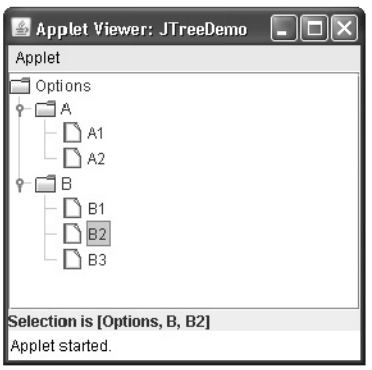
```
    // Handle tree selection events.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
      public void valueChanged(TreeSelectionEvent tse) {
        jlab.setText("Selection is " + tse.getPath());
      }
    });
  }
}
```

Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. **JTable** is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, **JTable** still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like **JTree**, **JTable** has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors. The one used here is

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A **JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple **JTable** that can be used to display data:

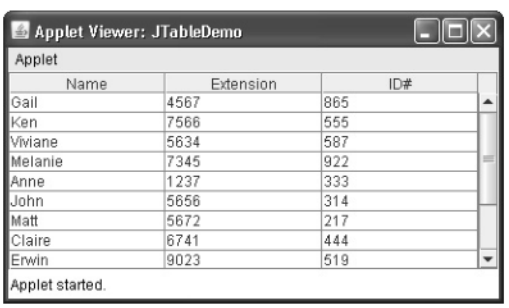
1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTableDemo" width=400 height=200>
  </applet>
*/

public class JTableDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
```

```
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Initialize column headings.
    String[] colHeads = { "Name", "Extension", "ID#" };

    // Initialize data.
    Object[][] data = {
      { "Gail", "4567", "865" },
      { "Ken", "7566", "555" },
      { "Viviane", "5634", "587" },
      { "Melanie", "7345", "922" },
      { "Anne", "1237", "333" },
      { "John", "5656", "314" },
      { "Matt", "5672", "217" },
      { "Claire", "6741", "444" },
      { "Erwin", "9023", "519" },
      { "Ellen", "1134", "532" },
      { "Jennifer", "5689", "112" },
      { "Ed", "9030", "133" },
      { "Helen", "6751", "145" }
    };

    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);

    // Add the scroll pane to the content pane.
    add(jsp);
  }
}
```

Output from this example is shown here:



CHAPTER

33 Introducing Swing Menus

This chapter introduces another fundamental aspect of the Swing GUI environment: the menu. Menus form an integral part of many applications because they present the program's functionality to the user. Because of their importance, Swing provides extensive support for menus. They are an area in which Swing's power is readily apparent.

The Swing menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The popup menu, which is usually activated by right-clicking the mouse.
- The toolbar, which provides rapid access to program functionality, often paralleling menu items.
- The action, which enables two or more different components to be managed by a single object. Actions are commonly used with menus and toolbars.

Swing menus also support accelerator keys, which enable menu items to be selected without having to activate the menu, and mnemonics, which allow a menu item to be selected by the keyboard once the menu options are displayed.

Menu Basics

The Swing menu system is supported by a group of related classes. The ones used in this chapter are shown in Table 33-1, and they represent the core of the menu system. Although they may seem a bit confusing at first, Swing menus are quite easy to use. Swing allows a high degree of customization, if desired; however, you will normally use the menu classes as-is because they support all of the most needed options. For example, you can easily add images and keyboard shortcuts to a menu.

Class	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more **JMenuItem**s.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item
JSeparator	The visual separator between menu items.
JPopupMenu	A menu that is typically activated by right-clicking the mouse.

Table 33-1 The Core Swing Menu Classes

Here is a brief overview of how the classes fit together. To create the top-level menu for an application, you first create a **JMenuBar** object. This class is, loosely speaking, a container for menus. To the **JMenuBar** instance, you will add instances of **JMenu**. Each **JMenu** object defines a menu. That is, each **JMenu** object contains one or more selectable items. The items displayed by a **JMenu** are objects of **JMenuItem**. Thus, a **JMenuItem** defines a selection that can be chosen by the user.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, popup menus. To create a popup menu, first create an object of type **JPopupMenu**. Then, add **JMenuItem**s to it. A popup menu is normally activated by clicking the right mouse button when the mouse is over a component for which a popup menu has been defined.

In addition to "standard" menu items, you can also include check boxes and radio buttons in a menu. A check box menu item is created by **JCheckBoxMenuItem**. A radio button menu item is created by **JRadioButtonMenuItem**. Both of these classes extend **JMenuItem**. They can be used in standard menus and popup menus.

JToolBar creates a stand-alone component that is related to the menu. It is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

JSeparator is a convenience class that creates a separator line in a menu.

One key point to understand about Swing menus is that each menu item extends **AbstractButton**. Recall that **AbstractButton** is also the superclass of all of Swing's button components, such as **JButton**. Thus, all menu items are, essentially, buttons. Obviously, they won't actually look like buttons when used in a menu, but they will, in many ways, act like buttons. For example, selecting a menu item generates an action event in the same way that pressing a button does.

Another key point is that **JMenuItem** is a superclass of **JMenu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **JMenu** object and then add it to another **JMenu** object. You will see this process in action in the following section.

As mentioned in passing previously, when a menu item is selected, an action event is generated. The action command string associated with that action event will, by default, be the name of the selection. Thus, you can determine which item was selected by examining

the action command. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item's action events. In this case, the menu selection is already known, and there is no need to examine the action command string to determine which item was selected.

Menus can also generate other types of events. For example, each time that a menu is activated, selected, or canceled, a **MenuEvent** is generated that can be listened for via a **MenuListener**. Other menu-related events include **MenuKeyEvent**, **MenuDragMouseEvent**, and **PopupMenuEvent**. In many cases, however, you need only watch for action events, and in this chapter, we will use only action events.

An Overview of JMenuBar, JMenu, and JMenuItem

Before you can create a menu, you need to know something about the three core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **JMenu** and **JMenuItem** are also used by popup menus. Thus, these classes form the foundation of the menu system.

JMenuBar

As mentioned, **JMenuBar** is essentially a container for menus. Like all components, it inherits **JComponent** (which inherits **Container** and **Component**). It has only one constructor, which is the default constructor. Therefore, initially the menu bar will be empty, and you will need to populate it with menus prior to use. Each application has one and only one menu bar.

JMenuBar defines several methods, but often you will only need to use one: **add( )**. The **add( )** method adds a **JMenu** to the menu bar. It is shown here:

JMenu add(JMenu *menu*)

Here, *menu* is a **JMenu** instance that is added to the menu bar. A reference to the menu is returned. Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add( )**, which is inherited from **Container**:

Component add(Component *menu*, int *idx*)

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove( )**, which is inherited from **Container**. It has these two forms:

void remove(Component *menu*)

void remove(int *idx*)

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

Another method that is sometimes useful is **getMenuCount( )**, shown here:

int getMenuCount( )

It returns the number of elements contained within the menu bar.

JMenuBar defines some other methods that you might find helpful in specialized applications. For example, you can obtain an array of references to the menus in the bar by calling **getSubElements( )**. You can determine if a menu is selected by calling **isSelected( )**.

Once a menu bar has been created and populated, it is added to a **JFrame** by calling **setJMenuBar( )** on the **JFrame** instance. (Menu bars *are not* added to the content pane.) The **setJMenuBar( )** method is shown here:

```
void setJMenuBar(JMenuBar mb)
```

Here, *mb* is a reference to the menu bar. The menu bar will be displayed in a position determined by the look and feel. Usually, this is at the top of the window.

JMenu

JMenu encapsulates a menu, which is populated with **JMenuItem**s. As mentioned, it is derived from **JMenuItem**. This means that one **JMenu** can be a selection in another **JMenu**. This enables one menu to be a submenu of another. **JMenu** defines a number of constructors. For example, here is the one used in the examples in this chapter:

```
JMenu(String name)
```

This constructor creates a menu that has the title specified by *name*. Of course, you don't have to give a menu a name. To create an unnamed menu, you can use the default constructor:

```
JMenu( )
```

Other constructors are also supported. In each case, the menu is empty until menu items are added to it.

JMenu defines many methods. Here is a brief description of some commonly used ones. To add an item to the menu, use the **add( )** method, which has a number of forms, including the two shown here:

```
JMenuItem add(JMenuItem item)
JMenuItem add(Component item, int idx)
```

Here, *item* is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by *idx*. As expected, indexing starts at zero. Both forms return a reference to the item added. As a point of interest, you can also use **insert( )** to add menu items to a menu.

You can add a separator (an object of type **JSeparator**) to a menu by calling **addSeparator( )**, shown here:

```
void addSeparator( )
```

The separator is added onto the end of the menu. You can insert a separator into a menu by calling **insertSeparator( )**, shown next:

```
void insertSeparator(int idx)
```

Here, *idx* specifies the zero-based index at which the separator will be added.

You can remove an item from a menu by calling **remove( )**. Two of its forms are shown here:

void remove(JMenuItem *menu*)

void remove(int *idx*)

In this case, *menu* is a reference to the item to remove and *idx* is the index of the item to remove.

You can obtain the number of items in the menu by calling **getMenuComponentCount( )**, shown here:

int getMenuComponentCount( )

You can get an array of the items in the menu by calling **getMenuComponents( )**, shown next:

Component[ ] getMenuComponents( )

An array containing the components is returned.

JMenuItem

JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. As mentioned, **JMenuItem** is derived from **AbstractButton**, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated. (This is similar to the way a **JButton** fires an action event when it is pressed.) **JMenuItem** defines many constructors. The ones used in this chapter are shown here:

JMenuItem(String *name*)

JMenuItem(Icon *image*)

JMenuItem(String *name*, Icon *image*)

JMenuItem(String *name*, int *mnem*)

JMenuItem(Action *action*)

The first constructor creates a menu item with the name specified by *name*. The second creates a menu item that displays the image specified by *image*. The third creates a menu item with the name specified by *name* and the image specified by *image*. The fourth creates a menu item with the name specified by *name* and uses the keyboard mnemonic specified by *mnem*. This mnemonic enables you to select an item from the menu by pressing the specified key. The last constructor creates a menu item using the information specified in *action*. A default constructor is also supported.

Because menu items inherit **AbstractButton**, you have access to the functionality provided by **AbstractButton**. One such method that is often useful with menus is **setEnabled( )**, which you can use to enable or disable a menu item. It is shown here:

void setEnabled(boolean *enable*)

If *enable* is **true**, the menu item is enabled. If *enable* is **false**, the item is disabled and cannot be selected.

Create a Main Menu

Traditionally, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. Fortunately, Swing makes creating and managing the main menu easy. This section shows you how to construct a basic main menu. Subsequent sections will show you how to add options to it.

Constructing the main menu requires several steps. First, create the **JMenuBar** object that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **JMenu** object and then adding **JMenuItem**s to it. After the menus have been created, add them to the menu bar. The menu bar, itself, must then be added to the frame by calling **setJMenuBar( )**. Finally, for each menu item, you must add an action listener that handles the action event fired when the menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label in the content pane. Sample output is shown in Figure 33-1.

```
// Demonstrate a simple main menu.

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

class MenuDemo implements ActionListener {

  JLabel jlab;

  MenuDemo() {
    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Menu Demo");
```

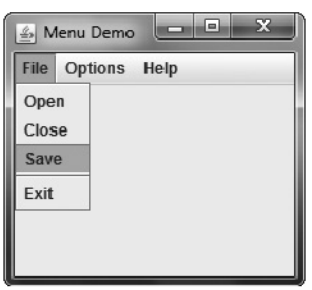


Figure 33-1 Sample output from the **MenuDemo** program

```
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());

// Give the frame an initial size.
jfrm.setSize(220, 200);

// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Create a label that will display the menu selection.
jlab = new JLabel();

// Create the menu bar.
JMenuBar jmb = new JMenuBar();

// Create the File menu.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// Create the Options menu.
JMenu jmOptions = new JMenu("Options");

// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
JMenuItem jmiRed = new JMenuItem("Red");
JMenuItem jmiGreen = new JMenuItem("Green");
JMenuItem jmiBlue = new JMenuItem("Blue");
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");
JMenuItem jmiHigh = new JMenuItem("High");
JMenuItem jmiLow = new JMenuItem("Low");
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);
```

```
    // Finally, add the entire options menu to
    // the menu bar
    jmb.add(jmOptions);

    // Create the Help menu.
    JMenu jmHelp = new JMenu("Help");
    JMenuItem jmiAbout = new JMenuItem("About");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);

    // Add action listeners for the menu items.
    jmiOpen.addActionListener(this);
    jmiClose.addActionListener(this);
    jmiSave.addActionListener(this);
    jmiExit.addActionListener(this);
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
    jmiAbout.addActionListener(this);

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

    // Add the menu bar to the frame.
    jfrm.setJMenuBar(jmb);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle menu item action events.
  public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```

Let's examine, in detail, how the menus in this program are created, beginning with the **MenuDemo** constructor. It starts by creating a **JFrame** and setting its layout manager, size, and default close operation. (These operations are described in Chapter 31.) A **JLabel** is then constructed. It will be used to display a menu selection. Next, the menu bar is constructed and a reference to it is assigned to **jmb** by this statement:

```
// Create the menu bar.
JMenuBar jmb = new JMenuBar();
```

Then, the File menu **jmFile** and its menu entries are created by this sequence:

```
// Create the File menu.
JMenu jmFile = new JMenu("File");
JMenuItem jmiOpen = new JMenuItem("Open");
JMenuItem jmiClose = new JMenuItem("Close");
JMenuItem jmiSave = new JMenuItem("Save");
JMenuItem jmiExit = new JMenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. Next, the menu entries are added to the file menu by this sequence:

```
jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
```

Finally, the File menu is added to the menu bar with this line:

```
jmb.add(jmFile);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit. However, notice that a separator has been added before Exit. This visually separates Exit from the preceding three selections.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the same process.

Notice that **MenuDemo** implements the **ActionListener** interface and action events generated by a menu selection are handled by the **actionPerformed( )** method defined by **MenuDemo**. Therefore, the program adds **this** as the action listener for the menu items. Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the frame by the following line:

```
jfrm.setJMenuBar(jmb);
```

As mentioned, menu bars are not added to the content pane. They are added directly to the **JFrame**.

The **actionPerformed( )** method handles the action events generated by the menu. It obtains the action command string associated with the selection by calling **getActionCommand( )** on the event. It stores a reference to this string in **comStr**. Then, it tests the action command against "Exit", as shown here:

```
if(comStr.equals("Exit")) System.exit(0);
```

If the action command is "Exit", then the program terminates by calling **System.exit( )**. This method causes the immediate termination of a program and passes its argument as a status code to the calling process, which is usually the operating system or the browser. By convention, a status code of zero means normal termination. Anything else indicates that the program terminated abnormally. For all other menu selections, the choice is displayed.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the course of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: mnemonics and accelerators. As it applies to menus, a *mnemonic* defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed. An *accelerator* is a key that lets you select a menu item without having to first activate the menu.

A mnemonic can be specified for both **JMenuItem** and **JMenu** objects. There are two ways to set the mnemonic for **JMenuItem**. First, it can be specified when an object is constructed using this constructor:

JMenuItem(String *name*, int *mnem*)

In this case, the name is passed in *name* and the mnemonic is passed in *mnen*. Second, you can set the mnemonic by calling **setMnemonic( )**. To specify a mnemonic for **JMenu**, you must call **setMnemonic( )**. This method is inherited by both classes from **AbstractButton** and is shown next:

void setMnemonic(int *mnem*)

Here, *mnem* specifies the mnemonic. It should be one of the constants defined in **java.awt.event.KeyEvent**, such as **KeyEvent.VK_F** or **KeyEvent.VK_Z**. (There is another version of **setMnemonic( )** that takes a **char** argument, but it is considered obsolete.) Mnemonics are not case sensitive, so in the case of **VK_A**, typing either *a* or *A* will work.

By default, the first matching letter in the menu item will be underscored. In cases in which you want to underscore a letter other than the first match, specify the index of the letter as an argument to **setDisplayedMnemonicIndex( )**, which is inherited by both **JMenu** and **JMenuItem** from **AbstractButton**. It is shown here:

void setDisplayedMnemonicIndex(int *idx*)

The index of the letter to underscore is specified by *idx*.

An accelerator can be associated with a **JMenuItem** object. It is specified by calling **setAccelerator( )**, shown next:

void setAccelerator(KeyStroke *ks*)

Here, *ks* is the key combination that is pressed to select the menu item. **KeyStroke** is a class that contains several factory methods that construct various types of keystroke accelerators. The following are three examples:

static KeyStroke getKeyStroke(char *ch*)

static KeyStroke getKeyStroke(Character *ch*, int *modifier*)

static KeyStroke getKeyStroke(int *ch*, int *modifier*)

Here, *ch* specifies the accelerator character. In the first version, the character is specified as a **char** value. In the second, it is specified as an object of type **Character**. In the third, it is a value of type **KeyEvent**, previously described. The value of *modifier* must be one or more of the following constants, defined in the **java.awt.event.InputEvent** class:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Therefore, if you pass **VK_A** for the key character and **InputEvent.CTRL_DOWN_MASK** for the modifier, the accelerator key combination is CTRL-A.

The following sequence adds both mnemonics and accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can select the File menu by typing ALT-F. Then, you can use the mnemonics O, C, S, or E to select an option. Alternatively, you can directly select a File menu option by pressing CTRL-O, CTRL-C, CTRL-S, or CTRL-E. Figure 33-2 shows how this menu looks when activated.

```
// Create the File menu with mnemonics and accelerators.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);

JMenuItem jmiOpen = new JMenuItem("Open",
                                  KeyEvent.VK_O);
jmiOpen.setAccelerator(
```

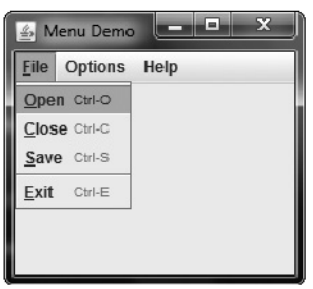


Figure 33-2 The File menu after adding mnemonics and accelerators

```
          KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiClose = new JMenuItem("Close",
                                   KeyEvent.VK_C);
jmiClose.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_C,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiSave = new JMenuItem("Save",
                                  KeyEvent.VK_S);
jmiSave.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                 InputEvent.CTRL_DOWN_MASK));

JMenuItem jmiExit = new JMenuItem("Exit",
                                  KeyEvent.VK_E);
jmiExit.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_E,
                                 InputEvent.CTRL_DOWN_MASK));
```

Add Images and Tooltips to Menu Items

You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using one of these constructors:

JMenuItem(Icon *image*)

JMenuItem(String *name*, Icon *image*)

The first creates a menu item that displays the image specified by *image*. The second creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created.

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

After this addition, the icon specified by **icon** will be displayed next to the text "About" when the Help menu is displayed. This is shown in Figure 33-3. You can also add an icon to a menu item after the item has been created by calling **setIcon( )**, which is inherited from

Figure 33-3 The About item with the addition of an icon

AbstractButton. You can specify the horizontal alignment of the image relative to the text by calling **setHorizontalTextPosition( )**.

You can specify a disabled icon, which is shown when the menu item is disabled, by calling **setDisabledIcon( )**. Normally, when a menu item is disabled, the default icon is shown in gray. If a disabled icon is specified, then that icon is displayed when the menu item is disabled.

A *tooltip* is a small message that describes an item. It is automatically displayed if the mouse remains over the item for a moment. You can add a tooltip to a menu item by calling **setToolTipText( )** on the item, specifying the text you want displayed. It is shown here:

void setToolTipText(String *msg*)

In this case, *msg* is the string that will be displayed when the tooltip is activated. For example, this creates a tooltip for the About item:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

As a point of interest, **setToolTipText( )** is inherited by **JMenuItem** from **JComponent**. This means you can add a tooltip to other types of components, such as a push button. You might want to try this on your own.

Use JRadioButtonMenuItem and JCheckBoxMenuItem

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, Swing defines two others: check boxes and radio buttons. These items can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes, including check boxes or radio buttons in a menu simply seems the most natural place for a specific set of features. Whatever your reason, Swing makes it easy to use check boxes and radio buttons in menus, and both are examined here.

To add a check box to a menu, create a **JCheckBoxMenuItem**. It defines several constructors. This is the one used in this chapter:

JCheckBoxMenuItem(String *name*)

Here, *name* specifies the name of the item. The initial state of the check box is unchecked. If you want to specify the initial state, you can use this constructor:

JCheckBoxMenuItem(String *name*, boolean *state*)

In this case, if *state* is **true**, the box is initially checked. Otherwise, it is cleared. **JCheckBoxMenuItem** also provides constructors that let you specify an icon. Here is one example:

JCheckBoxMenuItem(String *name*, Icon *icon*)

In this case, *name* specifies the name of the item and the image associated with the item is passed in *icon*. The item is initially unchecked. Other constructors are also supported.

Check boxes in menus work like stand-alone check boxes. For example, they generate action events and item events when their state changes. Check boxes are especially useful in menus when you have options that can be selected and you want to display their selected/deselected status.

A radio button can be added to a menu by creating an object of type **JRadioButtonMenuItem**. **JRadioButtonMenuItem** inherits **JMenuItem**. It provides a rich assortment of constructors. The ones used in this chapter are shown here:

JRadioButtonMenuItem(String *name*)

JRadioButtonMenuItem(String *name*, boolean *state*)

The first constructor creates an unselected radio button menu item that is associated with the name passed in *name*. The second lets you specify the initial state of the button. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Other constructors let you specify an icon. Here is one example:

JRadioButtonMenuItem(String *name*, Icon *icon*, boolean *state*)

This creates a radio button menu item that is associated with the name passed in *name* and the image passed in *icon*. If *state* is **true**, the button is initially selected. Otherwise, it is deselected. Several other constructors are supported.

A **JRadioButtonMenuItem** works like a stand-alone radio button, generating item and action events. Like stand-alone radio buttons, menu-based radio buttons must be put into a button group in order for them to exhibit mutually exclusive selection behavior.

Because both **JCheckBoxMenuItem** and **JRadioButtonMenuItem** inherit **JMenuItem**, each has all of the functionality provided by **JMenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

To try check box and radio button menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check boxes for the Colors submenu and radio buttons for the Priority submenu. After making the substitution, the Options menu will look like those shown in Figure 33-4.

```
// Create the Options menu.
JMenu jmOptions = new JMenu("Options");

// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
```

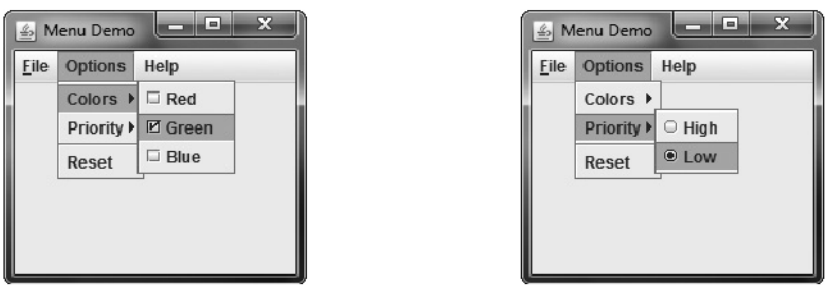


Figure 33-4 The effects of check box (a) and radio button (b) menu items

```
// Use check boxes for colors. This allows
// the user to select more than one color.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");

// Use radio buttons for the priority setting.
// This lets the menu show which priority is used
// but also ensures that one and only one priority
// can be selected at any one time. Notice that
// the High radio button is initially selected.
JRadioButtonMenuItem jmiHigh =
  new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
  new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// Create button group for the radio button menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);
```


Create a Popup Menu

A popular alternative or addition to the menu bar is the popup menu. Typically, a popup menu is activated by clicking the right mouse button when over a component. Popup menus are supported in Swing by the **JPopupMenu** class. **JPopupMenu** has two constructors. In this chapter, only the default constructor is used:

JPopupMenu( )

It creates a default popup menu. The other constructor lets you specify a title for the menu. Whether this title is displayed is subject to the look and feel.

In general, popup menus are constructed like regular menus. First, create a **JPopupMenu** object, and then add menu items to it. Menu item selections are also handled in the same way: by listening for action events. The main difference between a popup menu and regular menu is the activation process.

Activating a popup menu requires three steps.

1. You must register a listener for mouse events.
2. Inside the mouse event handler, you must watch for the popup trigger.
3. When a popup trigger is received, you must show the popup menu by calling **show( )**.

Let's examine each of these steps closely.

A popup menu is normally activated by clicking the right mouse button when the mouse pointer is over a component for which a popup menu is defined. Thus, the *popup trigger* is usually caused by right-clicking the mouse on a popup menu–enabled component. To listen for the popup trigger, implement the **MouseListener** interface and then register the listener by calling the **addMouseListener( )** method. As described in Chapter 24, **MouseListener** defines the methods shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

Of these, two are very important relative to the popup menu: **mousePressed( )** and **mouseReleased( )**. Depending on the installed look and feel, either of these two events can trigger a popup menu. For this reason, it is often easier to use a **MouseAdapter** to implement the **MouseListener** interface and simply override **mousePressed( )** and **mouseReleased( )**.

The **MouseEvent** class defines several methods, but only four are commonly needed when activating a popup menu. They are shown here:

int getX( )

int getY( )

boolean isPopupTrigger( )

Component getComponent( )

The current X,Y location of the mouse relative to the source of the event is found by calling **getX( )** and **getY( )**. These are used to specify the upper-left corner of the popup menu when it is displayed. The **isPopupTrigger( )** method returns **true** if the mouse event represents a popup trigger and **false** otherwise. You will use this method to determine when to pop up the menu. To obtain a reference to the component that generated the mouse event, call **getComponent( )**.

To actually display the popup menu, call the **show( )** method defined by **JPopupMenu**, shown next:

void show(Component *invoker*, int *upperX*, int *upperY*)

Here, *invoker* is the component relative to which the menu will be displayed. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to *invoker*. A common way to obtain the invoker is to call **getComponent( )** on the event object passed to the mouse event handler.

The preceding theory can be put into practice by adding a popup Edit menu to the **MenuDemo** program shown at the start of this chapter. This menu will have three items called Cut, Copy, and Paste. Begin by adding the following instance variable to **MenuDemo**:

```
JPopupMenu jpu;
```

The **jpu** variable will hold a reference to the popup menu.

Next, add the following code sequence to the **MenuDemo** constructor:

```
// Create an Edit popup menu.
jpu = new JPopupMenu();

// Create the popup menu items.
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// Add the menu items to the popup menu.
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// Add a listener for the popup trigger.
jfrm.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
  public void mouseReleased(MouseEvent me) {
    if(me.isPopupTrigger())
      jpu.show(me.getComponent(), me.getX(), me.getY());
  }
});
```

This sequence begins by constructing an instance of **JPopupMenu** and storing it in **jpu**. Then, it creates the three menu items, Cut, Copy, and Paste, in the usual way, and adds them to **jpu**. This finishes the construction of the popup Edit menu. Popup menus are not added to the menu bar or any other object.

Next, a **MouseListener** is added by creating an anonymous inner class. This class is based on the **MouseAdapter** class, which means that the listener need only override those

methods that are relevant to the popup menu: **mousePressed( )** and **mouseReleased( )**. The adapter provides default implementations of the other **MouseListener** methods. Notice that the mouse listener is added to **jfrm**. This means that a right-button click inside any part of the content pane will trigger the popup menu.

The **mousePressed( )** and **mouseReleased( )** methods call **isPopupTrigger( )** to determine if the mouse event is a popup trigger event. If it is, the popup menu is displayed by calling **show( )**. The invoker is obtained by calling **getComponent( )** on the mouse event. In this case, the invoker will be the content pane. The X,Y coordinates of the upper-left corner are obtained by calling **getX( )** and **getY( )**. This makes the menu pop up with its upper-left corner directly under the mouse pointer.

Finally, you also need to add these action listeners to the program. They handle the action events fired when the user selects an item from the popup menu.

```
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
```

After you have made these additions, the popup menu can be activated by clicking the right mouse button anywhere inside the content pane of the application. Figure 33-5 shows the result.

One other point about the preceding example. Because the invoker of the popup menu is always **jfrm**, in this case, you could pass it explicitly rather than calling **getComponent( )**. To do so, you must make **jfrm** into an instance variable of the **MenuDemo** class (rather than a local variable) so that it is accessible to the inner class. Then you can use this call to **show( )** to display the popup menu:

```
jpu.show(jfrm, me.getX(), me.getY());
```

Although this works in this example, the advantage of using **getComponent( )** is that the popup menu will automatically pop up relative to the invoking component. Thus, the same code could be used to display any popup menu relative to its invoking object.

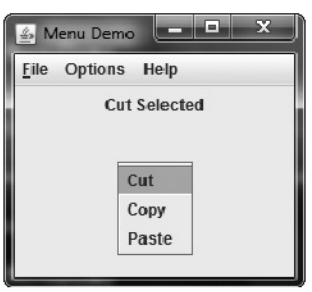


Figure 33-5 A popup Edit menu

Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. A toolbar contains a list of buttons (or other components) that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without needing to drop through a menu. Typically, toolbar buttons show icons rather than text, although either or both are allowed. Furthermore, tooltips are often associated with icon-based toolbar buttons. Toolbars can be positioned on any side of a window by dragging the toolbar, or they can be dragged out of the window entirely, in which case they become free floating.

In Swing, toolbars are instances of the **JToolBar** class. Its constructors enable you to create a toolbar with or without a title. You can also specify the layout of the toolbar, which will be either horizontal or vertical. The **JToolBar** constructors are shown here:

JToolBar( )

JToolBar(String *title*)

JToolBar(int *how*)

JToolBar(String *title*, int *how*)

The first constructor creates a horizontal toolbar with no title. The second creates a horizontal toolbar with the title specified by *title*. The title will show only when the toolbar is dragged out of its window. The third creates a toolbar that is oriented as specified by *how*. The value of *how* must be either **JToolBar.VERTICAL** or **JToolBar.HORIZONTAL**. The fourth constructor creates a toolbar that has the title specified by *title* and is oriented as specified by *how*.

A toolbar is typically used with a window that uses a border layout. There are two reasons for this. First, it allows the toolbar to be initially positioned along one of the four border positions. Frequently, the top position is used. Second, it allows the toolbar to be dragged to any side of the window.

In addition to dragging the toolbar to different locations within a window, you can also drag it out of the window. Doing so creates an *undocked* toolbar. If you specify a title when you create the toolbar, then that title will be shown when the toolbar is undocked.

You add buttons (or other components) to a toolbar in much the same way that you add them to a menu bar. Simply call **add( )**. The components are shown in the toolbar in the order in which they are added.

Once you have created a toolbar, you *do not* add it to the menu bar (if one exists). Instead, you add it to the window container. As mentioned, typically you will add a toolbar to the top (that is, north) position of a border layout, using a horizontal orientation. The component that will be affected is added to the center of the border layout. Using this approach causes the program to begin running with the toolbar in the expected location. However, you can drag the toolbar to any of the other positions. Of course, you can also drag the toolbar out of the window.

To illustrate the toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: set a breakpoint, clear a breakpoint, and resume program execution. Three steps are needed to add the toolbar.

First, remove this line from the program.

```
jfrm.setLayout(new FlowLayout());
```

By removing this line, the **JFrame** automatically uses a border layout.

Second, because **BorderLayout** is being used, change the line that adds the label **jlab** to the frame, as shown next:

```
jfrm.add(jlab, BorderLayout.CENTER);
```

This line explicitly adds **jlab** to the center of the border layout. (Explicitly specifying the center position is technically not necessary because, by default, components are added to the center when a border layout is used. However, explicitly specifying the center makes it clear to anyone reading the code that a border layout is being used and that **jlab** goes in the center.)

Next, add the following code, which creates the Debug toolbar.

```
// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Debug");

// Load the images.
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");

// Create the toolbar buttons.
JButton jbtnSet = new JButton(set);
jbtnSet.setActionCommand("Set Breakpoint");
jbtnSet.setToolTipText("Set Breakpoint");

JButton jbtnClear = new JButton(clear);
jbtnClear.setActionCommand("Clear Breakpoint");
jbtnClear.setToolTipText("Clear Breakpoint");

JButton jbtnResume = new JButton(resume);
jbtnResume.setActionCommand("Resume");
jbtnResume.setToolTipText("Resume");

// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

Let's look at this code closely. First, a **JToolBar** is created and given the title "Debug". Then, a set of **ImageIcon** objects are created that hold the images for the toolbar buttons. Next, three toolbar buttons are created. Notice that each has an image, but no text. Also, each is explicitly given an action command and a tooltip. The action commands are set

because the buttons are not given names when they are constructed. Tooltips are especially useful when applied to icon-based toolbar components because sometimes it's hard to design images that are intuitive to all users. The buttons are then added to the toolbar, and the toolbar is added to the north side of the border layout of the frame.

Finally, add the action listeners for the toolbar, as shown here:

```
// Add the toolbar action listeners.
jbtnSet.addActionListener(this);
jbtnClear.addActionListener(this);
jbtnResume.addActionListener(this);
```

Each time the user presses a toolbar button, an action event is fired, and it is handled in the same way as the other menu-related events. Figure 33-6 shows the toolbar in action.

Use Actions

Often, a toolbar and a menu item contain items in common. For example, the same functions provided by the Debug toolbar in the preceding example might also be offered through a menu selection. In such a case, selecting an option (such as setting a breakpoint) causes the same action to occur, independently of whether the menu or the toolbar was used. Also, both the toolbar button and the menu item would (most likely) use the same icon. Furthermore, when a toolbar button is disabled, the corresponding menu item would also need to be disabled. Such a situation would normally lead to a fair amount of duplicated, interdependent code, which is less than optimal. Fortunately, Swing provides a solution: the *action*.

An action is an instance of the **Action** interface. **Action** extends the **ActionListener** interface and provides a means of combining state information with the **actionPerformed( )** event handler. This combination allows one action to manage two or more components. For example, an action lets you centralize the control and handling of a toolbar button and a menu item. Instead of having to duplicate code, your program need only create an action that automatically handles both components.

Because **Action** extends **ActionListener**, an action must provide an implementation of the **actionPerformed( )** method. This handler will process the action events generated by the objects linked to the action.

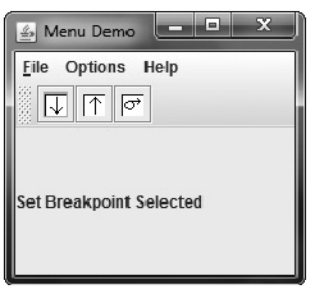


Figure 33-6 The Debug toolbar in action

In addition to the inherited **actionPerformed( )** method, **Action** defines several methods of its own. One of particular interest is **putValue( )**. It sets the value of the various properties associated with an action and is shown here:

void putValue(String *key*, Object *val*)

It assigns *val* to the property specified by *key* that represents the desired property. Although not used by the example that follows, it is helpful to note that **Action** also supplies the **getValue( )** method that obtains a specified property. It is shown here:

Object getValue(String *key*)

It returns a reference to the property specified by *key*.

The key values used by **putValue( )** and **getValue( )** include those shown here:

Key Value	Description
static final String ACCELERATOR_KEY	Represents the accelerator property. Accelerators are specified as **KeyStroke** objects.
static final String ACTION_COMMAND_KEY	Represents the action command property. An action command is specified as a string.
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Represents the index of the character displayed as the mnemonic. This is an **Integer** value.
static final String LARGE_ICON_KEY	Represents the large icon associated with the action. The icon is specified as an object of type **Icon**.
static final String LONG_DESCRIPTION	Represents a long description of the action. This description is specified as a string.
static final String MNEMONIC_KEY	Represents the mnemonic property. A mnemonic is specified as a **KeyEvent** constant.
static final String NAME	Represents the name of the action (which also becomes the name of the button or menu item to which the action is linked). The name is specified as a string.
static final String SELECTED_KEY	Represents the selection status. If set, the item is selected. The state is represented by a **Boolean** value.
static final String SHORT_DESCRIPTION	Represents the tooltip text associated with the action. The tooltip text is specified as a string.
static final String SMALL_ICON	Represents the icon associated with the action. The icon is specified as an object of type **Icon**.

For example, to set the mnemonic to the letter *X*, use this call to **putValue( )**:

```
actionOb.putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_X));
```

One **Action** property that is not accessible through **putValue( )** and **getValue( )** is the enabled/disabled status. For this, you use the **setEnabled( )** and **isEnabled( )** methods. They are shown here:

void setEnabled(boolean *enabled*)

boolean isEnabled( )

For **setEnabled( )**, if *enabled* is **true**, the action is enabled. Otherwise, it is disabled. If the action is enabled, **isEnabled( )** returns **true**. Otherwise, it returns **false**.

Although you can implement all of the **Action** interface yourself, you won't usually need to. Instead, Swing provides a partial implementation called **AbstractAction** that you can extend. By extending **AbstractAction**, you need implement only one method: **actionPerformed( )**. The other **Action** methods are provided for you. **AbstractAction** provides three constructors. The one used in this chapter is shown here:

AbstractAction(String *name*, Icon *image*)

It constructs an **AbstractAction** that has the name specified by *name* and the icon specified by *image*.

Once you have created an action, it can be added to a **JToolBar** and used to construct a **JMenuItem**. To add an action to a **JToolBar**, use this version of **add( )**:

void add(Action *actObj*)

Here, *actObj* is the action that is being added to the toolbar. The properties defined by *actObj* are used to create a toolbar button. To create a menu item from an action, use this **JMenuItem** constructor:

JMenuItem(Action *actObj*)

Here, *actObj* is the action used to construct a menu item according to its properties.

NOTE In addition to **JToolBar** and **JMenuItem**, actions are also supported by several other Swing components, such as **JPopupMenu**, **JButton**, **JRadioButton**, and **JCheckBox**. **JRadioButtonMenuItem** and **JCheckBoxMenuItem** also support actions.

To illustrate the benefit of actions, we will use them to manage the Debug toolbar created in the previous section. We will also add a Debug submenu under the Options main menu. The Debug submenu will contain the same selections as the Debug toolbar: Set Breakpoint, Clear Breakpoint, and Resume. The same actions that support these items in the toolbar will also support these items in the menu. Therefore, instead of having to create duplicate code to handle both the toolbar and menu, both are handled by the actions.

Begin by creating an inner class called **DebugAction** that extends **AbstractAction**, as shown here:

```
// A class to create an action for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
             KeyStroke.getKeyStroke(accel,
                                    InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, new Integer(mnem));
    putValue(SHORT_DESCRIPTION, tTip);
  }

  // Handle events for both the toolbar and the
  // Debug menu.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Toggle the enabled status of the
    // Set and Clear Breakpoint options.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}
```

DebugAction extends **AbstractAction**. It creates an action class that will be used to define the properties associated with the Debug menu and toolbar. Its constructor has five parameters that let you specify the following items:

- Name
- Icon
- Mnemonic
- Accelerator
- Tooltip

The first two are passed to **AbstractAction**'s constructor via **super**. The other three properties are set through calls to **putValue( )**.

The **actionPerformed( )** method of **DebugAction** handles events for the action. This means that when an instance of **DebugAction** is used to create a toolbar button and a menu

item, events generated by either of those components are handled by the **actionPerformed( )** method in **DebugAction**. Notice that this handler displays the selection in **jlab**. In addition, if the Set Breakpoint option is selected, then the Clear Breakpoint option is enabled and the Set Breakpoint option is disabled. If the Clear Breakpoint option is selected, then the Set Breakpoint option is enabled and the Clear Breakpoint option is disabled. This illustrates how an action can be used to enable or disable a component. When an action is disabled, it is disabled for all uses of that action. In this case, if Set Breakpoint is disabled, then it is disabled both in the toolbar and in the menu.

Next, add these **DebugAction** instance variables to **MenuDemo**:

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Next, create three **ImageIcon**s that represent the Debug options, as shown here:

```
// Load the images for the actions.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

Now, create the actions that manage the Debug options, as shown here:

```
// Create actions.
setAct =
  new DebugAction("Set Breakpoint",
                  setIcon,
                  KeyEvent.VK_S,
                  KeyEvent.VK_B,
                  "Set a break point.");

clearAct =
  new DebugAction("Clear Breakpoint",
                  clearIcon,
                  KeyEvent.VK_C,
                  KeyEvent.VK_L,
                  "Clear a break point.");

resumeAct =
  new DebugAction("Resume",
                  resumeIcon,
                  KeyEvent.VK_R,
                  KeyEvent.VK_R,
                  "Resume execution after breakpoint.");

// Initially disable the Clear Breakpoint option.
clearAct.setEnabled(false);
```

Notice that the accelerator for Set Breakpoint is B and the accelerator for Clear Breakpoint is L. The reason these keys are used rather than S and C is that these keys are already allocated by the File menu for Save and Close. However, they can still be used as mnemonics because each mnemonic is localized to its own menu. Also notice that the action that

represents Clear Breakpoint is initially disabled. It will be enabled only after a breakpoint has been set.

Next, use the actions to create buttons for the toolbar and then add those buttons to the toolbar, as shown here:

```
// Create the toolbar buttons by using the actions.
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Breakpoints");

// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

Finally, create the Debug menu, as shown next:

```
// Now, create a Debug menu that goes under the Options
// menu bar item. Use the actions to create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);
```

After making these changes and additions, the actions that you created will be used to manage both the Debug menu and the toolbar. Thus, changing a property in the action (such as disabling it) will affect all uses of that action. The program will now look as shown in Figure 33-7.

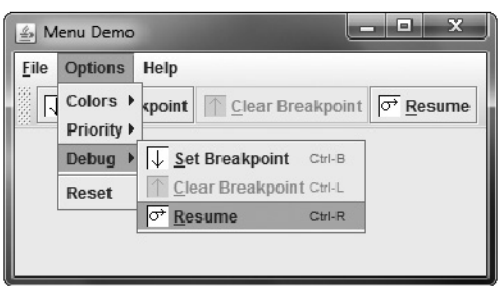


Figure 33-7 Using actions to manage the Debug toolbar and menu

Put the Entire MenuDemo Program Together

Throughout the course of this discussion, many changes and additions have been made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **jmb**, **jmFile**, and **jtb**, have been made into instance variables.

```
// The complete MenuDemo program.

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

class MenuDemo implements ActionListener {

  JLabel jlab;

  JMenuBar jmb;

  JToolBar jtb;

  JPopupMenu jpu;

  DebugAction setAct;
  DebugAction clearAct;
  DebugAction resumeAct;

  MenuDemo() {
    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Complete Menu Demo");

    // Use default border layout.

    // Give the frame an initial size.
    jfrm.setSize(360, 200);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a label that will display the menu selection.
    jlab = new JLabel();

    // Create the menu bar.
    jmb = new JMenuBar();

    // Make the File menu.
    makeFileMenu();
```

```
    // Construct the Debug actions.
    makeActions();

    // Make the toolbar.
    makeToolBar();

    // Make the Options menu.
    makeOptionsMenu();

    // Make the Help menu.
    makeHelpMenu();

    // Make the Edit popup menu.
    makeEditPUMenu();

    // Add a listener for the popup trigger.
    jfrm.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
      }
      public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
          jpu.show(me.getComponent(), me.getX(), me.getY());
      }
    });

    // Add the label to the center of the content pane.
    jfrm.add(jlab, SwingConstants.CENTER);

    // Add the toolbar to the north position of
    // the content pane.
    jfrm.add(jtb, BorderLayout.NORTH);

    // Add the menu bar to the frame.
    jfrm.setJMenuBar(jmb);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle menu item action events.
  // This does NOT handle events generated
  // by the Debug options.
  public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();

    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);

    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
  }
```

```
// An action class for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
  public DebugAction(String name, Icon image, int mnem,
                     int accel, String tTip) {
    super(name, image);
    putValue(ACCELERATOR_KEY,
             KeyStroke.getKeyStroke(accel,
                                    InputEvent.CTRL_DOWN_MASK));
    putValue(MNEMONIC_KEY, new Integer(mnem));
    putValue(SHORT_DESCRIPTION, tTip);
  }

  // Handle events for both the toolbar and the
  // Debug menu.
  public void actionPerformed(ActionEvent ae) {
    String comStr = ae.getActionCommand();

    jlab.setText(comStr + " Selected");

    // Toggle the enabled status of the
    // Set and Clear Breakpoint options.
    if(comStr.equals("Set Breakpoint")) {
      clearAct.setEnabled(true);
      setAct.setEnabled(false);
    } else if(comStr.equals("Clear Breakpoint")) {
      clearAct.setEnabled(false);
      setAct.setEnabled(true);
    }
  }
}

// Create the File menu with mnemonics and accelerators.
void makeFileMenu() {
  JMenu jmFile = new JMenu("File");
  jmFile.setMnemonic(KeyEvent.VK_F);

  JMenuItem jmiOpen = new JMenuItem("Open",
                                    KeyEvent.VK_O);
  jmiOpen.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                   InputEvent.CTRL_DOWN_MASK));

  JMenuItem jmiClose = new JMenuItem("Close",
                                     KeyEvent.VK_C);
  jmiClose.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_C,
                                   InputEvent.CTRL_DOWN_MASK));

  JMenuItem jmiSave = new JMenuItem("Save",
                                    KeyEvent.VK_S);
  jmiSave.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                   InputEvent.CTRL_DOWN_MASK));
```

```
    JMenuItem jmiExit = new JMenuItem("Exit",
                                      KeyEvent.VK_E);
    jmiExit.setAccelerator(
              KeyStroke.getKeyStroke(KeyEvent.VK_E,
                                     InputEvent.CTRL_DOWN_MASK));

    jmFile.add(jmiOpen);
    jmFile.add(jmiClose);
    jmFile.add(jmiSave);
    jmFile.addSeparator();
    jmFile.add(jmiExit);
    jmb.add(jmFile);

    // Add the action listeners for the File menu.
    jmiOpen.addActionListener(this);
    jmiClose.addActionListener(this);
    jmiSave.addActionListener(this);
    jmiExit.addActionListener(this);
  }

  // Create the Options menu.
  void makeOptionsMenu() {
    JMenu jmOptions = new JMenu("Options");

    // Create the Colors submenu.
    JMenu jmColors = new JMenu("Colors");

    // Use check boxes for colors. This allows
    // the user to select more than one color.
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

    // Add the items to the Colors menu.
    jmColors.add(jmiRed);
    jmColors.add(jmiGreen);
    jmColors.add(jmiBlue);
    jmOptions.add(jmColors);

    // Create the Priority submenu.
    JMenu jmPriority = new JMenu("Priority");

    // Use radio buttons for the priority setting.
    // This lets the menu show which priority is used
    // but also ensures that one and only one priority
    // can be selected at any one time. Notice that
    // the High radio button is initially selected.
    JRadioButtonMenuItem jmiHigh =
      new JRadioButtonMenuItem("High", true);
    JRadioButtonMenuItem jmiLow =
      new JRadioButtonMenuItem("Low");

    // Add the items to the Priority menu.
    jmPriority.add(jmiHigh);
```

```
    jmPriority.add(jmiLow);
    jmOptions.add(jmPriority);

    // Create a button group for the radio button
    //  menu items.
    ButtonGroup bg = new ButtonGroup();
    bg.add(jmiHigh);
    bg.add(jmiLow);

    // Now, create a Debug submenu that goes under
    // the Options menu bar item. Use actions to
    // create the items.
    JMenu jmDebug = new JMenu("Debug");
    JMenuItem jmiSetBP = new JMenuItem(setAct);
    JMenuItem jmiClearBP = new JMenuItem(clearAct);
    JMenuItem jmiResume = new JMenuItem(resumeAct);

    // Add the items to the Debug menu.
    jmDebug.add(jmiSetBP);
    jmDebug.add(jmiClearBP);
    jmDebug.add(jmiResume);
    jmOptions.add(jmDebug);

    // Create the Reset menu item.
    JMenuItem jmiReset = new JMenuItem("Reset");
    jmOptions.addSeparator();
    jmOptions.add(jmiReset);

    // Finally, add the entire options menu to
    // the menu bar
    jmb.add(jmOptions);

    // Add the action listeners for the Options menu,
    // except for those supported by the Debug menu.
    jmiRed.addActionListener(this);
    jmiGreen.addActionListener(this);
    jmiBlue.addActionListener(this);
    jmiHigh.addActionListener(this);
    jmiLow.addActionListener(this);
    jmiReset.addActionListener(this);
  }

  // Create the Help menu.
  void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // Add an icon to the About menu item.
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

    JMenuItem jmiAbout = new JMenuItem("About", icon);
    jmiAbout.setToolTipText("Info about the MenuDemo program.");
    jmHelp.add(jmiAbout);
    jmb.add(jmHelp);
```

```
    // Add action listener for About.
    jmiAbout.addActionListener(this);
  }

  // Construct the actions needed by the Debug menu
  // and toolbar.
  void makeActions() {
    // Load the images for the actions.
    ImageIcon setIcon = new ImageIcon("setBP.gif");
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");
    ImageIcon resumeIcon = new ImageIcon("resume.gif");

    // Create actions.
    setAct =
      new DebugAction("Set Breakpoint",
                      setIcon,
                      KeyEvent.VK_S,
                      KeyEvent.VK_B,
                      "Set a break point.");

    clearAct =
      new DebugAction("Clear Breakpoint",
                      clearIcon,
                      KeyEvent.VK_C,
                      KeyEvent.VK_L,
                      "Clear a break point.");

    resumeAct =
      new DebugAction("Resume",
                      resumeIcon,
                      KeyEvent.VK_R,
                      KeyEvent.VK_R,
                      "Resume execution after breakpoint.");

    // Initially disable the Clear Breakpoint option.
    clearAct.setEnabled(false);
  }

  // Create the Debug toolbar.
  void makeToolBar() {
    // Create the toolbar buttons by using the actions.
    JButton jbtnSet = new JButton(setAct);
    JButton jbtnClear = new JButton(clearAct);
    JButton jbtnResume = new JButton(resumeAct);

    // Create the Debug toolbar.
    jtb = new JToolBar("Breakpoints");

    // Add the buttons to the toolbar.
    jtb.add(jbtnSet);
    jtb.add(jbtnClear);
    jtb.add(jbtnResume);
  }
```

```
  // Create the Edit popup menu.
  void makeEditPUMenu() {
    jpu = new JPopupMenu();

    // Create the popup menu items
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // Add the menu items to the popup menu.
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);

    // Add the Edit popup menu action listeners.
    jmiCut.addActionListener(this);
    jmiCopy.addActionListener(this);
    jmiPaste.addActionListener(this);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new MenuDemo();
      }
    });
  }
}
```


Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, it supplies dialog classes, such as **JOptionPane** and **JDialog**, that you can use to streamline the construction of dialog windows. It also provides additional controls beyond those introduced in Chapter 31. Two you will want to explore are **JSpinner** (which creates a spin control) and **JFormattedTextField** (which supports formatted text). You will also want to experiment with defining your own models for the various components. Frankly, the best way to become familiar with Swing's capabilities is to experiment with it.

This page has been intentionally left blank

PART

IV

Introducing GUI Programming with JavaFX

This page has been intentionally left blank

CHAPTER

34 Introducing JavaFX GUI Programming

Like all successful languages, Java continues to evolve and improve. This also applies to its libraries. One of the most important examples of this evolutionary process is found in its GUI frameworks. As explained earlier in the book, the original GUI framework was the AWT. Because of its several limitations, it was soon followed by Swing, which offered a far superior approach to creating GUIs. Swing was so successful that it has remained the primary Java GUI framework for over a decade. (And a decade is a long time in the fast-moving world of programming!) However, Swing was designed when the enterprise application dominated software development. Today, consumer applications, and especially mobile apps, have risen in importance, and such applications often demand a GUI that has "visual sparkle." Furthermore, no matter the type of application, the trend is toward more exciting visual effects. To better handle these types of GUIs, a new approach was needed, and this lead to the creation of JavaFX. JavaFX is Java's next-generation client platform and GUI framework.

JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs. As such, it is a very large system, and, as was the case with Swing discussed in Part III, it is not possible to describe it fully in this book. Instead, the purpose of this and the next two chapters is to introduce several of its key features and techniques. Once you understand the fundamentals, you will find it easy to explore other aspects of JavaFX on your own.

One question that naturally arises relating to JavaFX is this: Is JavaFX designed as a replacement for Swing? The answer is a qualified Yes. However, given the large amount of Swing legacy code and the legions of programmers who know how to program for Swing, Swing will be in use for a very long time. This is especially true for enterprise applications. Nevertheless, JavaFX has clearly been positioned as the platform of the future. It is expected that, over the next few years, JavaFX will supplant Swing for new projects. JavaFX is something that no Java programmer can afford to ignore.

Before continuing, it is important to mention that the development of JavaFX occurred in two main phases. The original JavaFX was based on a scripting language called *JavaFX Script*. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX

also supports FXML, which can be (but is not required to be) used to specify the user interface. JavaFX has been bundled with Java since JDK 7, update 4. The latest version of JavaFX is JavaFX 8, which is bundled with JDK 8. (The version number is 8 to align with the JDK version. Thus, the numbers 3 through 7 were skipped.) Because, at the time of this writing, JavaFX 8 represents the latest version of JavaFX, it is the version of JavaFX discussed here. Furthermore, when the term *JavaFX* is used in this and the following chapters, it refers to JavaFX 8.

NOTE This and the following two chapters assume that you have a basic understanding of event handling as introduced in Chapter 24, and Swing fundamentals, as described by the preceding three chapters.

JavaFX Basic Concepts

In general, the JavaFX framework has all of the good features of Swing. For example, JavaFX is lightweight. It can also support an MVC architecture. Much of what you already know about creating GUIs using Swing is conceptually applicable to JavaFX. That said, there are significant differences between the two.

From a programmer's point of view, the first differences you notice between JavaFX and Swing are the organization of the framework and the relationship of the main components. Simply put, JavaFX offers a more streamlined, easier-to-use, updated approach. JavaFX also greatly simplifies the rendering of objects because it handles repainting automatically. It is no longer necessary for your program to handle this task manually. The preceding is not intended to imply that Swing is poorly designed. It is not. It is just that the art and science of programming has moved forward, and JavaFX has received the benefits of that evolution. Simply put, JavaFX facilitates a more visually dynamic approach to GUIs.

The JavaFX Packages

The JavaFX elements are contained in packages that begin with the **javafx** prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples: **javafx.application**, **javafx.stage**, **javafx.scene**, and **javafx.scene.layout**. Although we will only use a few of these packages in this chapter, you will want to spend some time browsing their capabilities. JavaFX offers a wide array of functionality.

The Stage and Scene Classes

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. Thus, loosely speaking, a stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes. To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**. Let's look a bit more closely at these two classes.

Stage is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of **Scene**.

Nodes and Scene Graphs

The individual elements of a scene are called *nodes*. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves. The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*.

There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region**, and **Control**, to name a few.

Layouts

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout. Several other layouts, such as **BorderPane** (which is similar to the AWT's **BorderLayout**), are available. The layout panes are packaged in **javafx.scene.layout**.

The Application Class and the Life-cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in **javafx.application**. Thus, your application class will extend **Application**. The **Application** class defines three life-cycle methods that your application can override. These are called **init( )**, **start( )**, and **stop( )**, and are shown here, in the order in which they are called:

```
void init( )
abstract void start(Stage primaryStage)
void stop( )
```

The **init( )** method is called when the application begins execution. It is used to perform various initializations. As will be explained, however, it *cannot* be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

The **start( )** method is called after **init( )**. This is where your application begins and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. (You can also create other stages, but you won't need to for simple applications.) Notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the **stop( )** method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

Launching a JavaFX Application

To start a free-standing JavaFX application, you must call the **launch( )** method defined by **Application**. It has two forms. Here is the one used in this chapter:

public static void launch(String ... *args*)

Here, *args* is a possibly empty list of strings that typically specify command-line arguments. When called, **launch( )** causes the application to be constructed, followed by calls to **init( )** and **start( )**. The **launch( )** method will not return until after the application has terminated. This version of **launch( )** starts the subclass of **Application** from which **launch( )** is called. The second form of **launch( )** lets you specify a class other than the enclosing class to start.

Before moving on, it is necessary to make an important point: JavaFX applications that have been packaged by using the **javafxpackager** tool (or its equivalent in an IDE) do not need to include a call to **launch( )**. However, its inclusion often simplifies the test/debug cycle, and it lets the program be used without the creation of a JAR file. Thus, it is included in all of the JavaFX programs in this book.

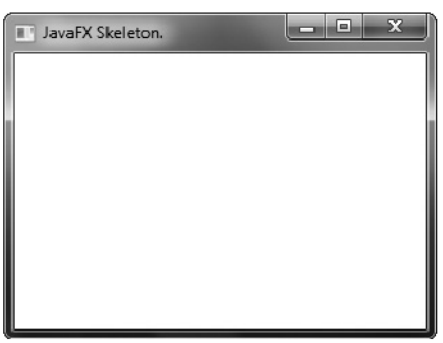
A JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like. In addition to showing the general form of a JavaFX application, the skeleton also illustrates how to launch the application and demonstrates when the life-cycle methods are called. A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

```
// A JavaFX application skeleton.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

  public static void main(String[] args) {

    System.out.println("Launching JavaFX application.");

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the init() method.
  public void init() {
```

```
    System.out.println("Inside the init() method.");
  }

  // Override the start() method.
  public void start(Stage myStage) {

    System.out.println("Inside the start() method.");

    // Give the stage a title.
    myStage.setTitle("JavaFX Skeleton.");

    // Create a root node. In this case, a flow layout pane
    // is used, but several alternatives exist.
    FlowPane rootNode = new FlowPane();

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Show the stage and its scene.
    myStage.show();

  }

  // Override the stop() method.
  public void stop() {
    System.out.println("Inside the stop() method.");
  }
}
```

Although the skeleton is quite short, it can be compiled and run. It produces the window shown here:



It also produces the following output on the console:

```
Launching JavaFX application.
Inside the init() method.
Inside the start() method.
```

When you close the window, this message is displayed on the console:

```
Inside the stop() method.
```

Of course, in a real program, the life-cycle methods would not normally output anything to **System.out**. They do so here simply to illustrate when each method is called. Furthermore, as explained earlier, you will need to override the **init( )** and **stop( )** methods only if your application must perform special startup or shutdown actions. Otherwise, you can use the default implementations of these methods provided by the **Application** class.

Let's examine this program in detail. It begins by importing four packages. The first is **javafx.application**, which contains the **Application** class. The **Scene** class is packaged in **javafx.scene**, and **Stage** is packaged in **javafx.stage**. The **javafx.scene.layout** package provides several layout panes. The one used by the program is **FlowPane**.

Next, the application class **JavaFXSkel** is created. Notice that it extends **Application**. As explained, **Application** is the class from which all JavaFX applications are derived. **JavaFXSkel** contains two methods. The first is **main( )**. It is used to launch the application via a call to **launch( )**. Notice that the **args** parameter to **main( )** is passed to the **launch( )** method. Although this is a common approach, you can pass a different set of parameters to **launch( )**, or none at all. One other point: As explained earlier, **launch( )** is required by a free-standing application, but not in other cases. When it is not needed, **main( )** is also not needed. However, for reasons already explained, both **main( )** and **launch( )** are included in the JavaFX programs in this book.

When the application begins, the **init( )** method is called first by the JavaFX run-time system. For the sake of illustration, it simply displays a message on **System.out**, but it would normally be used to initialize some aspect of the application. Of course, if no initialization is required, it is not necessary to override **init( )** because an empty, default implementation is provided. It is important to emphasize that **init( )** cannot be used to create the stage or scene portions of a GUI. Rather, these items should be constructed and displayed by the **start( )** method.

After **init( )** finishes, the **start( )** method executes. It is here that the initial scene is created and set to the primary stage. Let's look at this method line-by-line. First, notice that **start( )** has a parameter of type **Stage**. When **start( )** is called, this parameter will receive a reference to the primary stage of the application. It is to this stage that you will set a scene for the application.

After displaying a message on the console that **start( )** has begun execution, it sets the title of the stage using this call to **setTitle( )**:

```
myStage.setTitle("JavaFX Skeleton.");
```

Although this step is not necessarily required, it is customary for stand-alone applications. This title becomes the name of the main application window.

Next, a root node for a scene is created. The root node is the only node in a scene graph that does not have a parent. In this case, a **FlowPane** is used for the root node, but there are several other classes that can be used for the root.

```
FlowPane rootNode = new FlowPane();
```

As mentioned, a **FlowPane** is a layout manager that uses a flow layout. This is a layout in which elements are positioned line-by-line, with lines wrapping as needed. (Thus, it works much like the **FlowLayout** class used by the AWT and Swing.) In this case, a horizontal flow is used, but it is possible to specify a vertical flow. Although not needed by this skeletal application, it is also possible to specify other layout properties, such as a vertical and horizontal gap between elements, and an alignment. You will see an example of this later in this chapter.

The following line uses the root node to construct a **Scene**:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

Scene provides several versions of its constructor. The one used here creates a scene that has the specified root with the specified width and height. It is shown here:

Scene(Parent *rootnode*, double *width*, double *height*)

Notice that the type of *rootnode* is **Parent**. It is a subclass of **Node** and encapsulates nodes that can have children. Also notice that the width and the height are **double** values. This lets you pass fractional values, if needed. In the skeleton, the root is **rootNode**, the width is 300 and the height is 200.

The next line in the program sets **myScene** as the scene for **myStage**:

```
myStage.setScene(myScene);
```

Here, **setScene( )** is a method defined by **Stage** that sets the scene to that specified by its argument.

In cases in which you don't make further use of the scene, you can combine the previous two steps, as shown here:

```
myStage.setScene(new Scene(rootNode, 300, 200));
```

Because of its compactness, this form will be used by most of the subsequent examples.

The last line in **start( )** displays the stage and its scene:

```
myStage.show();
```

In essence, **show( )** shows the window that was created by the stage and screen.

When you close the application, its window is removed from the screen and the **stop( )** method is called by the JavaFX run-time system. In this case, **stop( )** simply displays a message on the console, illustrating when it is called. However, **stop( )** would not normally display anything. Furthermore, if your application does not need to handle any shutdown actions, there is no reason to override **stop( )** because an empty, default implementation is provided.

Compiling and Running a JavaFX Program

One important advantage of JavaFX is that the same program can be run in a variety of different execution environments. For example, you can run a JavaFX program as a stand-alone desktop application, inside a web browser, or as a Web Start application. However, different ancillary files may be needed in some cases, for example, an HTML file or a Java Network Launch Protocol (JNLP) file.

In general, a JavaFX program is compiled like any other Java program. However, because of the need for additional support for various execution environments, the easiest way to compile a JavaFX application is to use an Integrated Development Environment (IDE) that fully supports JavaFX programming, such as NetBeans. Just follow the instructions for the IDE you are using.

Alternatively, if you want to compile and test a JavaFX application using the command-line tools, you can easily do so. Just compile and run the application in the normal way, using **javac** and **java**. Be aware that using the command-line compiler neither creates any HTML or JNLP files that would be needed if you want to run the application in a way other than as a stand-alone application, nor does it create a JAR file for the program. To create these files, you need to use a tool such as **javafxpackager**.

The Application Thread

In the preceding discussion, it was mentioned that you cannot use the **init( )** method to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the *application thread*. However, the application's constructor and the **init( )** method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene. Instead, you must use the **start( )** method, as the skeleton demonstrates, to create the initial GUI because **start( )** is called on the application thread.

Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, in JavaFX, events are sent to your program on the application thread. Therefore, event handlers can be used to interact with the GUI. The **stop( )** method is also called on the application thread.

A Simple JavaFX Control: Label

The primary ingredient in most user interfaces is the control because a control enables the user to interact with the application. As you would expect, JavaFX supplies a rich assortment of controls. The simplest control is the label because it just displays a message, which, in this example, is text. Although quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

The JavaFX label is an instance of the **Label** class, which is packaged in **javafx.scene.control**. **Label** inherits **Labeled** and **Control**, among other classes. The **Labeled** class defines several features that are common to all labeled elements (that is, those that can contain text), and **Control** defines features related to all controls.

Label defines three constructors. The one we will use here is

Label(String *str*)

Here, *str* is the string that is displayed.

Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. To do this, you will first call **getChildren( )**

on the root node of the scene graph. It returns a list of the child nodes in the form of an **ObservableList<Node>**. **ObservableList** is packaged in **javafx.collections**, and it inherits **java.util.List**, which means that it supports all of the features available to a list as defined by the Collections Framework. Using the returned list of child nodes, you can add the label to the list by calling **add( )**, passing in a reference to the label.

The following program puts the preceding discussion into action by creating a simple JavaFX application that displays a label:

```
// Demonstrate a JavaFX label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate a JavaFX label.");

    // Use a FlowPane for the root node.
    FlowPane rootNode = new FlowPane();

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    Label myLabel = new Label("This is a JavaFX label");

    // Add the label to the scene graph.
    rootNode.getChildren().add(myLabel);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

This program produces the following window:

In the program, pay special attention to this line:

```
rootNode.getChildren().add(myLabel);
```

It adds the label to the list of children for which **rootNode** is the parent. Although this line could be separated into its individual pieces if necessary, you will often see it as shown here.

Before moving on, it is useful to point out that **ObservableList** provides a method called **addAll( )** that can be used to add two or more children to the scene graph in a single call. (You will see an example of this shortly.) To remove a control from the scene graph, call **remove( )** on the **ObservableList**. For example,

```
rootNode.getChildren().remove(myLabel);
```

removes **myLabel** from the scene.

Using Buttons and Events

Although the program in the preceding section presents a simple example of using a JavaFX control and constructing a scene graph, it does not show how to handle events. As you know, most GUI controls generate events that are handled by your program. For example, buttons, check boxes, and lists all generate events when they are used. In many ways, event handling in JavaFX is similar to event handling in Swing or the AWT, but it's more streamlined. Therefore, if you already are proficient at handling events for these other two GUIs, you will have no trouble using the event handling system provided by JavaFX.

One commonly used control is the button. This makes button events one of the most frequently handled. Therefore, a button is a good way to demonstrate the fundamentals of event handling in JavaFX. For this reason, the fundamentals of event handling and the button are introduced together.

Event Basics

The base class for JavaFX events is the **Event** class, which is packaged in **javafx.event**. **Event** inherits **java.util.EventObject**, which means that JavaFX events share the same basic functionality as other Java events. Several subclasses of **Event** are defined. The one that we will use here is **ActionEvent**. It handles action events generated by a button.

In general, JavaFX uses what is, in essence, the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return. In this regard, JavaFX events are managed much like Swing events, for example.

Events are handled by implementing the **EventHandler** interface, which is also in **javafx.event**. It is a generic interface with the following form:

```
interface EventHandler<T extends Event>
```

Here, **T** specifies the type of event that the handler will handle. It defines one method, called **handle( )**, which receives the event object as a parameter. It is shown here:

```
void handle(T eventObj)
```

Here, *eventObj* is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes for this purpose if it is more appropriate to your application (for example, if one event handler will handle events from more than one source).

Although not required by the examples in this chapter, it is sometimes useful to know the source of an event. This is especially true if you are using one handler to handle events from different sources. You can obtain the source of the event by calling **getSource( )**, which is inherited from **java.util.EventObject**. It is shown here:

```
Object getSource( )
```

Other methods in **Event** let you obtain the event type, determine if the event has been consumed, consume an event, fire an event, and obtain the target of the event. When an event is consumed, it stops the event from being passed to a parent handler.

One last point: In JavaFX, events are processed via an *event dispatch chain*. When an event is generated, it is passed to the root node of the chain. The event is then passed down the chain to the target of the event. After the target node processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event, if necessary. This is called *event bubbling*. It is possible for a node in the chain to consume an event, which prevents it from being further processed.

NOTE Although not used in this introduction to JavaFX, an application can also implement an *event filter*, which can be used to manage events. A filter is added to a node by calling **addEventFilter( )**, which is defined by **Node**. A filter can consume an event, thus preventing further processing.

Introducing the Button Control

In JavaFX, the push button control is provided by the **Button** class, which is in **javafx.scene.control**. **Button** inherits a fairly long list of base classes that include **ButtonBase**, **Labeled**, **Region**, **Control**, **Parent**, and **Node**. If you examine the API documentation for **Button**, you