The JList class's setListData method allows you to store items in an existing JList component. Here is the method's general format:

```
void setListData(Object[] data)
```

The argument passed into data is an array of objects that will become the items displayed in the JList component. Any items that are currently displayed in the component will be replaced by the new items.

In addition to replacing the existing items in a list, you can use this method to add items to an empty list. You can create an empty list by passing no argument to the JList constructor. Here is an example:

```
JList nameList = new JList();
```

This statement creates an empty JList component referenced by the nameList variable. You can then add items to the list, as shown here:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
nameList.setListData(names);
```

Multiple Selection Lists

For simplicity, the previous examples used a JList component in single selection mode. Recall that the other two selection modes are single interval and multiple interval. Both of these modes allow the user to select multiple items. Let's take a closer look at each of these modes.

Single Interval Selection Mode

You put a JList component in single interval selection mode by passing the constant ListSelectionModel.SINGLE_INTERVAL_SELECTION to the component's setSelectionMode method. In single interval selection mode, single or multiple items can be selected. An interval is a set of contiguous items. (See Figure 24-5 to see an example of an interval.)

To select an interval of items, the user selects the first item in the interval by clicking on it, and then selects the last item in the interval by holding down the Shift key while clicking on it. All of the items that appear in the list from the first item through the last item are selected.

In single interval selection mode, the getSelectedValue method returns the first item in the selected interval. The getSelectedIndex method returns the index of the first item in the selected interval. To get the entire selected interval, use the getSelectedValues method. This method returns an array of objects. The array will hold the items in the selected interval. You can also use the getSelectedIndices method, which returns an array of int values. The values in the array will be the indices of all the selected items in the list.

Multiple Interval Selection Mode

You put a JList component in multiple interval selection mode by passing the constant ListSelectionModel.MULTIPLE_INTERVAL_SELECTION to the component's setSelectionMode method. In multiple interval selection mode, multiple items can be selected and the items do not have to be in the same interval. (See Figure 24-5 for an example.)

In multiple interval selection mode, the user can select single items or intervals. When the user holds down the Ctrl key while clicking on an item, it selects the item without deselecting any items that are currently selected. This allows the user to select multiple items that are not in an interval.

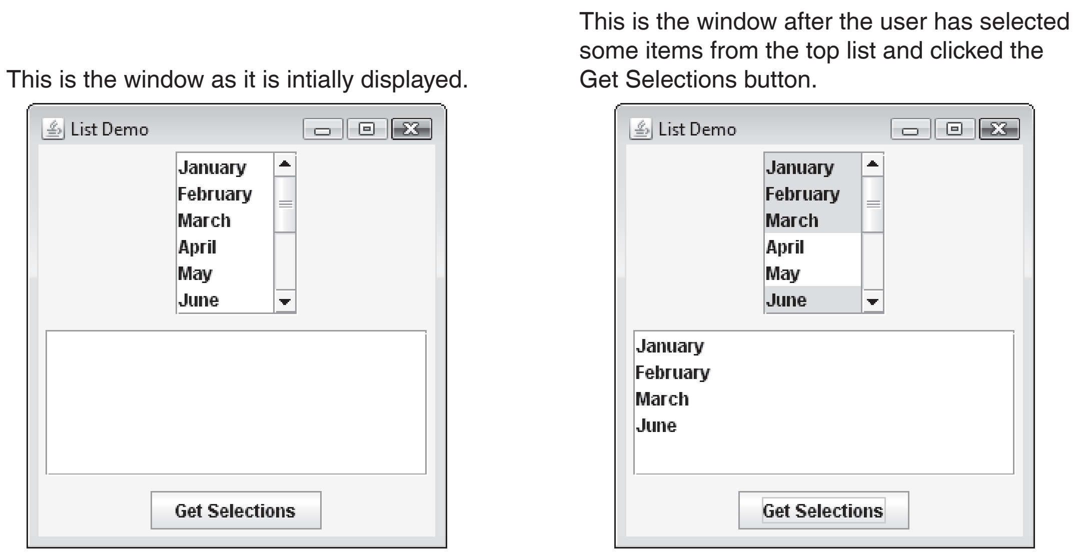
In multiple interval selection mode, the getSelectedValue method returns the first selected item. The getSelectedIndex method returns the index of the first selected item. The getSelectedValues method returns an array of objects containing the items that are selected. The getSelectedIndices method returns an int array containing the indices of all the selected items in the list.

The MultipleIntervalSelection class, shown in Code Listing 24-3, demonstrates a JList component used in multiple interval selection mode. The main method creates an instance of the class that displays the window shown on the left in Figure 24-10. When the user selects items from the top JList component and then clicks the Get Selections button, the selected items appear in the bottom JList component.

Code Listing 24-3 (MultipleIntervalSelection.java)

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

/**
   This class demonstrates the List Component in
   multiple interval selection mode.
*/

public class MultipleIntervalSelection extends JFrame
{
   private JPanel monthPanel;           // To hold components
   private JPanel selectedMonthPanel;   // To hold components
   private JPanel buttonPanel;          // To hold the button

   private JList monthList;            // To hold months
   private JList selectedMonthList;    // Selected months

   private JScrollPane scrollPane1;    // Scroll pane - first list
   private JScrollPane scrollPane2;    // Scroll pane - second list

   private JButton button;             // A button

   // The following array holds the values that
   // will be displayed in the monthList list component.
   private String[] months = { "January", "February",
             "March", "April", "May", "June", "July",
             "August", "September", "October", "November",
             "December" };

   /**
      Constructor
   */

   public MultipleIntervalSelection()
   {
      // Set the title.
      setTitle("List Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Add a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildMonthPanel();
      buildSelectedMonthsPanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(monthPanel, BorderLayout.NORTH);
      add(selectedMonthPanel,BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildMonthPanel method adds a list containing the
      names of the months to a panel.
   */

   private void buildMonthPanel()
   {
      // Create a panel to hold the list.
      monthPanel = new JPanel();

      // Create the list.
      monthList = new JList(months);

      // Set the selection mode to multiple
      // interval selection.
      monthList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);

      // Set the number of visible rows to 6.
      monthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane1 = new JScrollPane(monthList);

      // Add the scroll pane to the panel.
      monthPanel.add(scrollPane1);
   }

   /**
      The buildSelectedMonthsPanel method adds a list
      to a panel. This will hold the selected months.
   */

   private void buildSelectedMonthsPanel()
   {
      // Create a panel to hold the list.
      selectedMonthPanel = new JPanel();

      // Create the list.
      selectedMonthList = new JList();

      // Set the number of visible rows to 6.
      selectedMonthList.setVisibleRowCount(6);

      // Add the list to a scroll pane.
      scrollPane2 =
               new JScrollPane(selectedMonthList);

      // Add the scroll pane to the panel.
      selectedMonthPanel.add(scrollPane2);
   }

   /**
      The buildButtonPanel method adds a
      button to a panel.
   */

   private void buildButtonPanel()
   {
      // Create a panel to hold the list.
      buttonPanel = new JPanel();

      // Create the button.
      button = new JButton("Get Selections");

      // Add an action listener to the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected values.
         Object[] selections =
                         monthList.getSelectedValues();

         // Store the selected items in selectedMonthList.
         selectedMonthList.setListData(selections);
      }
   }

   /**
      The main method creates an instance of the
      MultipleIntervalSelection class which causes it
      to display its window.
   */

   public static void main(String[] args)
   {
      new MultipleIntervalSelection();
   }
}
```

Figure 24-10 The window displayed by the `MultipleIntervalSelection` class (Oracle Corporate Counsel)


This is the window as it is intially displayed.
This is the window after the user has selected some items from the top list and clicked the Get Selections button.
List Demo
January
February
March
April
May
June
Get Selections
List Demo
January
February
March
April
May
June
January
February
March
June
Get Selections

24.4 Combo Boxes

CONCEPT: A combo box allows the user to select an item from a drop-down list.

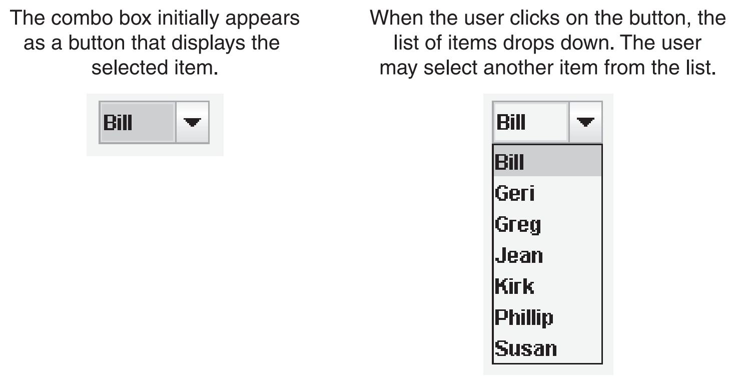
VideoNote The JComboBox Component

A combo box presents a list of items that the user may select from. Unlike a list component, a combo box presents its items in a drop-down list. You use the `JComboBox` class, which is in the `javax.swing` package, to create a combo box. You pass an array of objects that are to be displayed as the items in the drop-down list to the constructor. Here is an example:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

When displayed, the combo box created by this code will initially appear as the button shown on the left in Figure 24-11. The button displays the item that is currently selected. Notice that the first item in the list is automatically selected when the combo box is first displayed. When the user clicks the button, the drop-down list appears and the user may select another item.

Figure 24-11 A combo box (Oracle Corporate Counsel)


The combo box initially appears as a button that displays the selected item.
Bill
When the user clicks on the button, the list of items drops down. The user may select another item from the list.
Bill
Bill
Geri
Greg
Jean
Kirk
Phillip
Susan

As you can see, a combo box is a combination of two components. In the case of the combo box shown in Figure 24-11, it is the combination of a button and a list. This is where the name "combo box" comes from.

Responding to Combo Box Events

When an item in a `JComboBox` object is selected, it generates an action event. As with `JButton` components, you handle action events with an action event listener class, which must have an `actionPerformed` method. When the user selects an item in a combo box, the combo box executes its action event listener's `actionPerformed` method, passing an `ActionEvent` object as an argument.

Retrieving the Selected Item

There are two methods in the `JComboBox` class that you can use to determine which item in a combo box is currently selected: `getSelectedItem` and `getSelectedIndex`. The `getSelectedItem` method returns a reference to the item that is currently selected. For example, assume that `nameBox` references the `JComboBox` component shown earlier in Figure 24-11. The following code retrieves a reference to the name that is currently selected and assigns it to the `selectedName` variable:

```
String selectedName;
selectedName = (String) nameBox.getSelectedItem();
```

Note that the return value of the `getSelectedItem` method is an `Object` reference. In this code we had to cast the return value to the `String` type to store it in the `selectedName` variable.

The `getSelectedIndex` method returns the index of the selected item. As with `JList` components, the items that are stored in a combo box are numbered with indices that start at 0. You can use the index of the selected item to retrieve the item from an array. For example, assume that the following code was used to build the `nameBox` component shown in Figure 24-11:

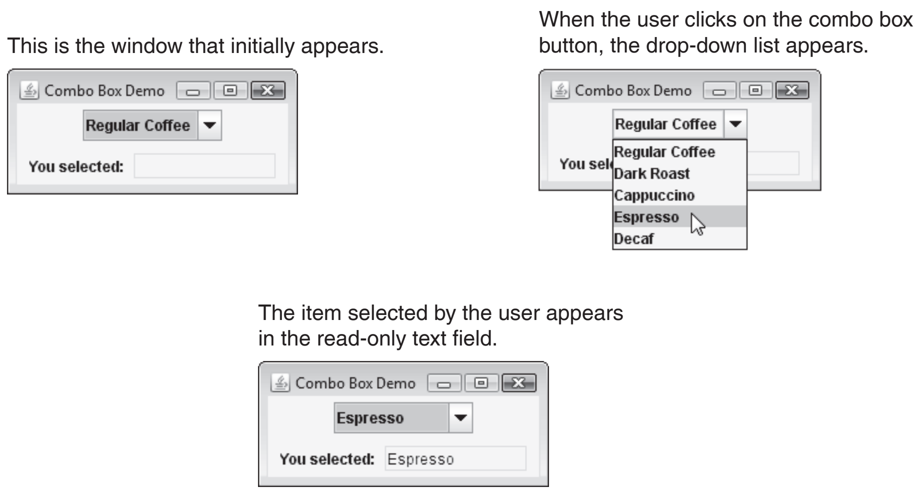
```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
```

Because the names array holds the values displayed in the namesBox component, the following code could be used to determine the selected item:

```
int index;
String selectedName;
index = nameList.getSelectedIndex();
selectedName = names[index];
```

The ComboBoxWindow class shown in Code Listing 24-4 demonstrates a combo box. It uses a JComboBox component with an action listener. When an item is selected from the combo box, it is displayed in a read-only text field. The main method creates an instance of the class, which initially displays the window shown at the top left of Figure 24-12. When the user clicks the combo box button, the drop-down list appears as shown in the top right of the figure. After the user selects Espresso from the list, the window appears as shown at the bottom of the figure.

Figure 24-12 The window displayed by the ComboBoxWindow class (Oracle Corporate Counsel)


This is the window that initially appears.
Combo Box Demo
Regular Coffee
You selected:
When the user clicks on the combo box button, the drop-down list appears.
Combo Box Demo
Regular Coffee
You sel
Regular Coffee
Dark Roast
Cappuccino
Espresso
Decaf
The item selected by the user appears in the read-only text field.
Combo Box Demo
Espresso
You selected: Espresso

Code Listing 24-4 (ComboBoxWindow.java)

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

/**
   This class demonstrates a combo box.
*/

public class ComboBoxWindow extends JFrame
{
   private JPanel coffeePanel;           // To hold components
   private JPanel selectedCoffeePanel;   // To hold components
   private JComboBox coffeeBox;          // A list of coffees
   private JLabel label;                 // Displays a message
   private JTextField selectedCoffee;    // Selected coffee

   // The following array holds the values that will
   // be displayed in the coffeeBox combo box.
   private String[] coffee = { "Regular Coffee",
                               "Dark Roast", "Cappuccino",
                               "Espresso", "Decaf"};

   /**
      Constructor
   */

   public ComboBoxWindow()
   {
      // Set the title.
      setTitle("Combo Box Demo");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildCoffeePanel();
      buildSelectedCoffeePanel();

      // Add the panels to the content pane.
      add(coffeePanel, BorderLayout.CENTER);
      add(selectedCoffeePanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildCoffeePanel method adds a combo box
      with the types of coffee to a panel.
   */

   private void buildCoffeePanel()
   {
      // Create a panel to hold the combo box.
      coffeePanel = new JPanel();

      // Create the combo box.
      coffeeBox = new JComboBox(coffee);

      // Register an action listener.
      coffeeBox.addActionListener(new ComboBoxListener());

      // Add the combo box to the panel.
      coffeePanel.add(coffeeBox);
   }

   /**
      The buildSelectedCoffeePanel method adds a
      read-only text field to a panel.
   */

   private void buildSelectedCoffeePanel()
   {
      // Create a panel to hold the components.
      selectedCoffeePanel = new JPanel();

      // Create the label.
      label = new JLabel("You selected: ");

      // Create the uneditable text field.
      selectedCoffee = new JTextField(10);
      selectedCoffee.setEditable(false);

      // Add the label and text field to the panel.
      selectedCoffeePanel.add(label);
      selectedCoffeePanel.add(selectedCoffee);
   }

   /**
      Private inner class that handles the event when
      the user selects an item from the combo box.
   */

   private class ComboBoxListener
                       implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Get the selected coffee.
         String selection =
                 (String) coffeeBox.getSelectedItem();

```

```
            // Display the selected coffee in the text field.
            selectedCoffee.setText(selection);
         }
      }

      /**
         The main method creates an instance of the
         ComboBoxWindow class, which causes it to display
         its window.
      */

      public static void main(String[] args)
      {
         new ComboBoxWindow();
      }
   }
```

Editable Combo Boxes

There are two types of combo boxes: uneditable and editable. The default type of combo box is uneditable. An uneditable combo box combines a button with a list and allows the user to select items from its list only. This is the type of combo box used in the previous examples.

An editable combo box combines a text field and a list. In addition to selecting items from the list, the user may also type input into the text field. You make a combo box editable by calling the component's `setEditable` method, passing `true` as the argument. Here is an example:

```
String[] names = { "Bill", "Geri", "Greg", "Jean",
                   "Kirk", "Phillip", "Susan" };
JComboBox nameBox = new JComboBox(names);
nameBox.setEditable(true);
```

When displayed, the combo box created by this code initially appears as shown on the left of Figure 24-13. An editable combo box appears as a text field with a small button displaying an arrow joining it. The text field displays the item that is currently selected. When the user clicks the button, the drop-down list appears, as shown in the center of the figure. The user may select an item from the list. Alternatively, the user may type a value into the text field, as shown on the right of the figure. The user is not restricted to the values that appear in the list, and may type any input into the text field.

You can use the `getSelectedItem` method to retrieve a reference to the item that is currently selected. This method returns the item that appears in the combo box's text field, so it may or may not be an item that appears in the combo box's list.

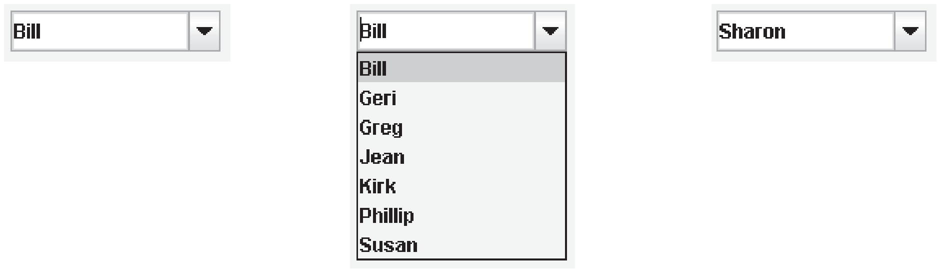
The `getSelectedIndex` method returns the index of the selected item. However, if the user has entered a value in the text field that does not appear in the list, this method will return −1.

Figure 24-13 An editable combo box (Oracle Corporate Counsel)

The editable combo box initially appears as a text field that displays the selected item. A small button with an arrow appears next to the text field.

When the user clicks on the button, the list of items drops down. The user may select another item from the list.

Alternatively, the user may type input into the text field. The user may type a value that does not appear in the list.



Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.1 How do you make a text field read-only? In code, how do you store text in a text field?

24.2 What is the index of the first item stored in a `JList` or a `JComboBox` component? If one of these components holds 12 items, what is the index of the 12th item?

24.3 How do you retrieve the selected item from a `JList` component? How do you get the index of the selected item?

24.4 How do you cause a scroll bar to be displayed with a `JList` component?

24.5 How do you retrieve the selected item from a `JComboBox` component? How do you get the index of the selected item?

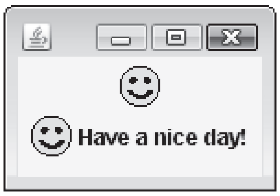
24.6 What is the difference between an uneditable and an editable combo box? Which of these is a combo box by default?

24.5 Displaying Images in Labels and Buttons

CONCEPT: Images may be displayed in labels and buttons. You use the `ImageIcon` class to get an image from a file.

In addition to displaying text in a label, you can also display an image. For example, Figure 24-14 shows a window with two labels. The top label displays a smiley face image and no text. The bottom label displays a smiley face image and text.

Figure 24-14 Labels displaying an image icon (Oracle Corporate Counsel)



To display an image, first you create an instance of the `ImageIcon` class, which can read the contents of an image file. The `ImageIcon` class is part of the `javax.swing` package. The constructor accepts a `String` argument that is the name of an image file. The supported file types are JPEG, GIF, and PNG. The name can also contain path information. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
```

This statement creates an `ImageIcon` object that reads the contents of the file *Smiley.gif*. Because no path was given, it is assumed that the file is in the current directory or folder. Here is an example that uses a path:

```
ImageIcon image = new ImageIcon("C:\\Chapter 24\\Images\\Smiley.gif");
```

Next, you can display the image in a label by passing the `ImageIcon` object as an argument to the `JLabel` constructor. Here is the general format of the constructor:

```
JLabel(Icon image)
```

The argument passed to the image parameter can be an `ImageIcon` object or any object that implements the `Icon` interface. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel(image);
```

This creates a label with an image, but no text. You can also create a label with both an image and text. An easy way to do this is to create the label with text, as usual, and then use the `JLabel` class's `setIcon` method to add an image to the label. The `setIcon` method accepts an `ImageIcon` object as its argument. Here is an example:

```
JLabel label = new JLabel("Have a nice day!");
label.setIcon(image);
```

The text will be displayed to the right of the image. The `JLabel` class also has the following constructor:

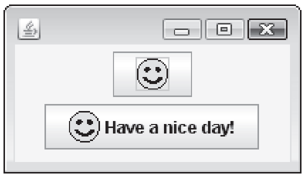
```
JLabel(String text, Icon image, int horizontalAlignment)
```

The first argument is the text to be displayed, the second argument is the image to be displayed, and the third argument is an `int` that specifies the horizontal alignment of the label contents. You should use the constants `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT` to specify the horizontal alignment. Here is an example:

```
ImageIcon image = new ImageIcon("Smiley.gif");
JLabel label = new JLabel("Have a nice day!",
                          image,
                          SwingConstants.RIGHT);
```

You can also display images in buttons, as shown in Figure 24-15.

Figure 24-15 Buttons displaying an image icon (Oracle Corporate Counsel)



The process of creating a button with an image is similar to that of creating a label with an image. You use an `ImageIcon` object to read the image file, then pass the `ImageIcon` object as an argument to the `JButton` constructor. To create a button with an image and no text, pass only the `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image, but no text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton(image);
```

To create a button with an image and text, pass a `String` and an `ImageIcon` object to the constructor. Here is an example:

```
// Create a button with an image and text.
ImageIcon image = new ImageIcon("Smiley.gif");
JButton button = new JButton("Have a nice day!", image);
```

To add an image to an existing button, pass an `ImageIcon` object to the button's `setIcon` method. Here is an example:

```
// Create a button with an image and text.
JButton button = new JButton("Have a nice day!");
ImageIcon image = new ImageIcon("Smiley.gif");
button.setIcon(image);
```

You are not limited to small graphical icons when placing images in labels or buttons. For example, the `MyCatImage` class in Code Listing 24-5 displays a digital photograph in a label when the user clicks a button. The `main` method creates an instance of the class, which displays the window shown at the left in Figure 24-16. When the user clicks the Get Image button, the window displays the image shown at the right in the figure.

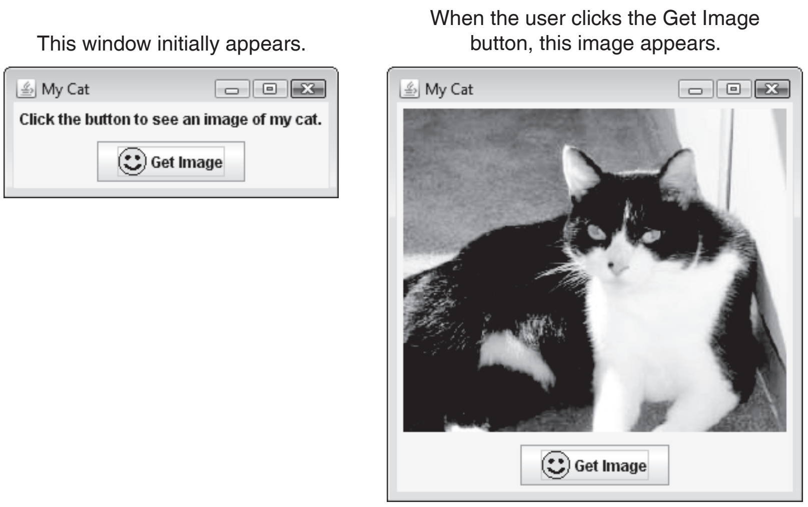
Code Listing 24-5 (`MyCatImage.java`)

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

/**
   This class demonstrates how to use an ImageIcon
   and a JLabel to display an image.
*/

```

```
public class MyCatImage extends JFrame
{
   private JPanel imagePanel;     // To hold the label
   private JPanel buttonPanel;    // To hold a button
   private JLabel imageLabel;     // To show an image
   private JButton button;        // To get an image


   /**
      Constructor
   */

   public MyCatImage()
   {
      // Set the title.
      setTitle("My Cat");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Build the panels.
      buildImagePanel();
      buildButtonPanel();

      // Add the panels to the content pane.
      add(imagePanel, BorderLayout.CENTER);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildImagePanel method adds a label to a panel.
   */

   private void buildImagePanel()
   {
      // Create a panel.
      imagePanel = new JPanel();

      // Create a label.
      imageLabel = new JLabel("Click the button to " +
                              "see an image of my cat.");

      // Add the label to the panel.
      imagePanel.add(imageLabel);
   }

   /**
      The buildButtonPanel method adds a button
      to a panel.
   */

   private void buildButtonPanel()
   {
      ImageIcon smileyImage;

      // Create a panel.
      buttonPanel = new JPanel();

      // Get the smiley face image.
      smileyImage = new ImageIcon("Smiley.gif");

      // Create a button.
      button = new JButton("Get Image");
      button.setIcon(smileyImage);

      // Register an action listener with the button.
      button.addActionListener(new ButtonListener());

      // Add the button to the panel.
      buttonPanel.add(button);
   }

   /**
      Private inner class that handles the event when
      the user clicks the button.
   */

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // Read the image file into an ImageIcon object.
         ImageIcon catImage = new ImageIcon("Cat.jpg");

         // Display the image in the label.
         imageLabel.setIcon(catImage);

         // Remove the text from the label.
         imageLabel.setText(null);
```

```
106
107             // Pack the frame again to accommodate the
108             // new size of the label.
109             pack();
110          }
111       }
112
113       /**
114          The main method creates an instance of the
115          MyCatImage class, which causes it to display
116          its window.
117       */
118       public static void main(String[] args)
119       {
120          new MyCatImage();
121       }
122 }
```

Figure 24-16 Window displayed by the `MyCatImage` class (Oracle Corporate Counsel)


This window initially appears.
When the user clicks the Get Image button, this image appears.
My Cat
Get Image
My Cat
Get Image

Click the button to see an image of my cat.

Let's take a closer look at the `MyCatImage` class. After some initial setup, the constructor calls the `buildImagePanel` method in line 34. Inside the `buildImagePanel` method, line 53 creates a `JPanel` component, referenced by the `imagePanel` variable, and then lines 56 and 57 create a `JLabel` component, referenced by the `imageLabel` variable. This is the label that will display the image when the user clicks the button. The last statement in the method, in line 60, adds the `imageLabel` component to the `imagePanel` panel.

Back in the constructor, line 35 calls the `buildButtonPanel` method, which creates the Get Image button and adds it to a panel. An instance of the `ButtonListener` inner class is also registered as the button's action listener. Let's look at the `ButtonListener` class's `actionPerformed` method. This method is executed when the user clicks the Get Image button. First, in line 99, an `ImageIcon` object is created from the file *Cat.jpg*. This file is in the same directory as the class. Next, in line 102, the image is stored in the `imageLabel` component. In line 105 the text that is currently displayed in the label is removed by passing `null` to the `imageLabel` component's `setText` method. The last statement, in line 109, calls the `JFrame` class's `pack` method. When the image was loaded into the `JLabel` component, the component resized itself to accommodate its new contents. The `JFrame` that encloses the window does not automatically resize itself, so we must call the `pack` method. This forces the `JFrame` to resize itself.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.7 How do you store an image in a `JLabel` component? How do you store both an image and text in a `JLabel` component?

24.8 How do you store an image in a `JButton` component? How do you store both an image and text in a `JButton` component?

24.9 What method do you use to store an image in an existing `JLabel` or `JButton` component?

24.6 Mnemonics and Tool Tips

CONCEPT: A mnemonic is a key that you press while holding down the Alt key to interact with a component. A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component.

Mnemonics

A mnemonic is a key on the keyboard that you press in combination with the Alt key to access a component such as a button quickly. These are sometimes referred to as shortcut keys, or hot keys. When you assign a mnemonic to a button, the user can click the button by holding down the Alt key and pressing the mnemonic key. Although users can interact with components with either the mouse or their mnemonic keys, those who are quick with the keyboard usually prefer to use mnemonic keys instead of the mouse.

You assign a mnemonic to a component through the component's `setMnemonic` method, which is inherited from the `AbstractButton` class. The method's general format is as follows:

```
void setMnemonic(int key)
```

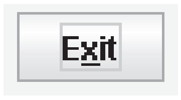
The argument that you pass to the method is an integer code that represents the key you wish to assign as a mnemonic. The `KeyEvent` class, which is in the `java.awt.event` package, has predefined constants that you can use. These constants take the form `KeyEvent.VK_x`, where `x` is a key on the keyboard. For example, to assign the A key as a mnemonic, you would use `KeyEvent.VK_A`. (The letters VK in the constants stand for "virtual key".) Here is an example of code that creates a button with the text "Exit" and assigns the X key as the mnemonic:

```
JButton exitButton = new JButton("Exit");
exitButton.setMnemonic(KeyEvent.VK_X);
```

The user may click this button by pressing a +X on the keyboard. (This means holding down the Alt key and pressing X.)

If the letter chosen as the mnemonic is in the component's text, the first occurrence of that letter will appear underlined when the component is displayed. For example, the button created with the previous code has the text "Exit". Because X was chosen as the mnemonic, the letter x will appear underlined, as shown in Figure 24-17.

Figure 24-17 Button with mnemonic X (Oracle Corporate Counsel)



If the mnemonic is a letter that does not appear in the component's text, then no letter will appear underlined.

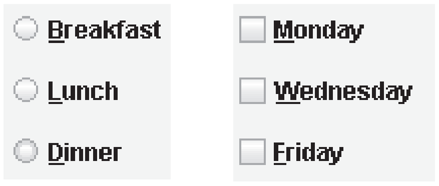
NOTE: The `KeyEvent` class also has constants for symbols. For example, the constant for the `!` symbol is `VK_EXCLAMATION_MARK`, and the constant for the `&` symbol is `VK_AMPERSAND`. See the Java API documentation for the `KeyEvent` class for a list of all the constants.

You can also assign mnemonics to radio buttons and check boxes, as shown in the following code:

```
//Create three radio buttons and assign mnemonics.
JRadioButton rb1 = new JRadioButton("Breakfast");
rb1.setMnemonic(KeyEvent.VK_B);
JRadioButton rb2 = new JRadioButton("Lunch");
rb2.setMnemonic(KeyEvent.VK_L);
JRadioButton rb3 = new JRadioButton("Dinner");
rb3.setMnemonic(KeyEvent.VK_D);

// Create three check boxes and assign mnemonics.
JCheckBox cb1 = new JCheckBox("Monday");
cb1.setMnemonic(KeyEvent.VK_M);
JCheckBox cb2 = new JCheckBox("Wednesday");
cb2.setMnemonic(KeyEvent.VK_W);
JCheckBox cb3 = new JCheckBox("Friday");
cb3.setMnemonic(KeyEvent.VK_F);
```

This code will create the components shown in Figure 24-18.

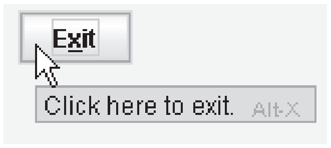
Figure 24-18 Radio buttons and check boxes with mnemonics assigned (Oracle Corporate Counsel)



Tool Tips

A tool tip is text that is displayed in a small box when the user holds the mouse cursor over a component. The box usually gives a short description of what the component does. Most GUI applications use tool tips as a way of providing immediate and concise help to the user. For example, Figure 24-19 shows a button with its tool tip displayed.

Figure 24-19 Button with tool tip displayed (Oracle Corporate Counsel)



You assign a tool tip to a component with the `setToolTipText` method, which is inherited from the `JComponent` class. Here is the method's general format:

```
void setToolTipText(String text)
```

The `String` that is passed as an argument is the text that will be displayed in the component's tool tip. For example, the following code creates the Exit button shown in Figure 24-19 and its associated tool tip:

```
JButton exitButton = new JButton("Exit");
exitButton.setToolTipText("Click here to exit.");
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.10 What is a mnemonic? How do you assign a mnemonic to a component?

24.11 What is a tool tip? How do you assign a tool tip to a component?

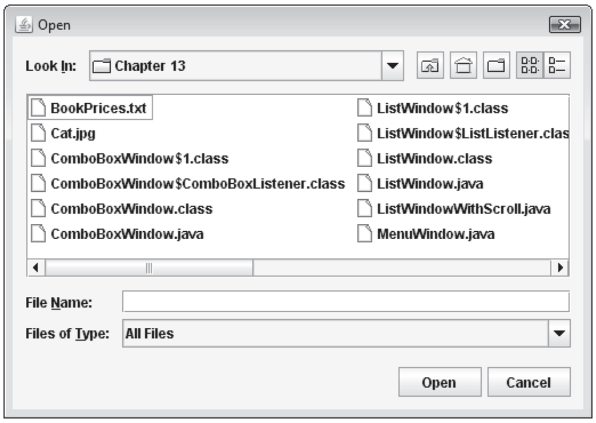
24.7 File Choosers and Color Choosers

CONCEPT: Java provides components that equip your applications with standard dialog boxes for opening files, saving files, and selecting colors.

File Choosers

A file chooser is a specialized dialog box that allows the user to browse for a file and select it. Figure 24-20 shows an example of a file chooser dialog box.

Figure 24-20 A file chooser dialog box for opening a file (Oracle Corporate Counsel)



You create an instance of the `JFileChooser` class, which is part of the `javax.swing` package, to display a file chooser dialog box. The class has several constructors. We will focus on two of them, which have the following general formats:

```
JFileChooser()
JFileChooser(String path)
```

The first constructor shown takes no arguments. This constructor uses the default directory as the starting point for all of its dialog boxes. If you are using Windows, this will probably be the "My Documents" folder under your account. If you are using UNIX, this will be your login directory. The second constructor takes a `String` argument containing a valid path. This path will be the starting point for the object's dialog boxes.

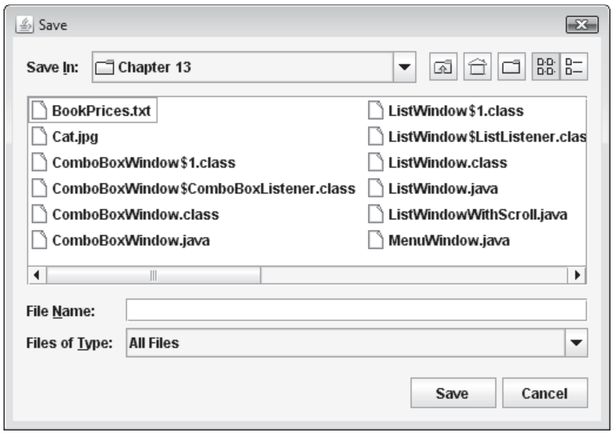
A `JFileChooser` object can display two types of predefined dialog boxes: an open file dialog box and a save file dialog box. Figure 24-20 shows an example of an open file dialog box. It lets the user browse for an existing file to open. A save file dialog box, as shown in Figure 24-21, is employed when the user needs to browse to a location to save a file. Both of these dialog boxes appear the same, except the open file dialog box displays "Open" in its title bar, and the save file dialog box displays "Save." Also, the open file dialog box has an Open button, and the save file dialog box has a Save button. There is no difference in the way they operate.

Displaying a File Chooser Dialog Box

To display an open file dialog box, use the `showOpenDialog` method. The method's general format is as follows:

```
int showOpenDialog(Component parent)
```

Figure 24-21 A save file dialog box (Oracle Corporate Counsel)



The argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component.

To display a save file dialog box, use the `showSaveDialog` method. The method's general format is as follows:

```
int showSaveDialog(Component parent)
```

Once again, the argument can be either `null` or a reference to a component. Both the `showOpenDialog` and `showSaveDialog` methods return an integer that indicates the action taken by the user to close the dialog box. You can compare the return value to one of the following constants:

- **`JFileChooser.CANCEL_OPTION`.** This return value indicates that the user clicked the Cancel button.
- **`JFileChooser.APPROVE_OPTION`.** This return value indicates that the user clicked the Open or Save button.
- **`JFileChooser.ERROR_OPTION`.** This return value indicates that an error occurred, or the user clicked the standard close button on the window to dismiss it.

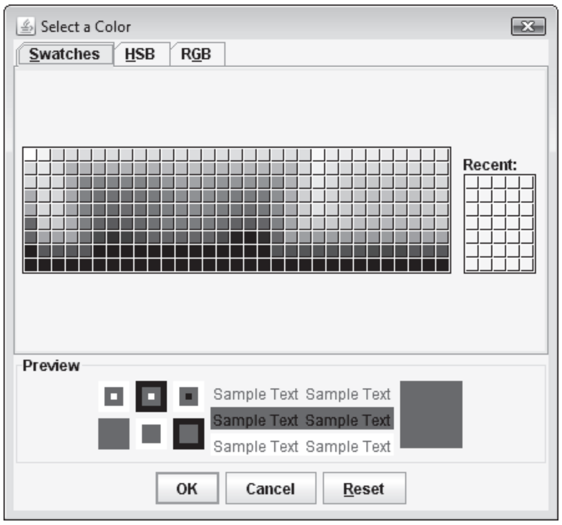
If the user selected a file, you can use the `getSelectedFile` method to determine the file that was selected. The `getSelectedFile` method returns a `File` object, which contains data about the selected file. The `File` class is part of the `java.io` package. You can use the `File` object's `getPath` method to get the path and file name as a `String`. Here is an example:

```
JFileChooser fileChooser = new JFileChooser();
int status = fileChooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION)
{
   File selectedFile = fileChooser.getSelectedFile();
   String filename = selectedFile.getPath();
   JOptionPane.showMessageDialog(null, "You selected " + filename);
}
```

Color Choosers

A color chooser is a specialized dialog box that allows the user to select a color from a predefined palette of colors. Figure 24-22 shows an example of a color chooser. By clicking the HSB tab you can select a color by specifying its hue, saturation, and brightness. By clicking the RGB tab you can select a color by specifying its red, green, and blue components.

Figure 24-22 A color chooser dialog box (Oracle Corporate Counsel)



You use the `JColorChooser` class, which is part of the `javax.swing` package, to display a color chooser dialog box. You do not create an instance of the class, however. It has a static method named `showDialog`, with the following general format:

```
Color showDialog(Component parent, String title, Color initial)
```

The first argument can be either `null` or a reference to a component. If you pass `null`, the dialog box is normally centered in the screen. If you pass a reference to a component, such as `JFrame`, the dialog box is displayed over the component. The second argument is text that is displayed in the dialog box's title bar. The third argument indicates the color that appears initially selected in the dialog box. This method returns the color selected by the user. The following code is an example. This code allows the user to select a color, and then that color is assigned as a panel's background color.

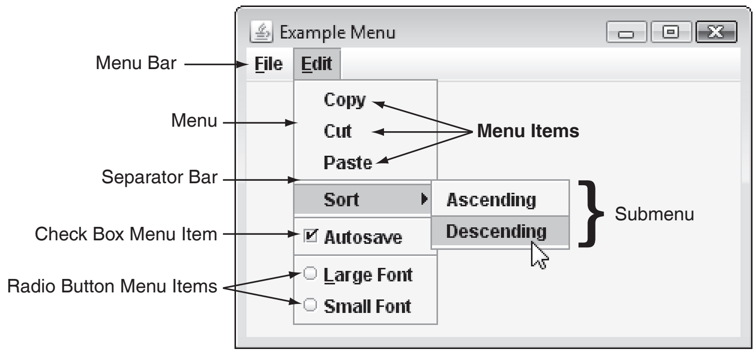
```
JPanel panel = new JPanel();
Color selectedColor;
selectedColor = JColorChooser.showDialog(null,
                "Select a Background Color", Color.BLUE);
panel.setBackground(selectedColor);
```

24.8 Menus

CONCEPT: Java provides classes for creating systems of drop-down menus. Menus can contain menu items, checked menu items, radio button menu items, and other menus.

In the GUI applications you have studied so far, the user initiates actions by clicking components such as buttons. When an application has several operations for the user to choose from, a menu system is more commonly used than buttons. A menu system is a collection of commands organized in one or more drop-down menus. Before learning how to construct a menu system, you must learn about the basic items that are found in a typical menu system. Look at the example menu system in Figure 24-23.

Figure 24-23 Example menu system (Oracle Corporate Counsel)



The menu system in the figure consists of the following items:

- **Menu Bar.** At the top of the window, just below the title bar, is a menu bar. The menu bar lists the names of one or more menus. The menu bar in Figure 24-23 shows the names of two menus: File and Edit.
- **Menu.** A menu is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In the figure, the Edit menu has been activated.
- **Menu Item.** A menu item can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check box menu item.** A check box menu item appears with a small box beside it. The item may be selected or deselected. When it is selected, a check mark appears in the box. When it is deselected, the box appears empty. Check box menu items are normally used to turn an option on or off. The user toggles the state of a check box menu item each time he or she selects it.
- **Radio button menu item.** A radio button menu item may be selected or deselected. A small circle appears beside it that is filled in when the item is selected and empty when the item is deselected. Like a check box menu item, a radio button menu item can be used to turn an option on or off. When a set of radio button menu items are grouped

with a `ButtonGroup` object, only one of them can be selected at a time. When the user selects a radio button menu item, the one that was previously selected is deselected.

- **Submenu.** A menu within a menu is called a submenu. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear. For example, in Figure 24-23, clicking on the Sort command causes a submenu to appear.
- **Separator bar.** A separator bar is a horizontal bar that is used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

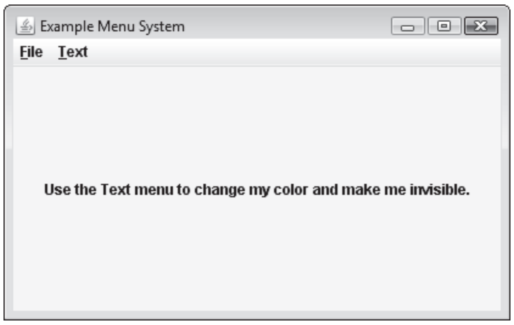
A menu system is constructed with the following classes:

- **`JMenuItem`.** Use this class to create a regular menu item. A `JMenuItem` component generates an action event when the user selects it.
- **`JCheckBoxMenuItem`.** Use this class to create a check box menu item. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JCheckBoxMenuItem` component generates an action event when the user selects it.
- **`JRadioButtonMenuItem`.** Use this class to create a radio button menu item. `JRadioButtonMenuItem` components can be grouped in a `ButtonGroup` object so that only one of them can be selected at a time. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `JRadioButtonMenuItem` component generates an action event when the user selects it.
- **`JMenu`.** Use this class to create a menu. A `JMenu` component can contain `JMenuItem`, `JCheckBoxMenuItem`, and `JRadioButton` components, as well as other `JMenu` components. A submenu is a `JMenu` component that is inside another `JMenu` component.
- **`JMenuBar`.** Use this class to create a menu bar. A `JMenuBar` object can contain `JMenu` components.

All of these classes are in the `javax.swing` package. A menu system is a `JMenuBar` component that contains one or more `JMenu` components. Each `JMenu` component can contain `JMenuItem`, `JRadioButtonMenuItem`, and `JCheckBoxMenuItem` components, as well as other `JMenu` components. The classes contain all of the code necessary to operate the menu system.

To see an example of an application that uses a menu system, we look at the `MenuWindow` class shown in Code Listing 24-6. The class displays the window shown in Figure 24-24.

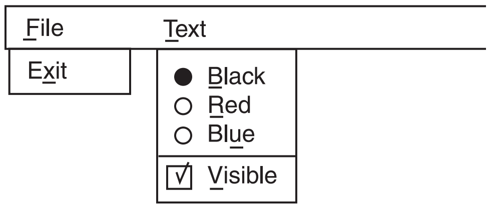
Figure 24-24 Window displayed by the `MenuWindow` class (Oracle Corporate Counsel)



Use the Text menu to change my color and make me invisible.

The class demonstrates how a label appears in different colors. Notice that the window has a menu bar with two menus: File and Text. Figure 24-25 shows a sketch of the menu system. When the user opens the Text menu, he or she can select a color using the radio button menu items and the label will change to the selected color. The Text menu also contains a Visible item, which is a check box menu item. When this item is selected (checked), the label is visible. When this item is deselected (unchecked), the label is invisible.

Figure 24-25 Sketch of the `MenuWindow` class's menu system (Oracle Corporate Counsel)



Code Listing 24-6 (`MenuWindow.java`)

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

/**
   The MenuWindow class demonstrates a menu system.
*/

public class MenuWindow extends JFrame
{
   private JLabel messageLabel;             // Displays a message
   private final int LABEL_WIDTH = 400;     // Label's width
   private final int LABEL_HEIGHT = 200;    // Label's height

   // The following will reference menu components.
   private JMenuBar menuBar;                // The menu bar
   private JMenu fileMenu;                  // The File menu
   private JMenu textMenu;                  // The Text menu
   private JMenuItem exitItem;              // To exit
   private JRadioButtonMenuItem blackItem;  // Makes text black
   private JRadioButtonMenuItem redItem;    // Makes text red
   private JRadioButtonMenuItem blueItem;   // Makes text blue
   private JCheckBoxMenuItem visibleItem;   // Toggle visibility

   /**
      Constructor
   */

```

```
public MenuWindow()
{
   // Set the title.
   setTitle("Example Menu System");

   // Specify an action for the close button.
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   // Create the messageLabel label.
   messageLabel = new JLabel("Use the Text menu to " +
             "change my color and make me invisible.",
             SwingConstants.CENTER);

   // Set the label's preferred size.
   messageLabel.setPreferredSize(
          new Dimension(LABEL_WIDTH, LABEL_HEIGHT));

   // Set the label's foreground color.
   messageLabel.setForeground(Color.BLACK);

   // Add the label to the content pane.
   add(messageLabel);

   // Build the menu bar.
   buildMenuBar();

   // Pack and display the window.
   pack();
   setVisible(true);
}

/**
   The buildMenuBar method builds the menu bar.
*/

private void buildMenuBar()
{
   // Create the menu bar.
   menuBar = new JMenuBar();

   // Create the file and text menus.
   buildFileMenu();
   buildTextMenu();

   // Add the file and text menus to the menu bar.
   menuBar.add(fileMenu);
   menuBar.add(textMenu);

```

```
      // Set the window's menu bar.
      setJMenuBar(menuBar);
   }

   /**
      The buildFileMenu method builds the File menu
      and returns a reference to its JMenu object.
   */

   private void buildFileMenu()
   {
      // Create an Exit menu item.
      exitItem = new JMenuItem("Exit");
      exitItem.setMnemonic(KeyEvent.VK_X);
      exitItem.addActionListener(new ExitListener());

      // Create a JMenu object for the File menu.
      fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);

      // Add the Exit menu item to the File menu.
      fileMenu.add(exitItem);
   }

   /**
      The buildTextMenu method builds the Text menu
      and returns a reference to its JMenu object.
   */

   private void buildTextMenu()
   {
      // Create the radio button menu items to change
      // the color of the text. Add an action listener
      // to each one.
      blackItem = new JRadioButtonMenuItem("Black", true);
      blackItem.setMnemonic(KeyEvent.VK_B);
      blackItem.addActionListener(new ColorListener());

      redItem = new JRadioButtonMenuItem("Red");
      redItem.setMnemonic(KeyEvent.VK_R);
      redItem.addActionListener(new ColorListener());

      blueItem = new JRadioButtonMenuItem("Blue");
      blueItem.setMnemonic(KeyEvent.VK_U);
      blueItem.addActionListener(new ColorListener());

      // Create a button group for the radio button items.
      ButtonGroup group = new ButtonGroup();
      group.add(blackItem);
      group.add(redItem);
      group.add(blueItem);

      // Create a check box menu item to make the text
      // visible or invisible.
      visibleItem = new JCheckBoxMenuItem("Visible", true);
      visibleItem.setMnemonic(KeyEvent.VK_V);
      visibleItem.addActionListener(new VisibleListener());

      // Create a JMenu object for the Text menu.
      textMenu = new JMenu("Text");
      textMenu.setMnemonic(KeyEvent.VK_T);

      // Add the menu items to the Text menu.
      textMenu.add(blackItem);
      textMenu.add(redItem);
      textMenu.add(blueItem);
      textMenu.addSeparator();    // Add a separator bar.
      textMenu.add(visibleItem);
   }

   /**
      Private inner class that handles the event that
      is generated when the user selects Exit from
      the File menu.
   */

   private class ExitListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }

   /**
      Private inner class that handles the event that
      is generated when the user selects a color from
      the Text menu.
   */

   private class ColorListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         if (blackItem.isSelected())
            messageLabel.setForeground(Color.BLACK);
```

```
173          else if (redItem.isSelected())
174             messageLabel.setForeground(Color.RED);
175          else if (blueItem.isSelected())
176             messageLabel.setForeground(Color.BLUE);
177       }
178    }
179 
180    /**
181       Private inner class that handles the event that
182       is generated when the user selects Visible from
183       the Text menu.
184    */
185 
186    private class VisibleListener implements ActionListener
187    {
188       public void actionPerformed(ActionEvent e)
189       {
190          if (visibleItem.isSelected())
191             messageLabel.setVisible(true);
192          else
193             messageLabel.setVisible(false);
194       }
195    }
196 
197    /**
198       The main method creates an instance of the
199       MenuWindow class, which causes it to display
200       its window.
201    */
202 
203    public static void main(String[] args)
204    {
205       MenuWindow mw = new MenuWindow();
206    }
207 }
```

Let's take a closer look at the `MenuWindow` class. Before we examine how the menu system is constructed, we should explain the code in lines 38 through 44. Lines 38 through 40 create the `messageLabel` component and align its text in the label's center. Then, in lines 43 and 44, the `setPreferredSize` method is called. The `setPreferredSize` method is inherited from the `JComponent` class, and it establishes a component's preferred size. It is called the *preferred size* because the layout manager adjusts the component's size when necessary. Normally, a label's preferred size is determined automatically, depending on its contents. We want to make this label larger, however, so the window will be larger when it is packed around the label.

The `setPreferredSize` method accepts a `Dimension` object as its argument. A `Dimension` object specifies a component's width and height. The first argument to the `Dimension` class constructor is the component's width, and the second argument is the component's height. In this class, the LABEL_WIDTH and LABEL_HEIGHT constants are defined with the values 400 and 200 respectively. So, this statement sets the label's preferred size to 400 pixels wide by 200 pixels high. (The Dimension class is part of the java.awt package.) Notice from Figure 24-24 that this code does not affect the size of the text that is displayed in the label, only the size of the label component.

To create the menu system, the constructor calls the buildMenuBar method in line 53. Inside this method, the statement in line 67 creates a JMenuBar component and assigns its address to the menuBar variable. The JMenuBar component acts as a container for JMenu components. The menu bar in this application has two menus: File and Text.

Next, the statement in line 70 calls the buildFileMenu method. The buildFileMenu method creates the File menu, which has only one item: Exit. The statement in line 89 creates a JMenuItem component for the Exit item, which is referenced by the exitItem variable. The String that is passed to the JMenuItem constructor is the text that will appear on a menu for this menu item. The statement in line 90 assigns the x key as a mnemonic to the exitItem component. Then, line 91 creates an action listener for the component (an instance of ExitListener, a private inner class), which causes the application to end.

Next, line 94 creates a JMenu object for the File menu. Notice that the name of the menu is passed as an argument to the JMenu constructor. Line 95 assigns the F key to the File menu as a mnemonic. The last statement in the buildFileMenu method, in line 98, adds exitItem to the fileMenu component.

Back in the buildMenuBar method, the statement in line 71 calls the buildTextMenu method. The buildTextMenu method builds the Text menu, which has three radio button menu items (Black, Red, and Blue), a separator bar, and a check box menu item (Visible). The code in lines 111 through 121 creates the radio button menu items, assigns mnemonic keys to them, and adds an action listener to each.

The JRadioButtonItem constructor accepts a String argument, which is the menu item's text. By default, a radio button menu item is not initially selected. The constructor can also accept an optional second argument, which is a boolean value indicating whether the item should be initially selected. Notice that in line 111, true is passed as the second argument to the JRadioButtonItem constructor. This causes the Black menu item to be initially selected.

Next, in lines 124 through 127, a button group is created and the radio button menu items are added to it. As with JRadioButton components, JRadioButtonMenuItem components may be grouped in a ButtonGroup object. As a result, only one of the grouped menu items may be selected at a time. When one is selected, any other menu item in the group is deselected.

Next, the Visible item, a check box menu item, is created in line 131. Notice that true is passed as the second argument to the constructor. This causes the item to be initially selected. A mnemonic key is assigned in line 132, and an action listener is added to the component in line 133.

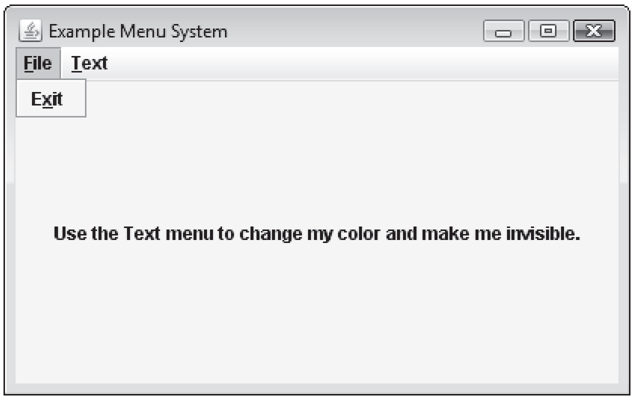
Line 136 creates a JMenu component for the Text menu, and line 137 assigns a mnemonic key to it. Lines 140 through 142 add the blackItem, redItem, and blueItem radio button menu items to the Text menu. In line 143, the addSeparator method is called to add a separator bar to the menu. Because the addSeparator method is called just after the blueItem component is added and just before the visibleItem component is added, it will appear between the Blue and Visible items on the menu. Line 144 adds the Visible item to the Text menu.

Back in the buildMenuBar method, in lines 74 and 75, the File menu and Text menu are added to the menu bar. In line 78, the setJMenuBar method is called, passing menuBar as an argument. The setJMenuBar method is a JFrame method that places a menu bar in a frame. You pass a JMenuBar component as the argument. When the JFrame is displayed, the menu bar will appear at its top.

Figure 24-26 shows how the class's window appears with the File menu and the Text menu opened. Selecting a color from the Text menu causes an instance of the ColorListener class to execute its actionPerformed method, which changes the color of the text. Selecting the Visible item causes an instance of the VisibleListener class to execute its actionPerformed method, which toggles the label's visibility.

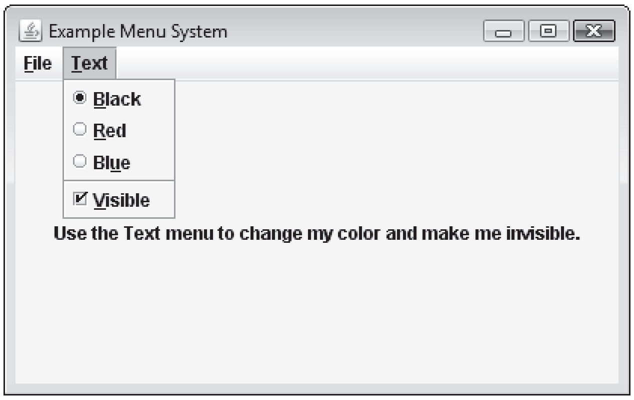
Figure 24-26 The window with the File menu and Text menu opened (Oracle Corporate Counsel)

The window with the File menu opened.



Use the Text menu to change my color and make me invisible.

The window with the Text menu opened.



Use the Text menu to change my color and make me invisible.

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.12 Briefly describe each of the following menu system items:
- a) Menu bar
- b) Menu item
- c) Check box menu item
- d) Radio button menu item
- e) Submenu
- f) Separator bar

24.13 What class do you use to create a regular menu item? What do you pass to the class constructor?

24.14 What class do you use to create a radio button menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

24.15 How do you create a relationship between radio button menu items so that only one may be selected at a time?

24.16 What class do you use to create a check box menu item? What do you pass to the class constructor? How do you cause it to be initially selected?

24.17 What class do you use to create a menu? What do you pass to the class constructor?

24.18 What class do you use to create a menu bar?

24.19 How do you place a menu bar in a JFrame?

24.20 What type of event do menu items generate when selected by the user?

24.21 How do you change the size of a component such as a JLabel after it has been created?

24.22 What arguments do you pass to the Dimension class constructor?

24.9 More about Text Components: Text Areas and Fonts

CONCEPT: A text area is a multi-line text field that can accept several lines of text input. Components that inherit from the JComponent class have a setFont method that allows you to change the font and style of the component's text.

Text Areas

In Chapter 23, you were introduced to the JTextField class, which is used to create text fields. A text field is a component that allows the user to enter a single line of text. A text area is like a text field that can accept multiple lines of input. You use the JTextArea class to create a text area. Here is the general format of two of the class's constructors:

```
JTextArea(int rows, int columns)
JTextArea(String text, int rows, int columns)
```

In both constructors, `rows` is the number of rows or lines of text that the text area is to display, and `columns` is the number of columns or characters that are to be displayed per line. In the second constructor, `text` is a string that the text area will initially display. For example, the following statement creates a text area with 20 rows and 40 columns:

```
JTextArea textInput = new JTextArea(20, 40);
```

The following statement creates a text area with 20 rows and 40 columns that will initially display the text stored in the `String` object `info`:

```
JTextArea textInput = new JTextArea(info, 20, 40);
```

As with the `JTextField` class, the `JTextArea` class provides the `getText` and `setText` methods for getting and setting the text contained in the component. For example, the following statement gets the text stored in the `textInput` text area and stores it in the `String` object `userText`:

```
String userText = textInput.getText();
```

The following statement stores the text that is in the `String` object `info` in the `textInput` text area:

```
textInput.setText(info);
```

`JTextArea` components do not automatically display scroll bars. To display scroll bars on a `JTextArea` component, you must add it to the scroll pane. As you already know, you create a scroll pane with the `JScrollPane` class. Here is an example of code that creates a text area and adds it to a scroll pane:

```
JTextArea textInput = new JTextArea(20, 40);
JScrollPane scrollPane = new JScrollPane(textInput);
```

The `JScrollPane` object displays both vertical and horizontal scroll bars on a text area. By default, the scroll bars are not displayed until they are needed; however, you can alter this behavior with two of the `JScrollPane` class's methods. The `setHorizontalScrollBarPolicy` method takes an `int` argument that specifies when a horizontal scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED.`** This is the default setting. A horizontal scroll bar is displayed only when there is not enough horizontal space to display the text contained in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_NEVER.`** This setting prevents a horizontal scroll bar from being displayed in the text area.
- **`JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS.`** With this setting, a horizontal scroll bar is always displayed, even when it is not needed.

The `setVerticalScrollBarPolicy` method also takes an `int` argument, which specifies when a vertical scroll bar should appear in the scroll pane. You can pass one of the following constants as an argument:

- **`JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED.`** This is the default setting. A vertical scroll bar is displayed only when there is not enough vertical space to display the text contained in the text area.

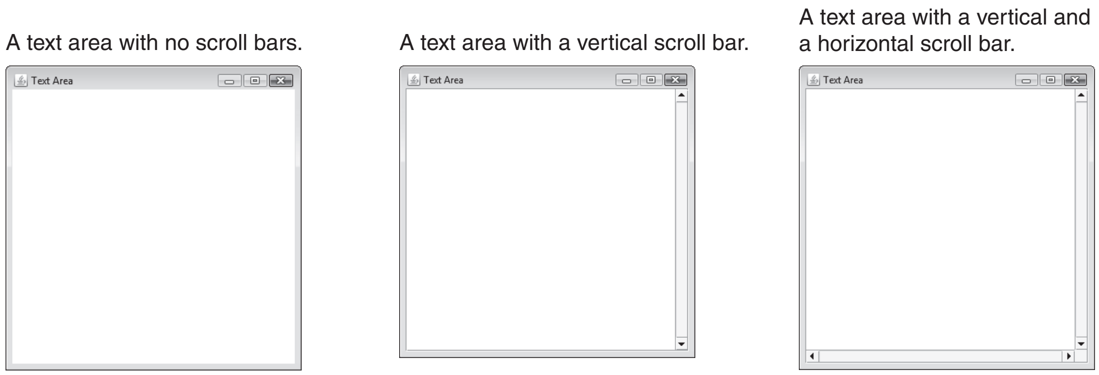
- **`JScrollPane.VERTICAL_SCROLLBAR_NEVER`.** This setting prevents a vertical scroll bar from being displayed in the text area.
- **`JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`.** With this setting, a vertical scroll bar is always displayed, even when it is not needed.

For example, the following code specifies that a vertical scroll bar should always appear on a scroll pane's component, but a horizontal scroll bar should not appear:

```
scrollPane.setHorizontalScrollBarPolicy(
               JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
scrollPane.setVerticalScrollBarPolicy(
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

Figure 24-27 shows a text area without scroll bars, a text area with a vertical scroll bar, and a text area with both a horizontal and a vertical scroll bar.

Figure 24-27 Text areas with and without scroll bars (Oracle Corporate Counsel)


A text area with no scroll bars.
A text area with a vertical scroll bar.
A text area with a vertical and a horizontal scroll bar.
Text Area
Text Area
Text Area

By default, `JTextArea` components do not perform line wrapping. This means that when text is entered into the component and the end of a line is reached, the text does not wrap around to the next line. If you want line wrapping, you use the `JTextArea` class's `setLineWrap` method to turn it on. The method accepts a `boolean` argument. If you pass `true`, line wrapping is turned on. If you pass `false`, line wrapping is turned off. Here is an example of a statement that turns a text area's line wrapping on:

```
textInput.setLineWrap(true);
```

There are two different styles of line wrapping: word wrapping and character wrapping. When word wrapping is performed, the line breaks always occur between words, never in the middle of a word. When character wrapping is performed, lines are broken between characters. This means that lines can be broken in the middle of a word. You specify the style of line wrapping that you prefer with the `JTextArea` class's `setWrapStyleWord` method. This method accepts a `boolean` argument. If you pass `true`, the text area will perform word wrapping. If you pass `false`, the text area will perform character wrapping. The default style is character wrapping.

Fonts

The appearance of a component's text is determined by the text's font, style, and size. The font is the name of the typeface—the style can be plain, bold, and/or italic—and the size is the size of the text in points. To change the appearance of a component's text you use the component's `setFont` method, which is inherited from the `JComponent` class. The general format of the method is as follows:

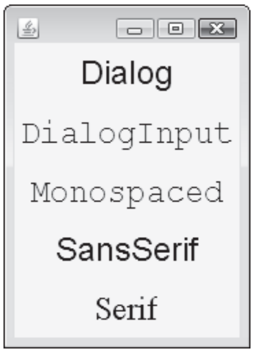
```
void setFont(Font appearance)
```

You pass a `Font` object as an argument to this method. The `Font` class constructor has the following general format:

```
Font(String fontName, int style, int size);
```

The first argument is the name of a font. Although the fonts that are available vary from system to system, Java guarantees that you will have Dialog, DialogInput, Monospaced, SansSerif, and Serif. Figure 24-28 shows an example of each of these.

Figure 24-28 Examples of fonts (Oracle Corporate Counsel)



The second argument to the `Font` constructor is an `int` that represents the style of the text. The `Font` class provides the following constants that you can use: `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The third argument is the size of the text in points. (There are 72 points per inch, so a 72-point font has a height of one inch. Ten- and twelve-point fonts are normally used for most applications.) Here is an example of a statement that changes the text of a label to a 24-point bold serif font:

```
label.setFont(new Font("Serif", Font.BOLD, 24));
```

You can combine styles by mathematically adding them. For example, the following statement changes a label's text to a 24-point bold and italic serif font:

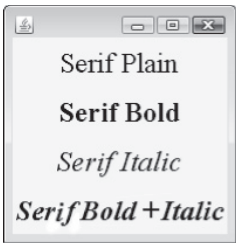
```
label.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

Figure 24-29 shows an example of the serif font in plain, bold, italic, and bold plus italic styles. The following code was used to create the labels:

```
JLabel label1 = new JLabel("Serif Plain", SwingConstants.CENTER);
label1.setFont(new Font("Serif", Font.PLAIN, 24));

JLabel label2 = new JLabel("Serif Bold", SwingConstants.CENTER);
label2.setFont(new Font("Serif", Font.BOLD, 24));
```

```
JLabel label3 = new JLabel("Serif Italic", SwingConstants.CENTER);
label3.setFont(new Font("Serif", Font.ITALIC, 24));

JLabel label4 = new JLabel("Serif Bold + Italic",
                           SwingConstants.CENTER);
label4.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 24));
```

Figure 24-29 Examples of serif plain, bold, italic, and bold plus italic (Oracle Corporate Counsel)



Serif Plain

Serif Bold

Serif Italic

Serif Bold +Italic

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.23 What arguments do you pass to the `JTextArea` constructor?

24.24 How do you retrieve the text that is stored in a `JTextArea` component?

24.25 Does the `JTextArea` component automatically display scroll bars? If not, how do you accomplish this?

24.26 What is line wrapping? What are the two styles of line wrapping? How do you turn a `JTextArea` component's line wrapping on? How do you select a line wrapping style?

24.27 What type of argument does a component's `setFont` method accept?

24.28 What are the arguments that you pass to the `Font` class constructor?

See the Simple Text Editor Application case study on this book's companion Web site (www.pearson.com/gaddis) for an in-depth example that uses menus and other topics from this chapter.

24.10 Sliders

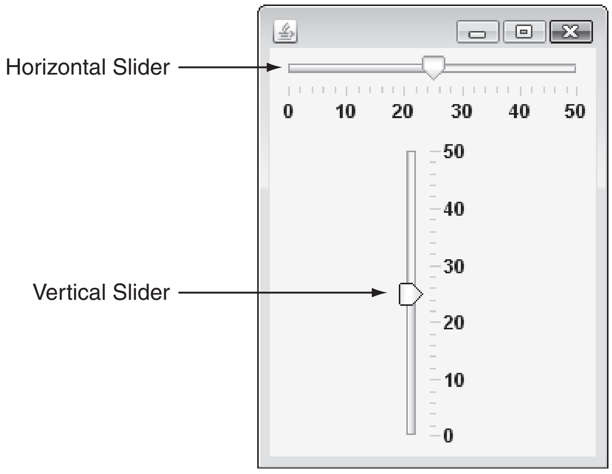
CONCEPT: A slider is a component that allows the user to adjust a number graphically within a range of values.

Sliders, which are created from the `JSlider` class, display an image of a "slider knob" that can be dragged along a track. Sliders can be horizontally or vertically oriented, as shown in Figure 24-30.

A slider is designed to represent a range of numeric values. At one end of the slider is the range's minimum value and at the other end is the range's maximum value. Both of the sliders shown in Figure 24-30 represent a range of 0 through 50. Sliders hold a numeric value in a field, and as the user moves the knob along the track, the numeric value is adjusted accordingly. Notice that the sliders in Figure 24-30 have accompanying tick marks. At every tenth value, a major tick mark is displayed along with a label indicating the value at that tick mark. Between the major tick marks are minor tick marks, which in this example are displayed at every second value. The appearance of tick marks, their spacing, and the appearance of labels can be controlled through methods in the `JSlider` class. The `JSlider` constructor has the following general format:

```
JSlider(int orientation, int minValue,
        int maxValue, int initialValue)
```

Figure 24-30 A horizontal and a vertical slider (Oracle Corporate Counsel)



The first argument is an `int` specifying the slider's orientation. You should use one of the constants `JSlider.HORIZONTAL` or `JSlider.VERTICAL`. The second argument is the minimum value of the slider's range and the third argument is the maximum value of the slider's range. The fourth argument is the initial value of the slider, which determines the initial position of the slider's knob. For example, the following code could be used to create the sliders shown in Figure 24-30:

```
JSlider slider1 = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
JSlider slider2 = new JSlider(JSlider.VERTICAL, 0, 50, 25);
```

You set the major and minor tick mark spacing with the methods `setMajorTickSpacing` and `setMinorTickSpacing`. Each of these methods accepts an `int` argument that specifies the intervals of the tick marks. For example, the following code sets the `slider1` object's major tick mark spacing at 10, and its minor tick mark spacing at 2:

```
slider1.setMajorTickSpacing(10);
slider1.setMinorTickSpacing(2);
```

If the `slider1` component's range is 0 through 50, then these statements would cause major tick marks to be displayed at values 0, 10, 20, 30, 40, and 50. Minor tick marks would be displayed at values 2, 4, 6, and 8, then at values 12, 14, 16, and 18, and so forth.

By default, tick marks are not displayed, and setting their spacing does not cause them to be displayed. You display tick marks by calling the `setPaintTicks` method, which accepts a `boolean` argument. If you pass `true`, then tick marks are displayed. If you pass `false`, they are not displayed. Here is an example:

```
slider1.setPaintTicks(true);
```

By default, labels are not displayed either. You display numeric labels on the slider component by calling the `setPaintLabels` method, which accepts a `boolean` argument. If you pass `true`, then numeric labels are displayed at the major tick marks. If you pass `false`, labels are not displayed. Here is an example:

```
slider1.setPaintLabels(true);
```

When the knob's position is moved, the slider component generates a change event. To handle the change event, you must write a change listener class. When you write a change listener class, it must meet the following requirements:

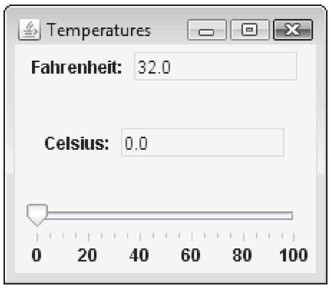
- It must implement the `ChangeListener` interface. This interface is in the `javax.swing.event` package.
- It must have a method named `stateChanged`. This method must take an argument of the `ChangeEvent` type.

To retrieve the current value stored in a `JSlider`, use the `getValue` method. This method returns the slider's value as an `int`. Here is an example:

```
currentValue = slider1.getValue();
```

The `TempConverter` class shown in Code Listing 24-7 demonstrates the `JSlider` component. This class displays the window shown in Figure 24-31. Two temperatures are initially shown: 32.0 degrees Fahrenheit and 0.0 degrees Celsius. A slider, which has the range of 0 through 100, allows you to adjust the Celsius temperature and immediately see the Fahrenheit conversion. The `main` method creates an instance of the class and displays the window.

Figure 24-31 Window displayed by the `TempConverterWindow` class (Oracle Corporate Counsel)


Temperatures

Fahrenheit: 32.0

Celsius: 0.0

Code Listing 24-7 (`TempConverter.java`)

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

/**
   This class displays a window with a slider component.
   The user can convert the Celsius temperatures from
   0 through 100 to Fahrenheit by moving the slider.
*/

public class TempConverter extends JFrame
{
   private JLabel label1, label2;     // Message labels
   private JTextField fahrenheitTemp; // Fahrenheit temp
   private JTextField celsiusTemp;    // Celsius temp
   private JPanel fpanel;             // Fahrenheit panel
   private JPanel cpanel;             // Celsius panel
   private JPanel sliderPanel;        // Slider panel
   private JSlider slider;            // Temperature adjuster

   /**
      Constructor
   */

   public TempConverter()
   {
      // Set the title.
      setTitle("Temperatures");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create the message labels.
      label1 = new JLabel("Fahrenheit: ");
      label2 = new JLabel("Celsius: ");

      // Create the read-only text fields.
      fahrenheitTemp = new JTextField("32.0", 10);
      fahrenheitTemp.setEditable(false);
      celsiusTemp = new JTextField("0.0", 10);
      celsiusTemp.setEditable(false);

      // Create the slider.
      slider = new JSlider(JSlider.HORIZONTAL, 0, 100, 0);
      slider.setMajorTickSpacing(20); // Major tick every 20
      slider.setMinorTickSpacing(5);  // Minor tick every 5
      slider.setPaintTicks(true);     // Display tick marks
      slider.setPaintLabels(true);    // Display numbers
      slider.addChangeListener(new SliderListener());

      // Create panels and place the components in them.
      fpanel = new JPanel();
      fpanel.add(label1);
      fpanel.add(fahrenheitTemp);
      cpanel = new JPanel();
      cpanel.add(label2);
      cpanel.add(celsiusTemp);
      sliderPanel = new JPanel();
      sliderPanel.add(slider);

      // Create a GridLayout manager.
      setLayout(new GridLayout(3, 1));

      // Add the panels to the content pane.
      add(fpanel);
      add(cpanel);
      add(sliderPanel);

      // Pack and display the frame.
      pack();
      setVisible(true);
   }

   /**
      Private inner class to handle the change events
      that are generated when the slider is moved.
   */

   private class SliderListener implements ChangeListener
   {
      public void stateChanged(ChangeEvent e)
      {
         double fahrenheit, celsius;

         // Get the slider value.
         celsius = slider.getValue();

         // Convert the value to Fahrenheit.
         fahrenheit = (9.0 / 5.0) * celsius + 32.0;

         // Store the celsius temp in its display field.
         celsiusTemp.setText(Double.toString(celsius));

         // Store the Fahrenheit temp in its display field.
         fahrenheitTemp.setText(String.format("%.1f", fahrenheit));
      }
   }

   /*
      The main method creates an instance of the
      class, which displays a window with a slider.
   */

   public static void main(String[] args)
   {
      new TempConverter();
   }
}
```

Checkpoint

MyProgrammingLab™ *www.myprogramminglab.com*

24.29 What type of event does a `JSlider` generate when its slider knob is moved?

24.30 What `JSlider` methods do you use to perform each of these operations?

a) Establish the spacing of major tick marks.
b) Establish the spacing of minor tick marks.
c) Cause tick marks to be displayed.
d) Cause labels to be displayed.

24.11 Look and Feel

CONCEPT: A GUI application's appearance is determined by its look and feel. Java allows you to select an application's look and feel.

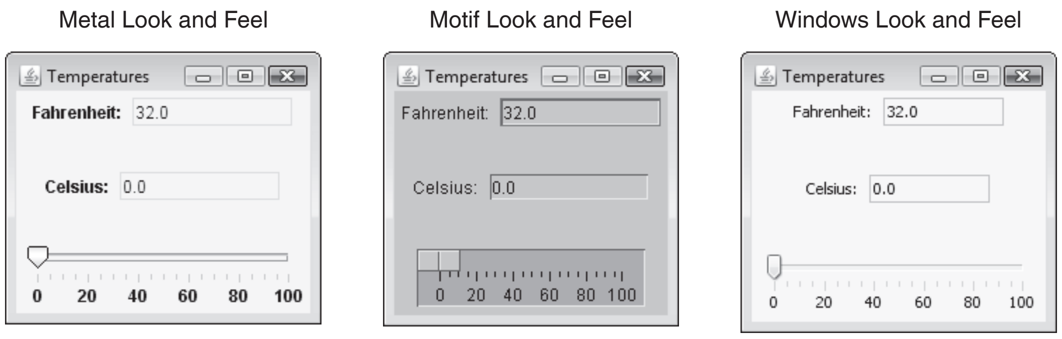
Most operating systems' GUIs have their own unique appearance and style conventions. For example, if a Windows user switches to a Macintosh, UNIX, or Linux system, the first thing he or she is likely to notice is the difference in the way the GUIs on each system appear. The appearance of a particular system's GUI is known as its look and feel.

Java allows you to select the look and feel of a GUI application. The default look and feel for Java is called *Ocean*. This is the look and feel that you have seen in all of the GUI applications that we have written in this text. Some of the other look and feel choices are Metal, Motif, and Windows. Metal was the default look and feel for previous versions of Java. Motif is similar to a UNIX look and feel. Windows is the look and feel of the Windows operating system. Figure 24-32 shows how the `TempConverterWindow` class window, presented earlier in this chapter, appears in each of these looks and feels.

NOTE: Ocean is actually a special theme of the Metal look and feel.

NOTE: Currently the Windows look and feel is available only on computers running the Microsoft Windows operating system.

Figure 24-32 Metal, Motif, and Windows looks and feels (Oracle Corporate Counsel)



To change an application's look and feel, you call the `UIManager` class's static `setLookAndFeel` method. Java has a class for each look and feel, and this method takes the fully qualified class name for the desired look and feel as its argument. The class name must be passed as a string. Table 24-1 lists the fully qualified class names for the Metal, Motif, and Windows looks and feels.

Table 24-1 Look and feel class names

Class Name	Look and Feel
`"javax.swing.plaf.metal.MetalLookAndFeel"`	Metal
`"com.sun.java.swing.plaf.motif.MotifLookAndFeel"`	Motif
`"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"`	Windows

When you call the `UIManager.setLookAndFeel` method, any components that have already been created need to be updated. You do this by calling the `SwingUtilities.updateComponentTreeUI` method, passing a reference to the component that you want to update as an argument.

The `UIManager.setLookAndFeel` method throws a number of exceptions. Specifically, it throws `ClassNotFoundException`, `InstantiationException`, `IllegalAccessException`, and `UnsupportedLookAndFeelException`. Unless you want to trap each of these types of exceptions, you can simply trap exceptions of type `Exception`. Here is an example of code that can be run from a `JFrame` object that changes its look and feel to Motif:

```
try
{
   UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
   SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
   JOptionPane.showMessageDialog(null, "Error setting " +
                                  "the look and feel.");
   System.exit(0);
}
```

And here is an example of code that can be run from a `JFrame` object that changes its look and feel to Windows:

```
try
{
   UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
   SwingUtilities.updateComponentTreeUI(this);
}
catch (Exception e)
{
   JOptionPane.showMessageDialog(null, "Error setting " +
                                  "the look and feel.");
   System.exit(0);
}
```

24.12 Common Errors to Avoid

- **Only retrieving the first selected item from a list component in which multiple items have been selected.** If multiple items have been selected in a list component, the `getSelectedValue` method returns only the first selected item. Likewise, the `getSelectedIndex` method returns only the index of the first selected item. You should use the `getSelectedValues` or `getSelectedIndices` methods instead.
- **Using 1 as the beginning index for a list or combo box.** The indices for a list or combo box start at 0, not 1.
- **Forgetting to add a list or text area to a scroll pane.** The `JList` and `JTextArea` components do not automatically display scroll bars. You must add these components to a scroll pane object in order for them to display scroll bars.
- **Using the add method instead of the constructor to add a component to a scroll pane.** To add a component to a scroll pane, you must pass a reference to the component as an argument to the `JScrollPane` constructor.
- **Adding a component to a scroll pane and then adding the component (not the scroll pane) to another container, such as a panel.** If you add a component to a scroll pane and then intend to add that same component to a panel or other container, you must add the scroll pane instead of the component. Otherwise, the scroll bars will not appear on the component.

- **Forgetting to call the `setEditable` method to give a combo box a text field.** By default, a combo box is the combination of a button and a list. To make it a combination of a text field and a list, you must call the `setEditable` method and pass `true` as an argument.
- **Trying to open an image file of an unsupported type.** Currently, an `ImageIcon` object can open image files that are stored in JPEG, GIF, or PNG formats.
- **Loading an image into an existing `JLabel` component and clipping part of the image.** If you have not explicitly set the preferred size of a `JLabel` component, it resizes itself automatically when you load an image into it. The `JFrame` that encloses the `JLabel` does not automatically resize, however. You must call the `JFrame` object's `pack` method or `setPreferredSize` method to resize it.
- **Assigning the same mnemonic to more than one component.** If you assign the same mnemonic to more than one component in a window, it works only for the first component that you assigned it to.
- **Forgetting to add menu items to a `JMenu` component, and `JMenu` components to a `JMenuBar` component.** After you create a menu item, you must add it to a `JMenu` component in order for it to be displayed on the menu. Likewise, `JMenu` components must be added to a `JMenuBar` component in order to be displayed on the menu bar.
- **Not calling the `JFrame` object's `setJMenuBar` method to place the menu bar.** To display a menu bar, you must call the `setJMenuBar` method and pass it as an argument.
- **Not grouping `JRadioButtonMenuItems` in a `ButtonGroup` object.** Just like regular radio button components, you must group radio button menu items in a button group in order to create a mutually exclusive relationship among them.

Review Questions and Exercises

Multiple Choice and True/False

1. You can use this method to make a text field read-only.
   a. `setReadOnly`
   b. `setChangeable`
   c. `setUneditable`
   d. `setEditable`
2. A `JList` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
3. To display a scroll bar with a `JList` component, you must ________.
   a. do nothing; the `JList` automatically appears with scroll bars if necessary
   b. add the `JList` component to a `JScrollPane` component
   c. call the `setScrollBar` method
   d. none of the above; you cannot display a scroll bar with a `JList` component

4. This is the `JList` component's default selection mode.
   a. single selection
   b. single interval selection
   c. multiple selection
   d. multiple interval selection
5. A list selection listener must have this method.
   a. `valueChanged`
   b. `selectionChanged`
   c. `actionPerformed`
   d. `itemSelected`
6. The `ListSelectionListener` interface is in this package.
   a. `java.awt`
   b. `java.awt.event`
   c. `javax.swing.event`
   d. `javax.event`
7. This `JList` method returns −1 if no item in the list is selected.
   a. `getSelectedValue`
   b. `getSelectedItem`
   c. `getSelectedIndex`
   d. `getSelection`
8. A `JComboBox` component generates this type of event when the user selects an item.
   a. action event
   b. item event
   c. list selection event
   d. list change event
9. You can pass an instance of this class to the `JLabel` constructor if you want to display an image in the label.
   a. `ImageFile`
   b. `ImageIcon`
   c. `JLabelImage`
   d. `JImageFile`
10. This method can be used to store an image in a `JLabel` or a `JButton` component.
   a. `setImage`
   b. `storeImage`
   c. `getIcon`
   d. `setIcon`
11. This is text that appears in a small box when the user holds the mouse cursor over a component.
   a. mnemonic
   b. instant message
   c. tool tip
   d. pop-up mnemonic

12. This is a key that activates a component just as if the user clicked it with the mouse.
    a. mnemonic
    b. key activator
    c. tool tip
    d. click simulator
13. To display an open file or save file dialog box, you use this class.
    a. `JFileChooser`
    b. `JOpenSaveDialog`
    c. `JFileDialog`
    d. `JFileOptionPane`
14. To display a dialog box that allows the user to select a color, you use this class.
    a. `JColor`
    b. `JColorDialog`
    c. `JColorChooser`
    d. `JColorOptionPane`
15. You use this class to create a menu bar.
    a. `MenuBar`
    b. `JMenuBar`
    c. `JMenu`
    d. `JBar`
16. You use this class to create a radio button menu item.
    a. `JMenuItem`
    b. `JRadioButton`
    c. `JRadioButtonItem`
    d. `JRadioButtonMenuItem`
17. You use this method to place a menu bar on a `JFrame`.
    a. `setJMenuBar`
    b. `setMenuBar`
    c. `placeMenuBar`
    d. `setJMenu`
18. The `setPreferredSize` method accepts this as its argument(s).
    a. a `Size` object
    b. two `int` values
    c. a `Dimension` object
    d. one `int` value
19. Components of this class are multi-line text fields.
    a. `JMultiLineTextField`
    b. `JTextArea`
    c. `JTextField`
    d. `JEditField`

20. This method is inherited from `JComponent` and changes the appearance of a component's text.
    a. `setAppearance`
    b. `setTextAppearance`
    c. `setFont`
    d. `setText`
21. This method sets the intervals at which major tick marks are displayed on a `JSlider` component.
    a. `setMajorTickSpacing`
    b. `setMajorTickIntervals`
    c. `setTickSpacing`
    d. `setIntervals`
22. **True or False:** You can use code to change the contents of a read-only text field.
23. **True or False:** A `JList` component automatically appears with a line border drawn around it.
24. **True or False:** In single interval selection mode, the user may select multiple items from a `JList` component.
25. **True or False:** With an editable combo box the user may only enter a value that appears in the component's list.
26. **True or False:** You can store either text or an image in a `JLabel` object, but not both.
27. **True or False:** You can store large images as well as small ones in a `JLabel` component.
28. **True or False:** Mnemonics are useful for users who are good with the keyboard.
29. **True or False:** A `JMenuBar` object acts as a container for `JMenu` components.
30. **True or False:** A `JMenu` object cannot contain other `JMenu` objects.
31. **True or False:** A `JTextArea` component does not automatically display scroll bars.
32. **True or False:** By default, a `JTextArea` component does not perform line wrapping.
33. **True or False:** A `JSlider` component generates an action event when the slider knob is moved.
34. **True or False:** By default, a `JSlider` component displays labels and tick marks.
35. **True or False:** When labels are displayed on a `JSlider` component, they are displayed on the major tick marks.

Find the Error

1.
```
// Create a read-only text field.
JTextField textField = new JTextField(10);
textField.setEditable(true);
```
2.
```
// Create a black 1-pixel border around list, a JList component.
list.setBorder(Color.BLACK, 1);
```
3.
```
// Create a JList and add it to a scroll pane.
// Assume that array already exists.
JList list = new JList(array);
JScrollPane scrollPane = new JScrollPane();
scrollPane.add(list);
```

4.
```
// Assume that nameBox is a combo box and is properly set up
// with a list of names to choose from.
// Get value of the selected item.
String selectedName = nameBox.getSelectedIndex();
```
5.
```
JLabel label = new JLabel("Have a nice day!");
label.setImage(image);
```
6.
```
// Add a menu to the menu bar.
JMenuBar menuBar = new JMenuBar(menuItem);
```
7.
```
// Create a text area with 20 columns and 5 rows.
JTextArea textArea = new JTextArea (20, 5);
```

Algorithm Workbench

1. Give an example of code that creates a read-only text field.
2. Write code that creates a list with the following items: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.
3. Write code that adds a scroll bar to the list you created in your answer to Algorithm Workbench 2.
4. Assume that the variable `myList` references a `JList` component, and `selection` is a `String variable`. Write code that assigns the selected item in the `myList` component to the `selection` variable.
5. Assume that the variable `myComboBox` references an uneditable combo box, and `selectionIndex` is an `int` variable. Write code that assigns the index of the selected item in the `myComboBox` component to the `selectionIndex` variable.
6. Write code that stores the image in the file *dog.jpg* in a label.
7. Assume that `label` references an existing `JLabel` object. Write code that stores the image in the file *picture.gif* in the label.
8. Write code that creates a button with the text "Open File." Assign the O key as a mnemonic and assign "This button opens a file" as the component's tool tip.
9. Write code that displays a file open dialog box. If the user selects a file, the code should store the file's path and name in a `String` variable.
10. Write code that creates a text area displaying 10 rows and 15 columns. The text area should be capable of displaying scroll bars, when necessary. It should also perform word style line wrapping.
11. Write the code that creates a menu bar with one menu named File. The File menu should have the F key assigned as a mnemonic. The File menu should have three menu items: Open, Print, and Exit. Assign mnemonic keys of your choice to each of these items. Register an instance of the `OpenListener` class as an action listener for the Open menu item, an instance of the `PrintListener` class as an action listener for the Print menu item, and an instance of the `ExitListener` class as an action listener for the Exit menu item. Assume these classes have already been created.
12. Write code that creates a `JSlider` component. The component should be horizontally oriented and its range should be 0 through 1000. Labels and tick marks should be displayed. Major tick marks should appear at every 100th number, and minor tick marks should appear at every 25th number. The initial value of the slider should be set at 500.

Short Answer

1. What selection mode should you select if you want the user to select a single item only in a list?
2. You want to provide 20 items in a list for the user to select from. Which component would take up less space, a `JList` or a `JComboBox`?
3. What is the difference between an uneditable combo box and an editable combo box? Which one is a combo box by default?
4. Describe how you can store both an image and text in a `JLabel` component.
5. What is a mnemonic? How does the user use it?
6. What happens when the mnemonic that you assign to a component is a letter that appears in the component's text?
7. What is a tool tip? What is its purpose?
8. What do you do to a group of radio button menu items so that only one of them can be selected at a time?
9. When a checked menu item shows a check mark next to it, what happens when the user clicks on it?
10. What fonts does Java guarantee you have?
11. Why would a `JSlider` component be ideal when you want the user to enter a number, but you want to make sure that the number is within a range?
12. What are the standard GUI looks and feels that are available in Java?

Programming Challenges

MyProgrammingLab™

Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.

1. Scrollable Tax Calculator

Create an application that allows you to enter the amount of a purchase and then displays the amount of sales tax on that purchase. Use a slider to adjust the tax rate between 0 percent and 10 percent.

2. Image Viewer

VideoNote
The Image Viewer Problem

Write an application that allows the user to view image files. The application should use either a button or a menu item that displays a file chooser. When the user selects an image file, it should be loaded and displayed.

3. Dorm and Meal Plan Calculator

A university has the following dormitories:

Allen Hall: $1,500 per semester
Pike Hall: $1,600 per semester
Farthing Hall: $1,200 per semester
University Suites: $1,800 per semester

The university also offers the following meal plans:

7 meals per week: $560 per semester
14 meals per week: $1,095 per semester
Unlimited meals: $1,500 per semester

Create an application with two combo boxes. One should hold the names of the dormitories, and the other should hold the meal plans. The user should select a dormitory and a meal plan, and the application should show the total charges for the semester.

4. Skateboard Designer

The Skate Shop sells the skateboard products listed in Table 24-2.

Table 24-2 Skateboard products

Decks	Truck Assemblies	Wheels
The Master Thrasher $60	7.75 inch axle $35	51 mm $20
The Dictator $45	8 inch axle $40	55 mm $22
The Street King $50	8.5 inch axle $45	58 mm $24
		61 mm $28

In addition, the Skate Shop sells the following miscellaneous products and services:

Grip tape: $10
Bearings: $30
Riser pads: $2
Nuts & bolts kit: $3

Create an application that allows the user to select one deck, one truck assembly, and one wheel set from either list components or combo boxes. The application should also have a list component that allows the user to select multiple miscellaneous products. The application should display the subtotal, the amount of sales tax (at 6 percent), and the total of the order.

5. Shopping Cart System

Create an application that works like a shopping cart system for a bookstore. In this chapter's source code folder (available on the book's companion Web site at www.pearson .com/gaddis), you will find a file named *BookPrices.txt*. This file contains the names and prices of various books, formatted in the following fashion:

I Did It Your Way, $11.95
The History of Scotland, $14.50
Learn Calculus in One Day, $29.95
Feel the Stress, $18.50

Each line in the file contains the name of a book, followed by a comma, followed by the book's retail price. When your application begins execution, it should read the contents of the file and store the book titles in a list component. The user should be able to select a title from the list and add it to a shopping cart, which is simply another list component. The application should have buttons or menu items that allow the user to remove items from the shopping cart, clear the shopping cart of all selections, and check out. When the user checks out, the application should calculate and display the subtotal of all the books in the shopping cart, the sales tax (which is 6 percent of the subtotal), and the total.

6. Cell Phone Packages

Cell Solutions, a cell phone provider, sells the following packages:

300 minutes per month: $45.00 per month
800 minutes per month: $65.00 per month
1500 minutes per month: $99.00 per month

The provider sells the following phones (a 6 percent sales tax applies to the sale of a phone):

Model 100: $29.95
Model 110: $49.95
Model 200: $99.95

Customers may also select the following options:

Voice mail: $5.00 per month
Text messaging: $10.00 per month

Write an application that displays a menu system. The menu system should allow the user to select one package, one phone, and any of the options desired. As the user selects items from the menu, the application should show the prices of the items selected.

7. Shade Designer

A custom window shade designer charges a base fee of $50 per shade. In addition, charges are added for certain styles, sizes, and colors as follows:

Styles:

Regular shades: Add $0
Folding shades: Add $10
Roman shades: Add $15

Sizes:

25 inches wide: Add $0
27 inches wide: Add $2
32 inches wide: Add $4
40 inches wide: Add $6

Colors:

Natural: Add $5
Blue: Add $0
Teal: Add $0
Red: Add $0
Green: Add $0

Create an application that allows the user to select the style, size, color, and number of shades from lists or combo boxes. The total charges should be displayed.

8. Conference Registration System

Create an application that calculates the registration fees for a conference. The general conference registration fee is $895 per person, and student registration is $495 per person. There is also an optional opening night dinner with a keynote speech for $30 per person. In addition, the optional preconference workshops listed in Table 24-3 are available.

Table 24-3 Optional preconference workshops

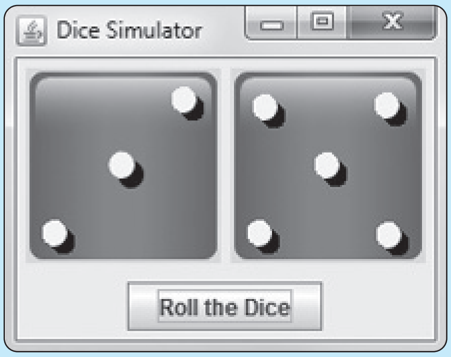
Workshop	Fee
Introduction to E-commerce	$295
The Future of the Web	$295
Advanced Java Programming	$395
Network Security	$395

The application should allow the user to select the registration type, the optional opening night dinner and keynote speech, and as many preconference workshops as desired. The total cost should be displayed.

9. Dice Simulator

Write a GUI application that simulates a pair of dice, similar to that shown in Figure 24-33. Each time the button is clicked, the application should roll the dice, using random numbers to determine the value of each die. (This chapter's source code folder contains images that you can use to display the dice.)

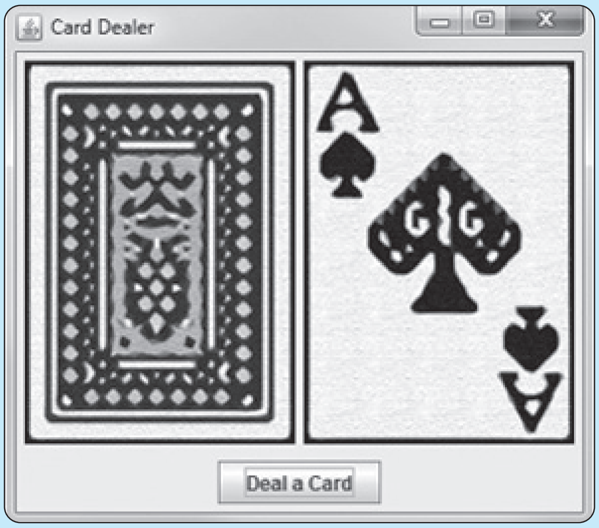
Figure 24-33 Dice simulator (Oracle Corporate Counsel)



10. Card Dealer

This chapter's source code folder contains images for a complete deck of poker cards. Write a GUI application, similar to the one shown in Figure 24-34, that randomly selects a card from the deck and displays it each time the user clicks the button. When a card has been selected, it is removed from the deck and cannot be selected again. Display a message when no more cards are left in the deck.

Figure 24-34 Card dealer (Oracle Corporate Counsel)

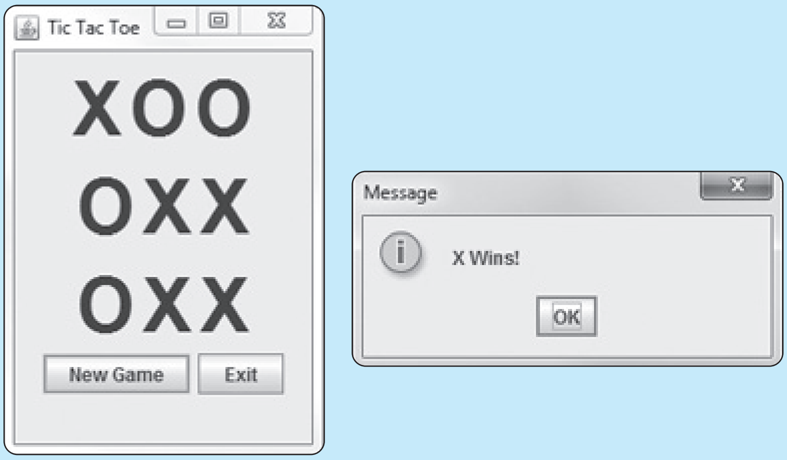


11. Tic Tac Toe Simulator

Create a GUI application that simulates a game of tic tac toe. Figure 24-35 shows an example of the application's window. The window shown in the figure uses nine large `JLabel` components to display the Xs and Os.

One approach in designing this application is to use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the *New Game* button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The `JLabel` components should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

Figure 24-35 The Tic Tac Toe application (Oracle Corporate Counsel)


Tic Tac Toe
XOO
OXX
OXX
New Game
Exit
Message
X Wins!
OK