CONTENTS

Chapter 15. Advanced Swing

Topics in This Chapter

- Using custom data models and renderers
- Adding and removing choices from a JList
- Displaying custom components in a JList
- Creating dynamic JTree displays
- Customizing icons in a JTree display
- Creating a scrollable JTable
- Editing cells in a JTable
- Printing Swing components
- Updating Swing components in a thread-safe manner

In this chapter, we focus on three advanced components available in Swing for building robust graphical user interfaces: `JList`, `JTree`, and `JTable`. A Swing list is similar to the familiar AWT `List` (Section 13.18, "List Boxes"); however, instead of just displaying a list of strings, a `JList` can also display icons. The `JTree` and `JTable` have no counterpart in the AWT package. A `JTree` displays data in a hierarchical structure. A common example of a tree display is the directory and file browser available in many software products. A table is equally familiar in graphical user interfaces. A `JTable` allows you to present data in a two-dimensional grid of table cells.

All three components are governed by the Model-View-Controller (MVC) architecture presented in Chapter 14 (Basic Swing), and the data for each component is stored in a data model. Multiple components may share the same data model. Once a change is made to the data through a listener (controller), an event is fired to update the display (view) of the component.

Because of their complexity, all three components provide default data models with built-in methods for modifying the data and firing change events. The default data models for a list, tree, and table all render their data as `JLabel`s. We show you how to get around this limitation, that is, how to create custom data models and cell renderers to display other objects in these components. By defining your own custom cell renderer, you can display any simple Swing component in your user control, for example, an image in a tree or a check box in a table cell.

In this chapter we also cover two advanced techniques: printing Swing components and updating a Swing component in a thread-safe manner. With the `Graphics2D` class (see Chapter 10, "Java 2D: Graphics in Java 2"), first introduced in JDK 1.2, you can properly scale components to produce high-quality graphics on printers. We show you how to open up printer dialogs to enable users to select print options and how to write a general-purpose method to print any Swing component.

Painting of components has changed significantly in Swing as compared to AWT components. Swing components have separate borders and UI delegates to display the component in the proper look and feel (LAF). See Chapter 14 (Basic Swing), for information on UI delegates and component look and feel. Changing the state of a Swing component must always be performed in the event dispatch thread. To ensure a thread-safe GUI design, we show you how to properly access a Swing component from a user-defined thread. And, we demonstrate how to update the component from within the `run` method of a `Runnable` object and how to place the object on the event queue for execution in the event dispatch thread.

The Swing components and topics covered in this chapter apply to the Java 2 Platform. If you use these advanced components in an applet, you must either provide the Swing classes in a JAR file or install the Java Plug-In on the client machine. See Section 9.9 (The Java Plug-In) for details.

15.1 Using Custom Data Models and Renderers

With almost any Swing component, you can separate the underlying data structure from the GUI control that renders and displays the data. Often you can use an array to hold the data, as is the case with lists, trees, and tables. As a convenience, Swing lets you define and use your own data structures directly in these advanced GUI controls. You simply need to implement an interface that tells the Swing component how to access and display the data. The advantage here is that you don't have to copy the data into the control, but simply tell the GUI control how to access it.

Swing has a few simple defaults for displaying values in a `JList`, `JTree`, and `JTable`. For example, in a `JList`, values that are `String`s or `Icon`s are drawn directly, whereas other `Object`s are converted to strings through their `toString` method and then displayed in a `JLabel`. In addition, Swing also lets you define arbitrary mappings (custom views) between values and display components, thus permitting display of each element in the `JList` as a custom component. To correctly display the custom list data, you must build a "cell renderer" that accepts a reference to the `JList`, the value of the entry, and a few state parameters (for example, whether or not the value is currently selected). Based on this input, the cell renderer returns the appropriate `JComponent` to display in the list.

The next section illustrates different approaches for using models and renderers to display data in a `JList`. Models and renderers for `JTree` and `JTable` are covered in later sections.

15.2 JList

A `JList` presents a list of choices for selection. Either a single item or multiple items can be selected. In the following subsections, we present four approaches for creating a `JList`. The first approach creates a fixed set of choices by simply passing data directly to the `JList`. The second approach takes advantage of the default list model to support changeable choices. The third approach demonstrates a custom list model (data structure) for holding the list items. Finally, the fourth approach uses a custom renderer to build a custom `JComponent` for each item in the list.

JList with a Fixed Set of Choices

The simplest way to create a `JList` is to supply an array of strings to the `JList` constructor. Unlike an AWT `List`, a `JList` does not support direct addition or removal of elements once the `JList` is created. To create a dynamic list, you must use a `ListModel`, as discussed later in this section. But the approach of supplying an array of strings is easier for the common case of just displaying a fixed set of choices. This approach is demonstrated as follows:

```
String[] options = { "Option 1", ... , "Option N" };
JList optionList = new JList(options);
```

You can set the number of visible rows through the `setVisibleRowCount` method. However, if the list contains more items than the number of visible rows, then this approach is not useful unless the `JList` has scrollbars. As in all of Swing, you support scrollbars by dropping the component in a

JScrollPane, as in

```
optionList.setVisibleRowCount(4);
JScrollPane optionPane = new JScrollPane(optionList);
someContainer.add(optionPane);
```

A JList generates ListSelectionEvents, so to handle selection events you attach a ListSelectionListener, which uses the valueChanged method. Be aware that a single click generates three events: one for the deselection of the originally selected entry, one for notification that the selection is moving, and one for the selection of the new entry. In the first two cases, the ListSelectionEvent's getValueIsAdjusting method returns true, so if you only care about the final selection, you would typically check that the return value is false. Of course, you can also totally ignore events and later look up which item (getSelectedValue) or index (getSelectedIndex) is currently selected. If the JList supports multiple selections, you use setSelectionMode to specify one of the ListSelectionModel constants: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, or MULTIPLE_INTERVAL_SELECTION. Then, you use getSelectedValues and getSelectedIndices to get an array of the selections. For example,

```
public class SomeClass {
  private JList optionList;
  ...
  public void someMethod() {
    ...
    MyListListener listener = new MyListListener();
    optionList.addListSelectionListener(listener);
  }
  ...
  private class MyListListener
                implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent event) {
      // Only concerned with the selection of the
      // new entry. Check to see if getValueIsAdjusting
      // returns false.
      if (!event.getValueIsAdjusting()) {
        String selection =
          optionList.getSelectedValue();
        doSomethingWith(selection);
      }
    }
  }
}
```

In the attached listener, the ListSelectionEvent is first checked to see if getValueIsAdjusting returns false. This approach effectively ignores the other two events that can occur when different items are selected in a list.

Core Approach

A single click in a list can generate three ListSelectionEvents. If you are only concerned with the selection of a list entry, then check to see that event object returns false for getValueIsAdjusting.

JList Constructors

The `JList` class defines four constructors:

public JList()

public JList(Object[] data)

public JList(Vector data)

public JList(ListModel model)

The first constructor produces an empty `JList` with no data. The second and third constructors accept an array and a `Vector`, respectively. The fourth constructor accepts a `ListModel`, which defines methods for determining the size of the list and retrieving values from the list.

Useful JList Methods

The `JList` class defines over 60 methods. The 13 most common methods are summarized below.

public void clearSelection()

This method clears the selection of any items in the list.

public ListModel getModel()

public void setModel(ListModel model)

These two methods return and set the data model containing the items to display, respectively.

public int getSelectedIndex()

public int[] getSelectedIndices()

The `getSelectedIndex` method returns the index of the first selected item in the list. The `getSelectedIndices` method returns an array containing indices of all selected items.

public Object getSelectedValue()

public Object[] getSelectedValues()

The first method returns the first selected item in the list. The second method returns an array of all items selected.

public int getSelectionMode()

public setSelectionMode(int mode)

These two methods set and return the list selection mode, respectively. Legal values are `ListSelectionModel.SINGLE_SELECTION`, `ListSelectionModel.SINGLE_INTERVAL_SELECTION`, and `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`.

public boolean getValueIsAdjusting()

A `ListSelectionEvent` can occur in three different situations: one that deselects the originally selected entry, one that indicates the selection is moving (mouse drag), and one that selects the new entry (occurs when the mouse is released). The

`getValueIsAdjusting` method returns `true` in the first two cases and `false` in the last case (when the final item is selected and the mouse is released).

public int getVisibleRowCount()

public void setVisibleRowCount(int rows)

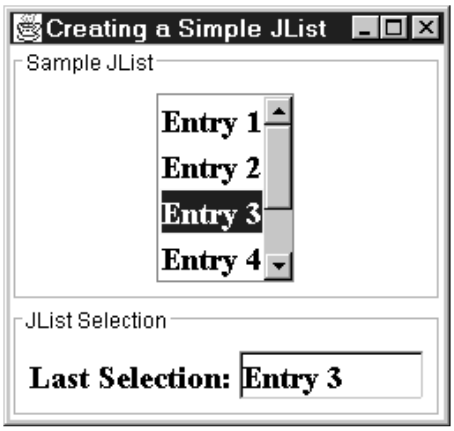
These two methods return and set the number of visible rows in the list, respectively.

public boolean isSelectedIndex(int index)

The `isSelectedIndex` method returns `true` if the item in the list at the indicated `index` is selected; otherwise, it returns `false`.

Listing 15.1 constructs a simple `JList` to hold an array of strings and places the `JList` into a scrollable pane. A listener attached to the `JList` displays the item selected by the user in a textfield, as shown in Figure 15-1. Listings for the `WindowUtilities` and `ExitListener` helper classes are given in Section 14.1 (Getting Started with Swing). This code, like all code presented in the book, is available on-line at http://www.corewebprogramming.com/.

Figure 15-1. A `JList` with a fixed set of choices.



Listing 15.1 `JListSimpleExample.java`

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

/** Simple JList example illustrating
 *  <UL>
 *    <LI>Creating a JList, which we do by passing values
 *        directly to the JList constructor, rather than
 *        using a ListModel, and
 *    <LI>Attaching a listener to determine when values change.
 *  </UL>
 */

public class JListSimpleExample extends JFrame {
  public static void main(String[] args) {
    new JListSimpleExample();
  }
```

```
  private JList sampleJList;
  private JTextField valueField;
  public JListSimpleExample() {
    super("Creating a Simple JList");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();

    // Create the JList, set the number of visible rows, add a
    // listener, and put it in a JScrollPane.
    String[] entries = { "Entry 1", "Entry 2", "Entry 3",
                         "Entry 4", "Entry 5", "Entry 6" };
    sampleJList = new JList(entries);
    sampleJList.setVisibleRowCount(4);
    sampleJList.addListSelectionListener(new ValueReporter());
    JScrollPane listPane = new JScrollPane(sampleJList);
    Font displayFont = new Font("Serif", Font.BOLD, 18);
    sampleJList.setFont(displayFont);

    JPanel listPanel = new JPanel();
    listPanel.setBackground(Color.white);
    Border listPanelBorder =
      BorderFactory.createTitledBorder("Sample JList");
    listPanel.setBorder(listPanelBorder);
    listPanel.add(listPane);
    content.add(listPanel, BorderLayout.CENTER);
    JLabel valueLabel = new JLabel("Last Selection:");
    valueLabel.setFont(displayFont);
    valueField = new JTextField("None", 7);
    valueField.setFont(displayFont);
    valueField.setEditable(false);
    JPanel valuePanel = new JPanel();
    valuePanel.setBackground(Color.white);
    Border valuePanelBorder =
      BorderFactory.createTitledBorder("JList Selection");
    valuePanel.setBorder(valuePanelBorder);
    valuePanel.add(valueLabel);
    valuePanel.add(valueField);
    content.add(valuePanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
  private class ValueReporter implements ListSelectionListener {

    /** You get three events in many cases -- one for the
     *  deselection of the originally selected entry, one
     *  indicating the selection is moving, and one for the
     *  selection of the new entry. In the first two cases,
     *  getValueIsAdjusting returns true; thus, the test below
     *  when only the third case is of interest.
     */

    public void valueChanged(ListSelectionEvent event) {
```

```
      if (!event.getValueIsAdjusting()) {
        Object value = sampleJList.getSelectedValue();
        if (value != null) {
          valueField.setText(value.toString());
        }
      }
    }
  }
}
```

JLists with Changeable Choices

To create a `JList` in which the choices are changeable, first create a `DefaultListModel` by using the default constructor. This `DefaultListModel` implements the same methods as does `java.util.Vector`, so you can manipulate the data as you would a `Vector`. Then, pass the list model to the `JList` constructor. Afterwards, any changes to the items in the `DefaultListModel` are reflected in the `JList`.

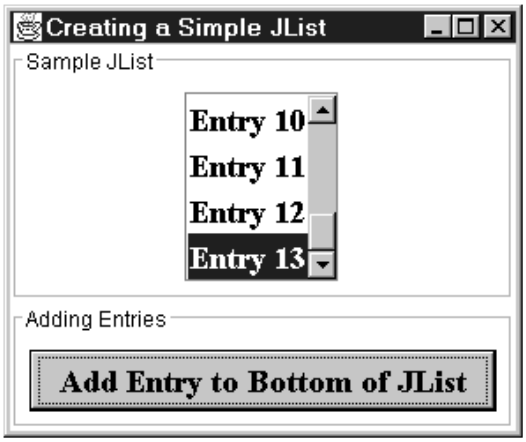
At runtime you can add entries and remove entries in a `JList` by calling the same methods on the `DefaultListModel` as you would on a `Vector`. For example,

```
String choices = { "Choice 1", ... , "Choice N"};
DefaultListModel sampleModel = new DefaultListModel();
for(int i=0; i<choices.length; i++) {
  sampleModel.addElement(choices[i]);
}
JList optionList = new JList(sampleModel);
```

Use `addElement` to insert an entry at the end of the list, and `remove(index)` to delete an entry at a specified index in the list. Note that changing entries in the list model can change the preferred size of the `JList`, which, depending on the layout manager in use, might require you to `revalidate` the window containing the `JList` to update the presentation.

In Listing 15.2, instead of a `String` array being supplied in the `JList` constructor, we first create an instance of `DefaultListModel` and add entries. Then, we supply the populated list model in the `JList` constructor. We add a `JButton` to the example with an attached `ActionListener` implemented by the class `ItemAdder`. When the user selects the button, the listener's `actionPerformed` method adds a new entry to the list model and then revalidates the panel containing the `JList`. The result is shown in Figure 15-2.

Figure 15-2. With a `JList`, choices can be added and deleted through the associated `DefaultListModel`.



Listing 15.2 DefaultListModelExample.java

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

/** JList example illustrating
 *  <UL>
 *    <LI>The creation of a JList by creating a DefaultListModel,
 *        adding the values there, then passing that to the
 *        JList constructor.
 *    <LI>Adding new values at runtime, the key thing that
 *        DefaultListModel lets you do that you can't do with
 *        a JList where you supply values directly.
 *  </UL>
 */

public class DefaultListModelExample extends JFrame {
  public static void main(String[] args) {
    new DefaultListModelExample();
  }
  JList sampleJList;
  private DefaultListModel sampleModel;

  public DefaultListModelExample() {
    super("Creating a Simple JList");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();

    String[] entries = { "Entry 1", "Entry 2", "Entry 3",
                         "Entry 4", "Entry 5", "Entry 6" };
    sampleModel = new DefaultListModel();
    for(int i=0; i<entries.length; i++) {
      sampleModel.addElement(entries[i]);
    }
    sampleJList = new JList(sampleModel);
    sampleJList.setVisibleRowCount(4);
    Font displayFont = new Font("Serif", Font.BOLD, 18);
    sampleJList.setFont(displayFont);
    JScrollPane listPane = new JScrollPane(sampleJList);

    JPanel listPanel = new JPanel();
    listPanel.setBackground(Color.white);
    Border listPanelBorder =
      BorderFactory.createTitledBorder("Sample JList");
    listPanel.setBorder(listPanelBorder);
    listPanel.add(listPane);
    content.add(listPanel, BorderLayout.CENTER);
    JButton addButton =
      new JButton("Add Entry to Bottom of JList");
```

```
    addButton.setFont(displayFont);
    addButton.addActionListener(new ItemAdder());
    JPanel buttonPanel = new JPanel();
    buttonPanel.setBackground(Color.white);
    Border buttonPanelBorder =
      BorderFactory.createTitledBorder("Adding Entries");
    buttonPanel.setBorder(buttonPanelBorder);
    buttonPanel.add(addButton);
    content.add(buttonPanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
  private class ItemAdder implements ActionListener {

    /** Add an entry to the ListModel whenever the user
     *  presses the button. Note that since the new entries
     *  may be wider than the old ones (e.g., "Entry 10" vs.
     *  "Entry 9"), you need to rerun the layout manager.
     *  You need to do this <I>before</I> trying to scroll
     *  to make the index visible.
     */

    public void actionPerformed(ActionEvent event) {
      int index = sampleModel.getSize();
      sampleModel.addElement("Entry " + (index+1));
      ((JComponent)getContentPane()).revalidate();
      sampleJList.setSelectedIndex(index);
      sampleJList.ensureIndexIsVisible(index);
    }
  }
}
```

JList with Custom Data Model

Instead of predetermining the data structure that holds the list elements, Swing lets you use your own data structure as long as you provide information on how to determine the number of elements in the data structure and how to read an element in the data structure. For use in a `JList`, any user-defined data structure must provide implementation of the four methods of the `ListModel` interface described below.

ListModel Interface

public Object getElementAt(int index)

Given an `index`, this method returns the corresponding data element.

public int getSize()

This method returns the number of entries in the list.

public void addListDataListener(ListDataListener listener)

This method adds to the model a listener that is notified when an item is selected or deselected.

public void removeListDataListener(ListDataListener listener)

This method removes the specified listener from the list model.

In Listing 15.3, we demonstrate the use of a custom list model. This example is based on a collection of `JavaLocation` objects that describe cities or regions named "Java." Rather than copying from the collection when the list requires an item, we simply define in Listing 15.4 a small helper class `JavaLocationModel`, that implements the `ListModel` interface and provides the specific methods to extract data from the collection. We then supply our custom list model to the `JList` constructor `JListCustomModel` (Listing 15.3).

The collection is stored in `JavaLocationCollection`, Listing 15.5. Listing 15.6 is the underlying data structure for the example. It contains each `JavaLocation` object, which in turn contains information on a city or region named Java, including the country in which the city is located, the country flag, and any neighboring landmark cities.

The result for this example is shown in Figure 15-3.

Figure 15-3. With a custom `ListModel`, a `JList` can display data that is stored in a custom data structure.

Listing 15.3 `JListCustomModel.java`

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

/** Simple JList example illustrating the use of a custom
 *  ListModel (JavaLocationListModel).
 */

public class JListCustomModel extends JFrame {
  public static void main(String[] args) {
    new JListCustomModel();
  }

  public JListCustomModel() {
    super("JList with a Custom Data Model");
    WindowUtilities.setNativeLookAndFeel();
```

```
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    JavaLocationCollection collection =
      new JavaLocationCollection();
    JavaLocationListModel listModel =
      new JavaLocationListModel(collection);
    JList sampleJList = new JList(listModel);
    Font displayFont = new Font("Serif", Font.BOLD, 18);
    sampleJList.setFont(displayFont);
    content.add(sampleJList);

    pack();
    setVisible(true);
  }
}
```

Listing 15.4 JavaLocationListModel.java

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

/** A simple illustration of writing your own ListModel.
 *  Note that if you wanted the user to be able to add and
 *  remove data elements at runtime, you should start with
 *  AbstractListModel and handle the event reporting part.
 */

public class JavaLocationListModel implements ListModel {
  private JavaLocationCollection collection;

  public JavaLocationListModel(JavaLocationCollection collection)
  {
    this.collection = collection;
  }

  public Object getElementAt(int index) {
    return(collection.getLocations()[index]);
  }

  public int getSize() {
    return(collection.getLocations().length);
  }

  public void addListDataListener(ListDataListener l) {}

  public void removeListDataListener(ListDataListener l) {}
}
```

Listing 15.5 JavaLocationCollection.java

```
/** A simple collection that stores multiple JavaLocation
 *  objects in an array and determines the number of
 *  unique countries represented in the data.
 */
```

```
public class JavaLocationCollection {
  private static JavaLocation[] defaultLocations =
    { new JavaLocation("Belgium",
                       "near Liege",
                       "flags/belgium.gif"),
      new JavaLocation("Brazil",
                       "near Salvador",
                       "flags/brazil.gif"),
      new JavaLocation("Colombia",
                       "near Bogota",
                       "flags/colombia.gif"),
      new JavaLocation("Indonesia",
                       "main island",
                       "flags/indonesia.gif"),
      new JavaLocation("Jamaica",
                       "near Spanish Town",
                       "flags/jamaica.gif"),
      new JavaLocation("Mozambique",
                       "near Sofala",
                       "flags/mozambique.gif"),
      new JavaLocation("Philippines",
                       "near Quezon City",
                       "flags/philippines.gif"),
      new JavaLocation("Sao Tome",
                       "near Santa Cruz",
                       "flags/saotome.gif"),
      new JavaLocation("Spain",
                       "near Viana de Bolo",
                       "flags/spain.gif"),
      new JavaLocation("Suriname",
                       "near Paramibo",
                       "flags/suriname.gif"),
      new JavaLocation("United States",
                       "near Montgomery, Alabama",
                       "flags/usa.gif"),
      new JavaLocation("United States",
                       "near Needles, California",
                       "flags/usa.gif"),
      new JavaLocation("United States",
                       "near Dallas, Texas",
                       "flags/usa.gif")
    };

  private JavaLocation[] locations;
  private int numCountries;

  public JavaLocationCollection(JavaLocation[] locations) {
    this.locations = locations;
    this.numCountries = countCountries(locations);
  }

  public JavaLocationCollection() {
    this(defaultLocations);
```

```
  }

  public JavaLocation[] getLocations() {
    return(locations);
  }

  public int getNumCountries() {
    return(numCountries);
  }

  // Count the number of unique countries in the data.
  // Assumes the list is sorted by country name
  private int countCountries(JavaLocation[] locations) {
    int n = 0;
    String currentCountry, previousCountry = "None";
    for(int i=0;i<locations.length;i++) {
      currentCountry = locations[i].getCountry();
      if (!previousCountry.equals(currentCountry)) {
        n++;
     }
      currentCountry = previousCountry;
    }
    return(n);
  }
}
```

Listing 15.6 `JavaLocation.java`

```
/** Simple data structure with three properties: country,
 *  comment, and flagFile. All are strings, and they are
 *  intended to represent a country that has a city or
 *  province named "Java," a comment about a more
 *  specific location within the country, and a path
 *  specifying an image file containing the country's flag.
 *  Used in examples illustrating custom models and cell
 *  renderers for JLists.
 */

public class JavaLocation {
  private String country, comment, flagFile;

  public JavaLocation(String country, String comment,
                      String flagFile) {
    setCountry(country);
    setComment(comment);
    setFlagFile(flagFile);
  }

  /** String representation used in printouts and in JLists */

  public String toString() {
    return("Java, " + getCountry() + " (" + getComment() + ").");
  }
```

```
  /** Return country containing city or province named "Java." */

  public String getCountry() {
    return(country);
  }

  /** Specify country containing city or province named "Java." */

  public void setCountry(String country) {
    this.country = country;
  }

  /** Return comment about city or province named "Java."
   *  Usually of the form "near such and such a city."
   */

  public String getComment() {
    return(comment);
  }
  /** Specify comment about city or province named "Java". */

  public void setComment(String comment) {
    this.comment = comment;
  }

  /** Return path to image file of country flag. */

  public String getFlagFile() {
    return(flagFile);
  }

  /** Specify path to image file of country flag. */

  public void setFlagFile(String flagFile) {
    this.flagFile = flagFile;
  }
}
```

JList with Custom Renderer

Instead of predetermining how the `JList` will draw the list elements, Swing lets you specify what graphical component to use for the various entries. Normally, Swing uses a `JLabel` to display the string representation of the list entry. If the entry is an `Icon`, that `Icon` is displayed in the `JLabel`. However, a string and an image cannot be presented in a `JList` entry at the same time unless a custom cell renderer that builds the special label (see Listing 15.8) is written. A custom cell renderer must implement the `ListCellRenderer` interface and return the desired component to display in the `JList`.

ListCellRenderer Interface

The only method in the `ListCellRenderer` interface is `getListCellRendererComponent`:

> **public Component getListCellRendererComponent(JList list, Object value, int index, boolean isSelected, boolean cellHasFocus)**

The `getListCellRendererComponent` method examines the object to render (`value`) and returns a component to render in the `JList`. In this approach, the `JList` is passed in as an argument so that the visual properties of the component match the properties of the list, for instance, the list foreground and background color. The remaining parameters in this method define the index of the cell in the list, state whether the cell is selected, and indicate whether the cell currently has focus.

As a convenience, the Java 2 Platform provides a `DefaultListCellRenderer` class that implements the `ListCellRenderer` interface. The default behavior for `DefaultListCellRenderer` is to return a `JLabel`.

In the following example we build upon the previous list example (Figure 15-3) that contains a custom data model of geographic areas named Java. We improve upon this example by adding a custom cell renderer to display both the country flag and name of each city. To build the custom renderer, in Listing 15.8 we define the class `JavaLocationRenderer` that implements the `ListCellRenderer` interface and provides a `getListCellRendererComponent` method to construct a customized `JLabel` component based on the `JavaLocation` data. This custom cell renderer is bound to the `JList` through the `setCellRenderer` method in Listing 15.7.

Listing 15.7 `JListCustomRenderer.java`

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

/** Simple JList example illustrating the use of a custom
 *  cell renderer (JavaLocationRenderer).
 */

public class JListCustomRenderer extends JFrame {
  public static void main(String[] args) {
    new JListCustomRenderer();
  }

  public JListCustomRenderer() {
    super("JList with a Custom Cell Renderer");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();

    JavaLocationCollection collection =
      new JavaLocationCollection();
    JavaLocationListModel listModel =
      new JavaLocationListModel(collection);
    JList sampleJList = new JList(listModel);
    sampleJList.setCellRenderer(new JavaLocationRenderer());
    Font displayFont = new Font("Serif", Font.BOLD, 18);
    sampleJList.setFont(displayFont);
    content.add(sampleJList);

    pack();
    setVisible(true);
  }
}
```

Rather than trying to explicitly determine how to color the component in the `JList` when the

component is selected, has focus, and so forth, it is often easier to extend `DefaultListCellRenderer` (which implements `ListCellRenderer`) and modify the returned `JLabel`, which already has the colors set appropriately. For example, in Listing 15.8 the default `JLabel` is returned from the call to `super.getListCellRenderer`, and an `Icon` is added to the `JLabel`, maintaining the existing text, foreground color, and background color. Note that `getListCellRendererComponent` is called every time the users click on a list entry, so for performance, you should cache any internal components that are generated to build the custom component. In this example, every time a list entry changes, the cell renderer needs an `ImageIcon` object to display the flag in the `JLabel`. To improve performance, a hash table that associates `ImageIcons` with `JavaLocations` is maintained. So, if the cell renderer has seen the `JavaLocation` before, we retrieve the previously created `ImageIcon` instead of generating a new `ImageIcon` object. The result for this improved `JList` with a custom cell renderer is shown in Figure 15-4.

Figure 15-4. In a `JList`, the elements are not limited to just `String` entries but can be custom Swing components.

Listing 15.8 `JavaLocationRenderer.java`

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

```
/** Simple custom cell renderer. The idea here is to augment
 *  the default renderer instead of building one from scratch.
 *  The advantage of this approach is that you don't have to
 *  handle the highlighting of the selected entries yourself,
 *  plus values that aren't of the new type you want to draw can
 *  be handled automatically. The disadvantage is that you are
 *  limited to a variation of a JLabel, which is what the default
 *  renderer returns.
 *  <P>
 *  Note that this method can get called lots and lots of times
 *  as you click on entries. We don't want to keep generating
 *  new ImageIcon objects, so we make a Hashtable that associates
 *  previously displayed values with icons, reusing icons for
 *  entries that have been displayed already.
 *  <P>
 *  Note that in the first release of JDK 1.2, the default
 *  renderer  has a bug: the renderer doesn't clear out icons for
 *  later entries. So if you mix plain strings and ImageIcons in
 *  your JList, the plain strings still get an icon. The
 *  call below clears the old icon when the value is not a
 *  JavaLocation.
 */

public class JavaLocationRenderer extends
                                  DefaultListCellRenderer {
  private Hashtable iconTable = new Hashtable();

  public Component getListCellRendererComponent(JList list,
                                                Object value,
                                                int index,
                                               boolean isSelected,
                                               boolean hasFocus) {
    // First build the label containing the text, then
    // later add the image.
    JLabel label =
      (JLabel)super.getListCellRendererComponent(list,
                                                 value,
                                                 index,
                                                 isSelected,
                                                 hasFocus);
    if (value instanceof JavaLocation) {
      JavaLocation location = (JavaLocation)value;
      ImageIcon icon = (ImageIcon)iconTable.get(value);
      if (icon == null) {
        icon = new ImageIcon(location.getFlagFile());
        iconTable.put(value, icon);
      }
      label.setIcon(icon);
    } else {
      // Clear old icon; needed in 1st release of JDK 1.2.
      label.setIcon(null);
    }
    return(label);
```

```
  }
}
```

15.3 JTree

A `JTree` can display data elements (nodes) in a hierarchical structure. In this section, we illustrate the basic use a of `JTree`, show how to respond to node selection events, provide an example of a custom tree model (a tree that builds children on the fly), and show how to replace the icons that appear at the tree nodes with custom ones.

Simple JTree

The simplest and most common way to use a `JTree` is to create objects of type `DefaultMutableTreeNode` to act as the nodes of the tree. Nodes that have no children are displayed as leaves; nodes that have children are displayed as folders. You can associate any object with a node by supplying a value, known as the "user object," to the `DefaultMutableTreeNode` constructor. Node labels are represented by `String`s; thus, before displaying the user object, you first call the `toString` method and display the resultant `String`.

Once you have created some nodes, you can hook them together in a tree structure through `parentNode.add(childNode)`. Finally, you pass the root node to the `JTree` constructor. Note that since a tree display can change size when user input so dictates (expanding and collapsing nodes), you usually place a tree inside a `JScrollPane`. For example, here is a very simple tree:

```
DefaultMutableTreeNode root =
  new DefaultMutableTreeNode("Root");
DefaultMutableTreeNode child1 =
  new DefaultMutableTreeNode("Child 1");
root.add(child1);
DefaultMutableTreeNode child2 =
  new DefaultMutableTreeNode("Child 2");
root.add(child2);
JTree tree = new JTree(root);
someWindow.add(new JScrollPane(tree));
```

For complicated trees, linking all the nodes together by hand is tedious. So, you may want to first create a simple treelike data structure, then build nodes and hook them together automatically from that data structure.

DefaultMutableTreeNode Constructors

The `DefaultMutableTreeNode` class has three constructors:

public DefaultMutableTreeNode()

public DefaultMutableTreeNode(Object data)

public DefaultMutableTreeNode(Object data,boolean allowChildren)

All of these constructors create a basic tree node. The first constructor creates a node with no data; the second constructor lets you specify the data for the node. The third constructor allows you to explicitly specify whether the node can have children; by default, a tree node can have children.

Useful DefaultMutableTreeNode Methods

The `DefaultMutableTreeNode` class defines 50 methods to modify the node's parent and

children nodes and to determine the relative location of the node in the tree hierarchy. Nine of the most common methods are listed below.

public void add(MutableTreeNode child)

public void remove(MutableTreeNode child)

These two methods add or remove a child node, respectively.

public Enumeration children()

The `children` method returns an `Enumeration` of the immediate children of the current node. If the node has no children, then `null` is returned.

public int getChildCount()

The `getChildCount` method returns the number of immediate children of the node.

public TreeNode getParent()

public TreeNode getRoot()

The `getParent` method returns the parent of the current node or `null` if the node has no parent. The `getRoot` method returns the root node of the tree containing the current node.

public boolean isLeaf()

public boolean isRoot()

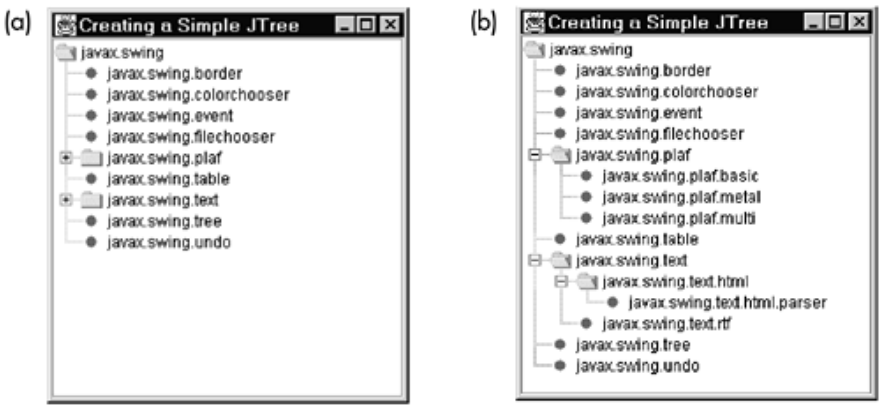
These two methods determine if the node is a leaf node or a root node.

public void removeAllChildren()

This method removes all the children from the node.

In Listing 15.9, the data for the tree is initially defined in a nested data structure represented by arrays. As the tree is built, each array element is assigned to a tree node through the `processHierarchy` method. If the array element is itself another array (`instanceof Object[]`), then `processHierarchy` is recursively called to build the subtree. In Figure 15-5(a), the initial tree is shown, and in Figure 15-5(b), the tree is shown with the folders expanded.

Figure 15-5. A `JTree` (a) before the folders are expanded and (b) after expansion of the nodes.



Listing 15.9 `SimpleTree.java`

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

/** Example tree built out of DefaultMutableTreeNodes. */

public class SimpleTree extends JFrame {
  public static void main(String[] args) {
    new SimpleTree();
  }

  public SimpleTree() {
    super("Creating a Simple JTree");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    Object[] hierarchy =
      { "javax.swing",
        "javax.swing.border",
        "javax.swing.colorchooser",
        "javax.swing.event",
        "javax.swing.filechooser",
        new Object[] { "javax.swing.plaf",
                       "javax.swing.plaf.basic",
                       "javax.swing.plaf.metal",
                       "javax.swing.plaf.multi" },
        "javax.swing.table",
        new Object[] { "javax.swing.text",
                       new Object[] { "javax.swing.text.html",
                                      "javax.swing.text.html.parser" },
                       "javax.swing.text.rtf" },
                       "javax.swing.tree",
                       "javax.swing.undo" };
    DefaultMutableTreeNode root = processHierarchy(hierarchy);
    JTree tree = new JTree(root);
    content.add(new JScrollPane(tree), BorderLayout.CENTER);
    setSize(275, 300);
    setVisible(true);
  }

  /** Small routine that will make a node out of the first entry
   *  in the array, then make nodes out of subsequent entries
   *  and make them child nodes of the first one. The process
   *  is repeated recursively for entries that are arrays.
   */

  private DefaultMutableTreeNode processHierarchy(
                                          Object[] hierarchy) {
    DefaultMutableTreeNode node =
      new DefaultMutableTreeNode(hierarchy[0]);
    DefaultMutableTreeNode child;
    for(int i=1; i<hierarchy.length; i++) {
      Object nodeSpecifier = hierarchy[i];
      if (nodeSpecifier instanceof Object[]) { //Node with children
```

```
        child = processHierarchy((Object[])nodeSpecifier);
      } else {
        child = new DefaultMutableTreeNode(nodeSpecifier); //Leaf
      }
      node.add(child);
    }
    return(node);
  }
}
```

Before showing you how to dynamically modify the tree through `JTree` events, we first summarize the `JTree` constructors and methods.

JTree Constructors

The `JTree` class provides seven different constructors. The two most common constructors are listed below. The less common constructors allow you to supply the tree information in a `Vector` or `Hashtable`. Note that `JTree` also has a no-argument constructor, but in this case, the tree is automatically populated with a sample data set defined by Sun—useful only for initial testing.

public JTree(TreeNode root)

public JTree(TreeModel model)

The first constructor creates a tree rooted at the specified node. The second constructor creates a tree according to the data in the specified tree model.

Useful JTree Methods

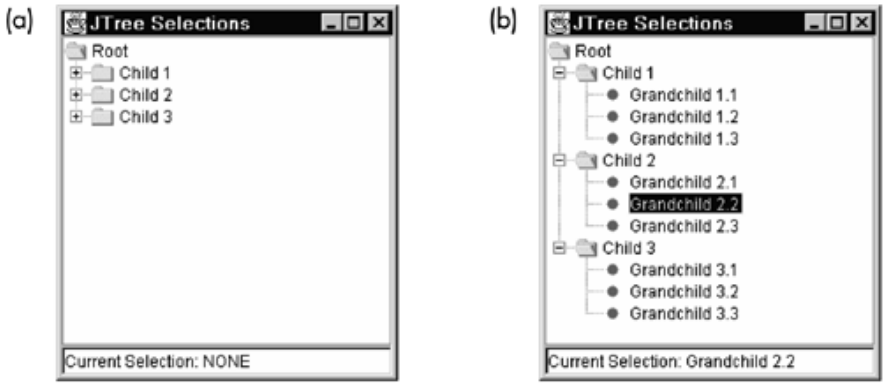
The `JTree` class is quite robust, defining over 110 methods. Many of these methods are for editing and working with selected branches of the tree, along with methods for firing `TreeExpansionEvent`s. Instead of covering this huge list of methods, we refer you to the `javax.swing.JTree` API.

JTree Event Handling

To handle selection events, attach a `TreeSelectionListener` to the `JTree`. The `TreeSelectionListener` interface requires implementation of a single method: `valueChanged`. Once a `TreeSelectionEvent` occurs, you can determine the selected node through `tree.getLastSelectedPathComponent` and then cast the returned object to your node type (usually `DefaultMutableTreeNode`). The actual data contained in the node is extracted through `getUserObject`. However, if all you want is the node label, you can just call `toString` on the result of `tree.getLastSelectedPathComponent`.

In Listing 15.10 a simple tree is created where each child node has three children. A `TreeSelectionListener` is attached to the tree. When the user selects a node, the corresponding node's `toString` value is displayed in a text field below the tree, as shown in Figure 15-6(a) and 15-6(b).

Figure 15-6. A selectable `JTree` (a) in an initial state and (b) after selection of a child node.



Listing 15.10 `SelectableTree.java`

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

/** JTree that reports selections by placing their string values
 *  in a JTextField.
 */

public class SelectableTree extends JFrame
                            implements TreeSelectionListener {
  public static void main(String[] args) {
    new SelectableTree();
  }

  private JTree tree;
  private JTextField currentSelectionField;

  public SelectableTree() {
    super("JTree Selections");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    DefaultMutableTreeNode root =
      new DefaultMutableTreeNode("Root");
    DefaultMutableTreeNode child;
    DefaultMutableTreeNode grandChild;
    for(int childIndex=1; childIndex<4; childIndex++) {
      child = new DefaultMutableTreeNode("Child " + childIndex);
      root.add(child);
      for(int grandChildIndex=1; grandChildIndex<4;
           grandChildIndex++) {
        grandChild =
          new DefaultMutableTreeNode("Grandchild " + childIndex +
                                     "." + grandChildIndex);
        child.add(grandChild);
      }
    }
    tree = new JTree(root);
    tree.addTreeSelectionListener(this);
```

```
    content.add(new JScrollPane(tree), BorderLayout.CENTER);
    currentSelectionField =
      new JTextField("Current Selection: NONE");
    content.add(currentSelectionField, BorderLayout.SOUTH);
    setSize(250, 275);
    setVisible(true);
  }

  public void valueChanged(TreeSelectionEvent event) {
    Object selection = tree.getLastSelectedPathComponent();
    if (selection != null) {
      currentSelectionField.setText
        ("Current Selection: " + selection.toString());
    }
  }
}
```

A `TreeExpansionEvent` occurs whenever a node in a tree is expanded or collapsed. You can handle `TreeExpansionEvent`s by attaching a `TreeExpansionListener` to the `JTree`. The expansion listener defines two methods, `treeExpanded` and `treeCollapsed`, that are called when a node is expanded or collapsed, respectively. The `TreeExpansionEvent` has a single method, `getPath`, which returns a `Path` from the root node to the source node producing the event. Use `getLastPathComponent` to obtain the source node from the `Path`.

Custom Models and Dynamic Trees

A `JTree` uses a `TreeModel` to obtain the data from the underlying data structure. As with a `JList`, you can replace the model altogether, specifying how to extract data from a custom data structure. See Section 15.2 (JList) for an example of this general approach.

Technically, the `TreeModel` just keeps track of the root node in the tree and defines support methods for accessing node children and handling listeners. The nodes in a tree are interconnected in a linked manner. Thus, the common approach to create a custom tree is to leave the class implementing the `TreeModel` interface unchanged and, instead, create a custom `TreeNode`. The easiest approach is to inherit from `DefaultMutableTreeNode`, which provides the basic structure for a parent node and multiple children nodes.

Next, we define some of the useful `DefaultMutableTreeNode` methods and then show an example of how you might inherit from `DefaultMutableTreeNode` to create a custom tree node.

Useful DefaultMutableTreeNode Methods

The `DefaultMutableTreeNode` class defines over 50 methods for accessing related nodes in a general tree. References to the child nodes are stored in a `Vector` internal to the class. We summarize the more common methods below.

public void add(MutableTreeNode child)

public void remove(MutableTreeNode child)

These two methods add and remove a child node, respectively. The child is added to the end of the internal vector structure referring to the children.

public void insert(MutableTreeNode child, int index)

public void remove(int index)

The first method inserts a new child at the specified vector index, shifting up by one position all siblings in the vector with a higher or equal index. The greatest allowable index value is determined by the size of the internal vector; use `getChildCount` to determine the vector size and corresponding maximum index value. The second method removes the child at the specified index. If the index is outside the range of the vector, an `ArrayIndexOutOfBoundsException` is thrown.

public Enumeration children()

This method returns an `Enumeration` of the children of the current node.

public TreeNode getChildAt(int index)

public int getIndex(TreeNode child)

The `getChildAt` method returns the child node defined at the given index. An `ArrayIndexOutOfBoundsException` is thrown if the index value is outside the range of the internal vector storing the references to the children. Similarly, `getIndex` returns either the corresponding index of the child or `-1` if the child is not a child of the current node.

public int getChildCount()

The `getChildCount` method returns the number of children of the node.

public TreeNode getParent()

This method returns the parent of the current node or `null` if the node has no parent.

public TreeNode[] getPath()

The `getPath` method returns an array of tree nodes representing the path from the root node to the current node.

public TreeNode getRoot()

This method returns the root node of the tree.

public boolean isLeaf()

public boolean isRoot()

The first method, `isLeaf`, returns `true` if the node is a leaf (no children); otherwise, it returns `false`. Similarly, the second method, `isRoot`, returns `true` if the node is the root of the tree; otherwise it returns `false`.

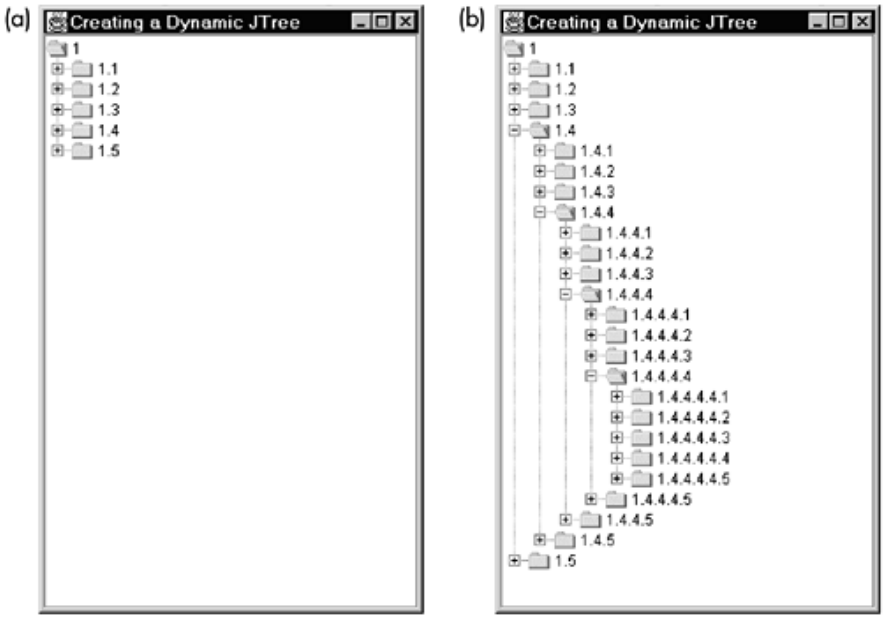
You might want to build a tree dynamically as user input is received, for example, to expand a folder—a common case. So, you would want to inherit from `DefaultMutableTreeNode` to create a custom node so that you don't have to explicitly lay out each node in the tree beforehand. In that case, you would provide an algorithm that describes the children of a given node but actually generates children only for those instances when the user expands the folder.

In Listing 15.11, the tree represents a "numbered outline," where each node describes a separate part in the outline. The root is 1, the first-level children are 1.1, 1.2, 1.3, etc., the second-level children are 1.1.1, 1.1.2, etc., and so forth. The children are built dynamically, and the actual number of children of each node is determined by a command-line argument to the program.

The key to building a `JTree` dynamically is to ensure that `getChildCount` is called before any of the children are actually retrieved. Thus, you must define a flag indicating whether the children were

built earlier. The approach is to simply wait until `getChildCount` is called, and then, if the flag is `false`, build the children and add them to the node. To keep the tree from trying to count the children (and subsequently build their nodes) to determine which nodes are leaf nodes, override `isLeaf` to always return `false`. This approach is shown in `OutlineNode`, Listing 15.12, which builds the children nodes dynamically. The result is shown in Figure 15-7.

Figure 15-7. A custom `TreeNode` permits dynamic growth of the tree: (a) an initial tree, (b) nodes dynamically generated after user expands a node.



Listing 15.11 `DynamicTree.java`

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

/** Example tree that builds child nodes on the fly.
 *  See OutlineNode for details.
 */

public class DynamicTree extends JFrame {
  public static void main(String[] args) {
    int n = 5; // Number of children to give each node.
    if (args.length > 0) {
      try {
        n = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {
        System.out.println(
          "Can't parse number; using default of " + n);
      }
   }
    new DynamicTree(n);
  }

  public DynamicTree(int n) {
    super("Creating a Dynamic JTree");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
```

```
    Container content = getContentPane();
    JTree tree = new JTree(new OutlineNode(1, n));
    content.add(new JScrollPane(tree), BorderLayout.CENTER);
    setSize(300, 475);
    setVisible(true);
  }
}
```

Listing 15.12 `OutlineNode.java`

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

/** Simple TreeNode that builds children on the fly.
 *  The key idea is that getChildCount is always called before
 *  any actual children are requested. That way, getChildCount
 *  builds the children if they don't already exist.
 *  <P>
 *  In this case, it just builds an "outline" tree. I.e.,
 *  if the root is current node is "x", the children are
 *  "x.0", "x.1", "x.2", and "x.3".
 * <P>
 */

public class OutlineNode extends DefaultMutableTreeNode {
  private boolean areChildrenDefined = false;
  private int outlineNum;
  private int numChildren;

  public OutlineNode(int outlineNum, int numChildren) {
    this.outlineNum = outlineNum;
    this.numChildren = numChildren;
  }

  public boolean isLeaf() {
    return(false);
  }

  public int getChildCount() {
    if (!areChildrenDefined) {
      defineChildNodes();
    }
    return(super.getChildCount());
  }

  private void defineChildNodes() {
    // You must set the flag before defining children if you
    // use "add" for the new children. Otherwise, you get an
    // infinite recursive loop since add results in a call
    // to getChildCount. However, you could use "insert" in such
    // a case.
    areChildrenDefined = true;
    for(int i=0; i<numChildren; i++) {
```

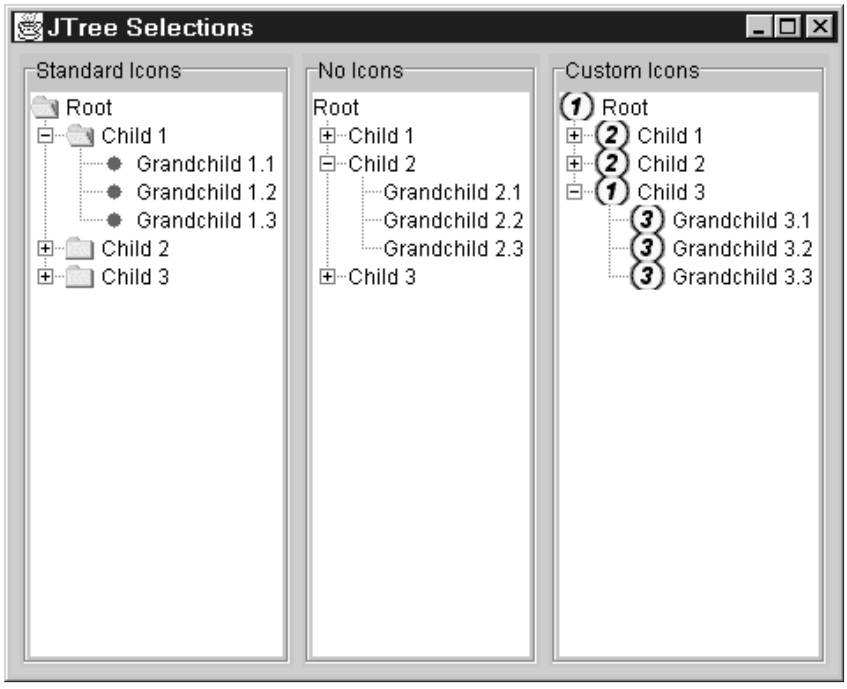
```
      add(new OutlineNode(i+1, numChildren));
    }
  }

  public String toString() {
    TreeNode parent = getParent();
    if (parent == null) {
      return(String.valueOf(outlineNum));
    } else {
      return(parent.toString() + "." + outlineNum);
    }
  }
}
```

Icon Replacement at the Tree Nodes

A common enhancement to a default `JTree` is to simply change the three icons displayed for an unexpanded internal node (i.e., nonleaf), an expanded internal node, and a leaf node. To replace the `ImageIcon`s displayed by the `JTree`, simply create an instance of `DefaultTreeCellRenderer` and then call `setOpenIcon`, `setClosedIcon`, and `setLeafIcon` either with the `Icon` of interest (usually an `ImageIcon` made from a small image file) or `null` to just turn off the node icons. Then, associate this cell renderer with the tree through `setCellRenderer`.

In Listing 15.13, three separate trees are created: in the first tree, the default node icons are displayed; in the second tree, the node icons are set to `null`; and not displayed; and in the third tree, the nodes are represented by custom icons. The three trees are shown in Figure 15-8.

Figure 15-8. An example of customizing the node icons in a `JTree`.



Listing 15.13 `CustomIcons.java`

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

```
import javax.swing.*;
import javax.swing.tree.*;

/** JTree with missing or custom icons at the tree nodes. */

public class CustomIcons extends JFrame {
  public static void main(String[] args) {
    new CustomIcons();
  }

  private Icon customOpenIcon =
            new ImageIcon("images/Circle_1.gif");
  private Icon customClosedIcon =
            new ImageIcon("images/Circle_2.gif");
  private Icon customLeafIcon =
            new ImageIcon("images/Circle_3.gif");

  public CustomIcons() {
    super("JTree Selections");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setLayout(new FlowLayout());
    DefaultMutableTreeNode root =
      new DefaultMutableTreeNode("Root");
    DefaultMutableTreeNode child;
    DefaultMutableTreeNode grandChild;
    for(int childIndex=1; childIndex<4; childIndex++) {
      child = new DefaultMutableTreeNode("Child " + childIndex);
      root.add(child);
      for(int grandChildIndex=1; grandChildIndex<4;
                                 grandChildIndex++) {
        grandChild =
          new DefaultMutableTreeNode("Grandchild " +
                                     childIndex +
                                     "." + grandChildIndex);
        child.add(grandChild);
      }
    }
    JTree tree1 = new JTree(root);
    tree1.expandRow(1); // Expand children to illustrate leaf icons.
    JScrollPane pane1 = new JScrollPane(tree1);
    pane1.setBorder(
            BorderFactory.createTitledBorder("Standard Icons"));
    content.add(pane1);

    JTree tree2 = new JTree(root);
    // Expand children to illustrate leaf icons.
    tree2.expandRow(2);
    DefaultTreeCellRenderer renderer2 =
                               new DefaultTreeCellRenderer();
    renderer2.setOpenIcon(null);
    renderer2.setClosedIcon(null);
    renderer2.setLeafIcon(null);
```

```
    tree2.setCellRenderer(renderer2);
    JScrollPane pane2 = new JScrollPane(tree2);
    pane2.setBorder(
            BorderFactory.createTitledBorder("No Icons"));
    content.add(pane2);

    JTree tree3 = new JTree(root);
    // Expand children to illustrate leaf icons.
    tree3.expandRow(3);
    DefaultTreeCellRenderer renderer3 =
                              new DefaultTreeCellRenderer();
    renderer3.setOpenIcon(customOpenIcon);
    renderer3.setClosedIcon(customClosedIcon);
    renderer3.setLeafIcon(customLeafIcon);
    tree3.setCellRenderer(renderer3);
    JScrollPane pane3 = new JScrollPane(tree3);
    pane3.setBorder(
           BorderFactory.createTitledBorder("Custom Icons"));
    content.add(pane3);

    pack();
    setVisible(true);
  }
}
```

15.4 JTable

A `JTable` provides a way to display rows and columns of data in a two-dimensional display. It is by far the most complex component available in the Swing API. A `JTable` is actually based upon three separate models: a table model, a column model, and a list selection model. The table model maintains the data in the table cells, the column model controls addition and removal of columns in the table, and the list selection model is responsible for the selection of rows in the table (selection of columns is controlled by the column model).

Because of the complexity of a `JTable`, each of the three table models has a default implementation in the `javax.swing.table` package. In this section, we examine the default and custom table models. For detailed coverage of column models and selection models, see *Core Java Foundation Classes* by Kim Topley.

Simple JTable

The simplest way to create a `JTable` is to supply the constructor with a two-dimensional array of strings for the table cells and a one-dimensional array of strings for the column names. For example, the following code creates a simple table with *N* columns and *M* rows,

```
String[][] data = { { "Cell (1,1)", ..., "Cell (1,N)" },
                    ...
                    { "Cell (M,1)", ..., "Cell (M,N)" } };
String[] columnNames = { "Column 1", ..., "Column N" };
JTable table = new JTable(data, columnNames);
```

Supplying an `Object` array for the column names and cell data is also legal because a `JTable` renders each object as a `JLabel` based upon the object's `toString` method.

Like a `JList` and a `JTree`, a table is most often placed inside a scroll pane. Without a scroll pane,

the column labels are not displayed. The preferred size of the scroll pane is determined by the viewport size of the table, which, by default, is 450 x 400 pixels. You can change the default size of the viewport by providing a `Dimension` object to the table's `setPreferredScrollableViewportSize` method.

Core Note

The column labels for a `JTable` are not displayed unless the table is placed in a `JScrollPane`.

JTable Constructors

The `JTable` class defines seven constructors. In general, if data for the tables cells is not provided, then the cells are populated with `null` values. If not specified, column labels follow standard spreadsheet conventions (A, B, C, …, AA, BB, CC, …). We outline the five most common `JTable` constructors below. In all but the last constructor, the table is implemented with a `DefaultTableModel`, `DefaultColumnModel`, and `DefaultListSelectionModel`.

public JTable()

This constructor creates an empty table with an internal `DefaultTableModel`, `DefaultColumnModel`, and `DefaultListSelectionModel`. Use `addRow` and `addColumn` to populate the table. By default, every cell in the table is editable.

public JTable(int rows, int columns)

This constructor is identical to the default class constructor except that the `DefaultTableModel` contains the specified number of rows and columns (each cell value is `null`).

public JTable(Object[][] data, Object[] columnNames)

This constructor lets you specify the table data and column names. Usually, the data and column names are strings and are rendered as `JLabel`s when displayed. The use of `toString` is the default rendering for the table data; if a data type does not have a defined renderer to display the information, the `toString` method is called and the result is displayed in a `JLabel`. Note that you can take advantage of other cell renderers to display `Boolean`, `ImageIcon`, and `Number` objects in the table cells as described later in this section.

public JTable(Vector data, Vector columnNames)

This constructor allows you to define the table data and column names in `Vectors`. Each element of the data vector should itself be a `Vector` of `Objects` representing a single row in the table.

public JTable(TableModel model)

This constructor creates a `JTable` with cells and columns defined by the `TableModel` argument. Most often, the table model is based either on `AbstractTableModel` or `DefaultTableModel`, both of which fire `TableModelEvents`. The table also contains a `DefaultColumnModel` and `DefaultListSelectionModel`.

Useful JTable Methods

`JTable` defines over 125 methods, but many of these methods are mirrored by one of the three

models from which a `JTable` is built: the `TableModel`, `TableColumnModel`, and `ListSelectionModel`. In fact, a reference to each of these three models is defined as a `protected` field member in the `JTable` class, and many of the class methods simply call the matching method in the corresponding model. We describe only a few of the methods below.

public TableModel getModel()

public void setModel(TableModel tableModel)

These two methods return and set the `TableModel`, respectively. The `TableModel` contains the data that is displayed in the `JTable` cells.

public TableColumnModel getColumnModel()

public setColumnModel(TableColumnModel columnModel)

These two methods retrieve or set the `TableColumnModel` for the table, respectively. By default, the table column model is automatically created. If your table design requires you to programmatically move your columns, you should write own custom column model. The `TableColumnModel` also controls selection of a single or multiple columns. For details, see the `javax.swing.table` API.

public ListSelectionModel getSelectionModel()

public void setSelectionModel(ListSelectionModel selectionModel)

The `getListSelectionModel` method returns the table selection model, and the `setListSelectionModel` method sets the table selection model. The `ListSelectionModel` is used by both `JTable` and `JList` (see Section 15.2 for `JList`) and supports single selection, single-interval selection, and multiple-interval selection of cells. Whenever the user changes the selected rows, a `ListSelectionEvent` is placed on the event queue and later dispatched to the table's `valueChanged` method. Use `getSelectedRow` or `getSelectedRows` to determine which rows the user selected. Also, note that a `TableColumnModel` can support a `ListSelectionModel` for selecting columns. See the `javax.swing` API for additional information on the `ListSelectionModel`.

public int getRowHeight()

public void setRowHeight(int height)

public int getRowHeight(int row)

public void setRowHeight(int row, int height)

The first two methods return or set the row height in pixels. The default row height is 16 pixels. In JDK 1.3, methods were added to return or set the row height for an individual row.

public int getRowMargin()

public void setRowMargin(int margin)

The `getRowMargin` and `setRowMargin` methods return or set the pixel width (margin) between table rows. The default margin is one pixel. The column margin is specified in the `TableColumnModel` by `setColumnMargin`.

public void setShowGrid(boolean show)

public boolean getShowHorizontalLines()

public void setShowHorizontalLines(boolean show)

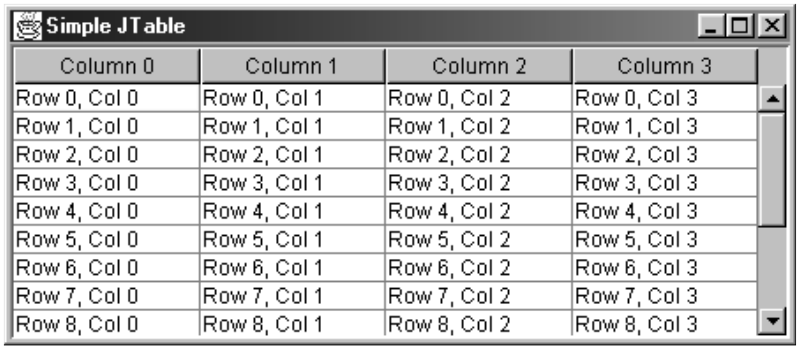
public boolean getShowVerticalLines()

public void setShowVerticalLines(boolean show)

The `setShowGrid` method turns off or on the display of the horizontal and vertical grid lines in the table. The color of the lines is determined by the installed UI delegate. The remaining four methods allow you to set and get the display status for the horizontal and vertical lines in the table.

Listing 15.14 creates a simple `JTable` with 4 columns and 15 rows. The height of each row is determined by the default font size of the text, and the width of each column is determined by equal division of viewport width among the four columns. The result for this simple `JTable` is shown in Figure 15-9.

Figure 15-9. A simple `JTable` placed in a scrollable viewport.



Listing 15.14 `JTableSimpleExample.java`

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

/** Simple JTable example that uses a String array for the
 *  table header and table data.
 */

public class JTableSimpleExample extends JFrame {
  public static void main(String[] args) {
    new JTableSimpleExample();
  }

  private final int COLUMNS = 4;
  private final int ROWS = 15;
  private JTable sampleJTable;

  public JTableSimpleExample() {
    super("Creating a Simple JTable");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
```

```
    String[]   columnNames = buildColumnNames(COLUMNS);
    String[][] tableCells = buildTableCells(ROWS, COLUMNS);

    sampleJTable = new JTable(tableCells, columnNames);
    JScrollPane tablePane = new JScrollPane(sampleJTable);
    content.add(tablePane, BorderLayout.CENTER);
    setSize(450,150);
    setVisible(true);
  }

  private String[] buildColumnNames(int columns) {
    String[] header = new String[columns];
    for(int i=0; i<columns; i++) {
      header[i] = "Column " + i;
    }
    return(header);
  }
  private String[][] buildTableCells(int rows, int columns) {
    String[][] cells = new String[rows][columns];
    for(int i=0; i<rows ; i++) {
      for(int j=0; j<columns; j++ ) {
        cells[i][j] = "Row " + i + ", Col " + j;
      }
    }
    return(cells);
  }
}
```

Table Data Models

When a table is displayed, the data is mapped from the data model (data structure) to the cells in the `JTable`. To display the data correctly, the data model must implement the `TableModel` interface. The interface defines methods for determining the number of row and columns in the model, for changing data values in the model, and for adding or removing event listeners to the model. In practice, most programmers don't implement the `TableModel` interface directly but start with `AbstractTableModel`, which inherits from `TableModel` and adds management of event listeners and methods for firing `TableModelEvent`s. To use `AbstractTableModel` you must provide concrete methods for

- `public int getRowCount()`
- `public int getColumnCount()`
- `public Object getValueAt(int row, int column)`

The underlying structure of your data model will determine your implementation of these three methods.

The `AbstractTableModel` class also provides a concrete implementation of the `setValueAt` method to support noneditable data structures. Because `setValueAt` does not modify the data, the `isCellEditable` method always returns `false`.

As a convenience, the Swing API also includes a `DefaultTableModel` that directly inherits from `AbstractTableModel` and provides implementation of all the abstract methods in the superclass. `DefaultTableModel` is a vector-based data structure; the model is a `Vector` of rows, where each row is a `Vector` of data cells. To accommodate the internal data structure, arrays or vectors

passed into the `DefaultTableModel` constructor are reformatted into the underlying vector model. For very large tables you may observe a performance hit when the `DefaultTableModel` object is first initialized with the data. The `DefaultTableModel` always returns `true` for `isCellEditable`; therefore, by default the user is able to select any cell in the table and modify the contents.

Core Note

In the `AbstractTableModel` class, `isCellEditable` always returns `false`; whereas in `DefaultTableModel`, the method always returns `true`.

DefaultTableModel Constructors

The `DefaultTableModel` class defines six constructors. The three most common constructors are described below.

public DefaultTableModel(Object[][] data,Object[] columnNames)

This constructor creates a `DefaultTableModel` where the tables cells are defined by the two-dimensional array `data` and the names for the columns are provided by the one-dimensional array `columnNames`. Often, the cell data and column names are represented by `String`s but may be other objects (images, check boxes) if the proper cell renderers are installed. We discuss "Table Cell Renderers" later in this section.

public DefaultTableModel(Vector data, Vector columnNames)

This constructor accepts the table cell data in a `Vector`, where each element in the vector represents a row within the table and each row is represented by a vector of cells. The column names for the table are defined in the `columnNames` vector.

public DefaultTableModel(Vector columnNames, int numRows)

This constructor create a table model with the given column names and number of rows. Each row in the table is a vector equal in size to the number of columns. The data for the rows is initially set to `null`.

Useful DefaultTableModel Methods

`DefaultTableModel` supports methods to add or remove rows and columns to or from the data model, as well as to change cell values. The most common methods are summarized below:

public void addColumn(Object columnName)

public void addColumn(Object columnName,Object[] columnData)

public void addColumn(Object columnName,Vector columnData)

The first method adds a column with the given name to the table model. The corresponding row cells are filled with `null` values. The second and third methods also add a column to the table model, but you can also specify the data for the column either as an array or vector.

public void addRow(Object[] rowData)

public void addRow(Vector rowData)

public void insertRow(int row, Object[] rowData)

public void insertRow(int row, Vector rowData)

The first two `addRow` methods add a row to the data model, specified either as an array or vector. The row is appended *after* the last row in the data model. The third and fourth methods specify the row in the data model in which to insert the data. If `rowData` is `null`, the row is populated with `null` values. If you specify a row outside the range of the internal table model vector, an `ArrayIndexOutOfBoundsException` is thrown.

public void removeRow(int row)

The `removeRow` method removes the specified `row` from the table model. If the row value is less than zero or greater than the number of rows in the model, an `ArrayIndexOutOfBoundsException` is thrown.

public int getColumnCount()

public int getRowCount()

These two methods return the number of columns and rows in the table model, respectively.

public String getColumnName(int column)

This method returns the `toString` description of the specified `column`. If the column name is `null`, then a letter (following spreadsheet naming conventions, A, B, C, …, AA, BB, …) is returned. If the column value is less than zero or greater than or equal to the number of columns in the model, then an `ArrayIndexOutOfBoundsException` is thrown.

public Object getValueAt(int row, int column)

public setValueAt(Object value, int row, int column)

These two methods return and set the cell value at the specified `row` and `column`, respectively. After changing the cell value, the `setValueAt` method fires a `TableModelEvent` to update the display. An `ArrayIndexOutOfBoundsException` is thrown if the `row` or `column` is outside the valid range of the table model.

public boolean isCellEditable(int row, int column)

For the `DefaultTableModel`, the `isCellEditable` method returns `true` regardless of the row and column value; every cell in a `DefaultTableModel` is editable.

Listing 15.15 provides an example of using the `DefaultTableModel`. The table contents are derived from the `JavaLocationCollection` presented earlier in Listing 15.5. Each `JavaLocation` object defines the following information for each region named Java: the country in which the region is located, the country flag (GIF file), and a comment describing any neighboring landmark cities. In this example, the `DefaultTableModel` is initially empty; later, the columns are added as needed to the model by a call to `addColumn`. After the columns are added to the table model, each row is subsequently added to the table model by a call to `getRowData` to build a vector from the `JavaLocation` information, followed by a call to `addRow` to add the new row to the table. In addition to the city and country, the table includes a column for the country flag (`ImageIcon`) and a column containing a boolean field to indicate whether the user has visited the city.

The result of Listing 15.15 is shown in Figure 15-10. Notice that each cell contains the `String` representation of the corresponding object. In the next section we explain how to take advantage of built-in cell renderers to display the `ImageIcon` as an image and `Boolean` as a check box in the table.

Figure 15-10. A table built with the `DefaultTableModel`.

Listing 15.15 `DefaultTableExample.java`

```
import java.util.Vector;
import javax.swing.*;
import javax.swing.table.*;

/** JTable that uses the DefaultTableModel, which permits
 *  adding rows and columns programmatically.
 */

public class DefaultTableExample extends JTable {

  private String[] columnNames =
    { "Flag", "City", "Country", "Comment", "Visited" };

  public DefaultTableExample() {
    this(new DefaultTableModel());
}
  public DefaultTableExample(DefaultTableModel model) {
    super(model);

    JavaLocationCollection collection =
      new JavaLocationCollection();
    JavaLocation[] locations = collection.getLocations();

    // Set up the column labels and data for the table model.
    int i;
    for(i=0; i<columnNames.length; i++ ) {
      model.addColumn(columnNames[i]);
    }
    for(i=0; i<locations.length; i++) {
      model.addRow(getRowData(locations[i]));
    }
  }

  private Vector getRowData(JavaLocation location) {
    Vector vector = new Vector();
    vector.add(new ImageIcon(location.getFlagFile()));
    vector.add("Java");
    vector.add(location.getCountry());
    vector.add(location.getComment());
    vector.add(new Boolean(false));
    return(vector);
  }
```

```
  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(
       new JScrollPane(new DefaultTableExample()), 600, 150,
                       "Using a DefaultTableModel");
  }
}
```

Table Cell Renderers

Typically, the table renders an object in a table by using a `JLabel`. However, for certain classes of objects, the Java 2 Platform provides additional cell renderers to display the object in a more appropriate format. Specifically, default table cell renderers are already defined for the following class types:

- `Boolean`— Displayed by a `JCheckBox`.
- `Date`— Displayed by a `JLabel` after the date is formatted with the `DateFormat` class.
- `ImageIcon`— Displayed as an image by a `JLabel`.
- `Number`— Displayed by a `JLabel` after the number is formatted with the `NumberFormat` class.
- `Object`— Displayed by a `JLabel` after a call to the `toString` method.

To take advantage of the default cell renderers, you must override the `getColumnClass` method in `DefaultTableModel` to provide explicit class information about the type of objects contained in each column. By default, `getColumnClass` returns `Object.class`, which always results in the object being rendered as a `JLabel`.

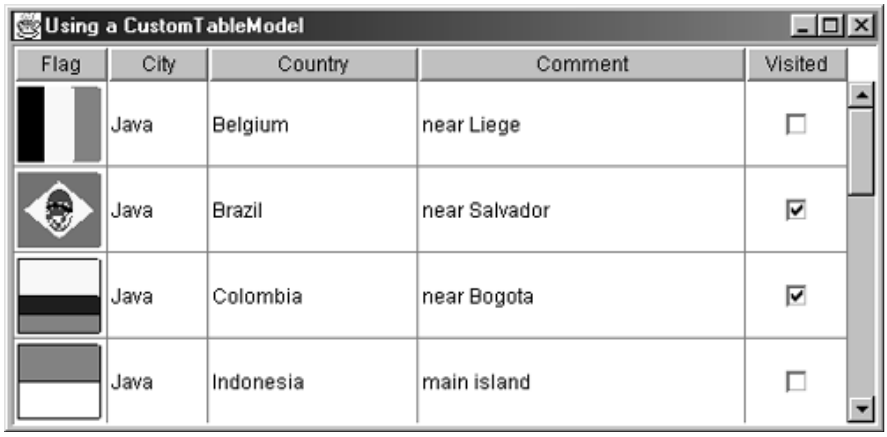
The standard approach for enabling the default cell renderers is to create a custom table model that inherits from `DefaultTableModel` and then to override `getColumnClass` to return the true underlying class of the objects. For example,

```
public Class getColumnClass(int column) {
  return (getValueAt(0, column).getClass());
}
```

This method returns the class of object in the first cell of the specified column. Here, the assumption is that all objects in the column are of the same class, or at least, the objects have a common superclass that `getColumnClass` can return. In addition, a renderer for this common superclass needs to be installed in the table.

In Listing 15.16, `CustomTableExample` inherits from the previous example, `DefaultTableExample`, and defines a new constructor to instantiate an instance of `CustomTableModel` (Listing 15.17) as the data model for the table. Here, `CustomTableModel` inherits from `DefaultTableModel` and overrides `getColumnClass` to take advantage of the default cell renderers. In addition, `CustomTableModel` overrides `isCellEditable` to restrict the user to modification of data only in the Comment and Visited columns. The result is shown in Figure 15-11.

Figure 15-11. A table with cell renderers to display images and check boxes. The column widths are explicitly set to override default values.



To improve the table layout, explicitly set the pixel width of each column by using `setMinWidth`, `setMaxWidth`, or `setPreferredWidth`. By default, the preferred width of a column is 75 pixels and the minimum width is 15 pixels. To set the column width, use

```
TableColumn column = table.getColumn(columnName);
column.setPreferredWidth(numPixels);
```

Because of a Swing bug in some JDK releases, after setting the column sizes, you must call

```
table.sizeColumnsToFit(JTable.AUTO_RESIZE_OFF);
```

to lay out the columns correctly to the viewport. `TableColumn` also defines a `setWidth` method; however, the column width does not persist when the table is subsequently resized.

Listing 15.16 `CustomTableExample.java`

```
import javax.swing.*;
import javax.swing.table.*;

/** JTable that uses a CustomTableModel to correctly render
 *  the table cells that contain images and boolean values.
 */

public class CustomTableExample extends DefaultTableExample {

  public CustomTableExample() {
    super(new CustomTableModel());
    setCellSizes();
  }

  private void setCellSizes() {
    setRowHeight(50);
    getColumn("Flag").setMaxWidth(55);
    getColumn("City").setPreferredWidth(60);
    getColumn("Country").setMinWidth(80);
    getColumn("Comment").setMinWidth(150);
    // Call to resize columns in viewport (bug).
    sizeColumnsToFit(JTable.AUTO_RESIZE_OFF);
  }

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

```
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(
       new JScrollPane(new CustomTableExample()), 525, 255,
                       "Using a CustomTableModel");
  }
}
```

Listing 15.17 `CustomTableModel.java`

```
import javax.swing.table.*;

/** A custom DefaultTableModel that returns the class
 *  type for the default cell renderers to use. The user is
 *  restricted to editing only the Comment and Visited columns.
 */

public class CustomTableModel extends DefaultTableModel {

  public Class getColumnClass(int column) {
    return(getValueAt(0, column).getClass());
  }

  // Only permit edit of "Comment" and "Visited" columns.
  public boolean isCellEditable(int row, int column) {
    return(column==3 || column==4);
  }
}
```

Table Event Handling

Table events are not directly handled by listeners attached to the `JTable`. Instead, events are handled by listeners attached to one or more of the three available models in a table: `TableModel`, `TableColumnModel`, and `ListSelectionModel`. To handle changes to cell values, attach a `TableModelListener` to the table's `TableModel`. The `TableModelListener` defines a single method, `tableChanged`, which receives a `TableModelEvent` when the data in the table is modified (observed when the user presses the Enter key or the cell loses focus). From the `TableModelEvent` you can call `getColumn` and `getFirstRow` or `getLastRow` to determine which table cell caused the event. Remember that a `DefaultTableModel` always returns `true` for `isCellEditable`, so unless you use default or custom cell editors, every cell in the table is editable and the default cell editor treats the cell as text that is rendered as a `JLabel`.

When the user presses the Enter key or moves to a different table cell, a `TableModelEvent` is fired. To handle this event, attach a `TableModelListener` to the `TableModel`, as below.

```
tablemode.addTableModelListener(
  new TableModelListner() {
    public void tableChanged(TableModelEvent event) {
      int row = event.getFirstRow();
      int column = event.getColumn();
      ...
    }
  });
```

Once you know which cell caused the event, you can retrieve the data through `getValueAt(row, column)` and, similarly, set the data in a table cell through `setValueAt(row, column)`. The

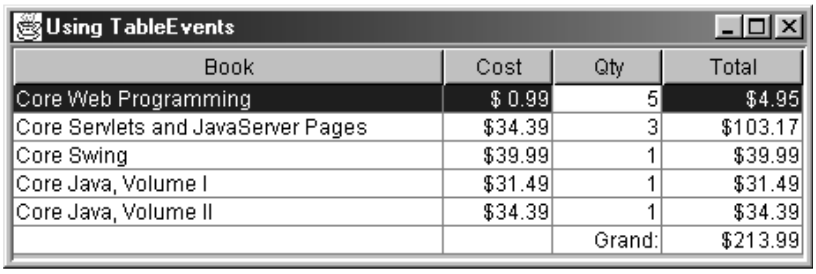
`getValueAt` method returns an `Object`, so you may need to cast the object to the appropriate subclass type. The data changes are actually made to the structure of the underlying data model. So, to update the `JTable` display, you must fire a `TableModelEvent` that refreshes the display.

The `AbstractTableModel` class, from which `TableModel` inherits, provides seven helper methods for firing events. The two most commonly used methods are `fireTableCellUpdated(row, column)`, which updates an individual cell, and `fireTableDataChanged()`, which updates the whole table.

The Java 2 Platform also defines `TableColumnModelEvent`s for changes to the `TableColumnModel`, and `ListSelectionEvent`s for changes to the `ListSelectionModel`.

Listing 15.18 creates a table from which the user can enter the number of books to purchase. The class imports the `javax.swing.event` package and attaches an event listener to the table model. When the user changes the number of desired books, the attached `TableModelListener` receives the `TableModelEvent` and a new total cost is calculated for the purchase. Once the calculation is completed and the data model updated, a `TableModelEvent` is fired to update the display. Figure 15-12 shows the results.

Figure 15-12. A table that calculates the grand total cost for a purchase order.



Much of the code in the event handler is for converting the data from `String`s to numerical values and back again. If you are unfamiliar with the `DecimalFormat` class, see the `java.text.DecimalFormat` API.

Listing 15.18 `JTableEvents.java`

```
import java.awt.*;
import java.text.DecimalFormat;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

/** A JTable that responds to TableModelEvents and
 *  updates other cells in the table, based on user input.
 */

public class JTableEvents extends JFrame {
  private final int COL_COST     = 1;
  private final int COL_QTY      = 2;
  private final int COL_TOTAL    = 3;
  private final int ROW_LAST     = 5;
  private DecimalFormat df = new DecimalFormat("$####.##");
  private JTable sampleJTable;
  private DefaultTableModel tableModel;
```

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

public JTableEvents() {
  super("Using TableEvents");
  WindowUtilities.setNativeLookAndFeel();
  addWindowListener(new ExitListener());
  Container content = getContentPane();

  String[] columnNames = { "Book", "Cost", "Qty", "Total" };

  final Object[][] data = {
    {"Core Web Programming", "$ 0.99", "0", "$0.00"},
    {"Core Servlets and JavaServer Pages",
                           "$34.39", "0", "$0.00"},
    {"Core Swing",         "$39.99", "0", "$0.00"},
    {"Core Java, Volume I",  "$31.49", "0", "$0.00"},
    {"Core Java, Volume II", "$34.39", "0", "$0.00"},
    {null, null,                "Grand:", "$0.00"} };
  tableModel = new DefaultTableModel(data, columnNames);
  tableModel.addTableModelListener(
    new TableModelListener() {
      int row, col;
      int quantity;
      float cost, subTotal, grandTotal;
      public void tableChanged(TableModelEvent event) {
        row = event.getFirstRow();
        col = event.getColumn();
        // Only update table if a new book quantity entered.
        if (col == COL_QTY) {
          try {
            cost = getFormattedCellValue(row, COL_COST);
            quantity = (int)getFormattedCellValue(row, COL_QTY);
            subTotal = quantity * cost;

            // Update row total.
            tableModel.setValueAt(df.format(subTotal),
                                  row, COL_TOTAL);
            // Update grand total.
            grandTotal =0;
            for(int row=0; row<data.length--1; row++) {
              grandTotal += getFormattedCellValue(row, COL_TOTAL);
            }
            tableModel.setValueAt(df.format(grandTotal),
                                  ROW_LAST,COL_TOTAL);
            tableModel.fireTableDataChanged();
          } catch (NumberFormatException nfe) {
              // Send error message to user.
              JOptionPane.showMessageDialog(
                          JTableEvents.this,
                          "Illegal value entered!");
          }
```

```
          }
        }

        private float getFormattedCellValue(int row, int col) {
          String value = (String)tableModel.getValueAt(row, col);
          return(Float.parseFloat(value.replace('$',' ')));
        }
      });

    sampleJTable = new JTable(tableModel);
    setColumnAlignment(sampleJTable.getColumnModel());
    JScrollPane tablePane = new JScrollPane(sampleJTable);

    content.add(tablePane, BorderLayout.CENTER);
    setSize(460,150);
    setVisible(true);
  }
  // Right--align all but the first column.
  private void setColumnAlignment(TableColumnModel tcm) {
    TableColumn column;
    DefaultTableCellRenderer renderer =
      new DefaultTableCellRenderer();
    for(int i=1; i<tcm.getColumnCount(); i++) {
      column = tcm.getColumn(i);
      renderer.setHorizontalAlignment(SwingConstants.RIGHT);
      column.setCellRenderer(renderer);
    }
  }
}
```

15.5 Swing Component Printing

The Java 2 Platform supports high-quality printing through classes in the `java.awt.print` package. This section describes how to use this printing package to print Swing components. In a sense, printing is little more than painting the component on the `Graphics2D` object and rendering the graphics on the printer. When printing in JDK 1.2, you must first globally turn off double buffering before painting the component. In JDK 1.3, the printing model changed and a `print` method was added to the `JComponent` class to automatically take care of double buffering. In the following sections, we look at the basics of printing and explain the importance of disabling double buffering before printing in JDK 1.2. Then, we illustrate the approach to printing in JDK 1.3.

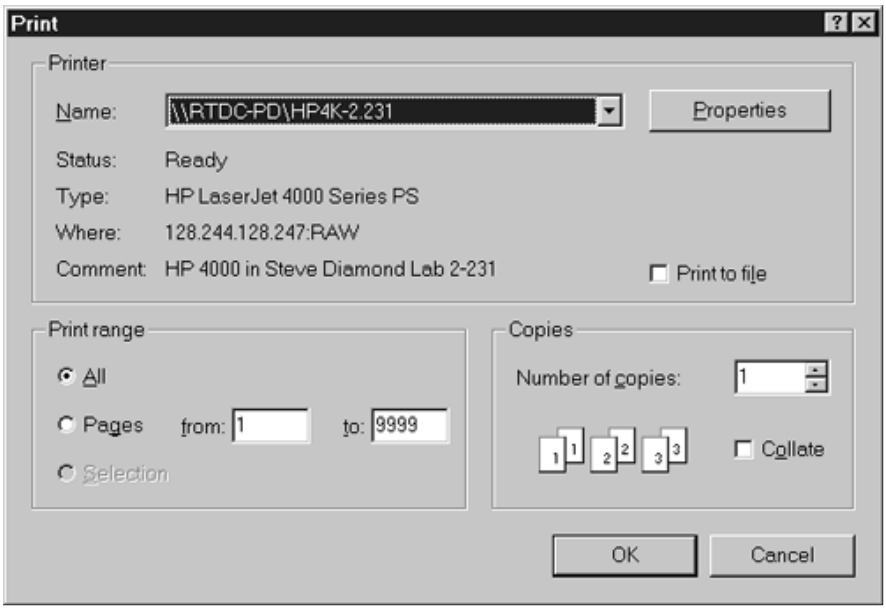
Printing Basics

Two steps are required to print: setting up the print job and rendering graphics on the printer.

Setting Up the Print Job

Setting up a print job is always done in the same manner: get a `PrinterJob` object, pass a `Printable` object to the `setPrintable` method, call `printDialog` to open an operating-system-specific print dialog, and finally, call `print` on the `PrinterJob`. Figure 15-13 shows a representative print dialog on Windows 98. The user can select Cancel from the print dialog, so you should check the return value from `printDialog` before calling `print`.

Figure 15-13. A typical display after `printDialog` is called.



The typical code for setting up a print job is:

```
PrinterJob printJob = PrinterJob.getPrinterJob();
printJob.setPrintable(this);
if (printJob.printDialog())
  try {
    printJob.print();
  } catch(PrinterException pe) {
      System.out.println("Error printing: " + pe);
}
```

Rendering Graphics on the Printer

The `Printable` that is passed to `setPrintable` must implement the `print` method that describes how to send the drawing to the printer. The print method accepts three arguments: the `Graphics` object, the `PageFormat`, and the `pageIndex`. `PrinterJob` calls the print method to render the graphics object according to the specified page format, where `pageIndex` specifies which page (in the document) to print. `PrinterJob` initially calls `print` with a `pageIndex` of 0. You should return `PAGE_EXISTS` if you have printed that page. Return `NO_SUCH_PAGE` to indicate that no pages are left.

In general, the first step is to decide what to do for different pages of your print job, since Java repeatedly calls `print` with higher and higher page indexes until print returns `NO_SUCH_PAGE`. In the specific case of printing the GUI interface, however, you basically print a single page document—the image on the screen. So, you return `PAGE_EXISTS` for index 0 and `NO_SUCH_PAGE` otherwise.

The second step is to start the drawing. For Swing components, your drawing should be a high-resolution version of what the component looks like on the screen. To create this drawing, cast the `Graphics` object to `Graphics2D`, scale the resolution of the object to that of the printer, and call the component's `paint` method with the scaled `Graphics2D` object. For more details on the `Graphics2D` object, see Chapter 10 (Java 2D: Graphics in Java 2).

The general approach for performing the print task is shown below.

```
public int print(Graphics g,
                 PageFormat pageFormat,
                 int pageIndex) {
```

```
  if (pageIndex > 0) {
    return(NO_SUCH_PAGE);
  } else {
    Graphics2D g2d = (Graphics2D)g;
    g2d.translate(pageFormat.getImageableX(),
                  pageFormat.getImageableY());

    // In JDK 1.2 you need to turn off double
    // buffering before painting the component,
    // and then turn double buffering back on.
    // In JDK 1.3 you would replace these three
    // lines with componentToBePainted.print(g2d).
    setDoubleBufferEnabled(false);
    componentToBePrinted.paint(g2d);
    setDoubleBufferEnabled(true);

    return(PAGE_EXISTS);
  }
}
```

The `getImageableX` and `getImageableY` methods return the upper-left coordinate of the page's printable image, which is based on the printer, paper size, and page orientation selected in Printer dialog window.

The Role of Double Buffering

With Swing, almost all components have double buffering turned on by default. In general, double buffering is a great boon, making for a convenient and efficient `paintComponent` method. However, in the specific case of printing in JDK 1.2, double buffering can be a huge problem. First, since component printing relies on scaling the coordinate system and then simply calling the component's `paint` method, if double buffering is enabled, then printing amounts to little more than scaling up the buffer (off-screen image). Scaling the off-screen image simply results in ugly low-resolution printing. Second, sending huge buffers to the printer produces large spooler files that take a very long time to print.

Consequently, you need to make sure double buffering is turned off before you print the Swing component. If you have only a single `JPanel` or other `JComponent`, you can call `setDoubleBuffered(false)` on the component before calling the `paint` method and then call `setDoubleBuffered(true)` afterward. However, this approach suffers from the flaw that if you later nest another container inside the component, the nested container has double buffering turned on by default and you're right back where you started. A better approach is to globally turn off double buffering through

```
RepaintManager currentManager =
  RepaintManager.currentManager(theComponent);
currentManager.setDoubleBufferingEnabled(false);
```

and then to reenable double buffering after calling `paint` through the `setDoubleBufferingEnabled(true)`. Although this approach will completely fix the problem with low-resolution printouts, if the components have large, complex filled backgrounds, you can still get big spool files and slow printing. In JDK 1.3, this step is automatically performed during the printing process and you no longer need to turn double buffering off and then on again.

Core Note

In JDK 1.2 printing you need to globally turn double buffering off before painting the component and then turn double buffering back on.

A General-Purpose Component-Printing Routine

The role of the `print` method when printing Swing components is to do nothing more than scale the `Graphics` object, turn off double buffering, and then call `paint`. Nothing in the API requires that the `print` method be placed in the component that is to be printed. Furthermore, requiring printable components to directly implement the `Printable` interface and define a `print` method produces cumbersome code and prevents you from printing components that were not originally planned for in the design. A better approach is to put the `print` method in an arbitrary object and tell that object which component's `paint` method to call when printing. This approach permits you to build a generic `printComponent` method to which you simply pass the component you want printed.

Core Approach

Instead of modifying the code for each Swing component you would like printed, simply provide a helper class that implements the `Printable` interface.

Listing 15.19 provides a utility class, `PrintUtilities`, for printing Swing components. Simply pass the component to the `PrintUtilities.printComponent` method. Listing 15.20 is an example of using the print utilities in JDK 1.2.

Listing 15.19 `PrintUtilities.java`

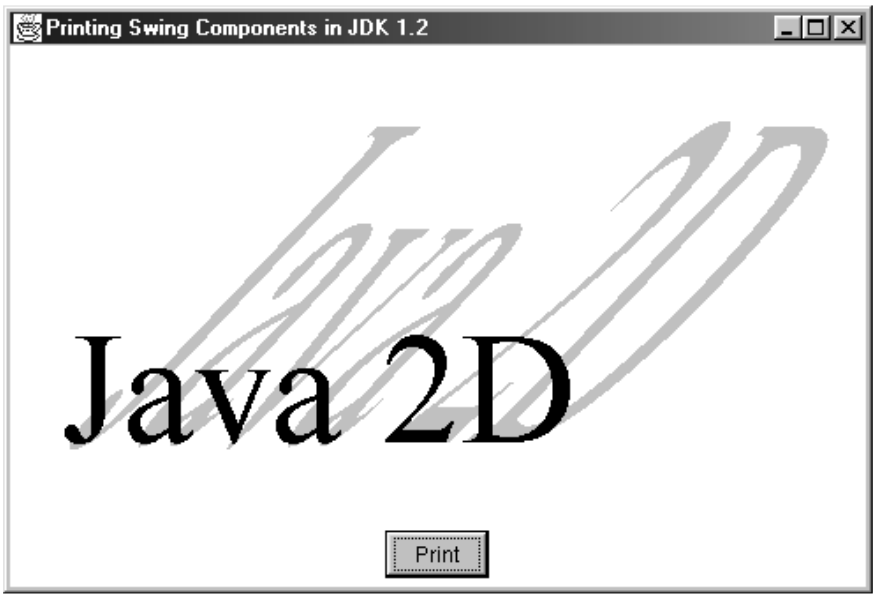
```
import java.awt.*;
import javax.swing.*;
import java.awt.print.*;

/** A simple utility class that lets you very simply print
 *  an arbitrary component in JDK 1.2. Just pass the
 *  component to PrintUtilities.printComponent. The
 *  component you want to print doesn't need a print method
 *  and doesn't have to implement any interface or do
 *  anything special at all.
 *  <P>
 *  If you are going to be printing many times, it is marginally
 *  more efficient to first do the following:
 *  <PRE>
 *    PrintUtilities printHelper =
 *      new PrintUtilities(theComponent);
 *  </PRE>
 *  then later do printHelper.print(). But this is a very tiny
 *  difference, so in most cases just do the simpler
 *  PrintUtilities.printComponent(componentToBePrinted).
 */

public class PrintUtilities implements Printable {
  protected Component componentToBePrinted;

  public static void printComponent(Component c) {
    new PrintUtilities(c).print();
  }

  public PrintUtilities(Component componentToBePrinted) {
    this.componentToBePrinted = componentToBePrinted;
```

```
  }

  public void print() {
    PrinterJob printJob = PrinterJob.getPrinterJob();
    printJob.setPrintable(this);
    if (printJob.printDialog())
      try {
        printJob.print();
      } catch(PrinterException pe) {
        System.out.println("Error printing: " + pe);
      }
  }

  // General print routine for JDK 1.2. Use PrintUtilities2
  // for printing in JDK 1.3.
  public int print(Graphics g, PageFormat pageFormat,
                   int pageIndex) {
    if (pageIndex > 0) {
      return(NO_SUCH_PAGE);
    } else {
      Graphics2D g2d = (Graphics2D)g;
      g2d.translate(pageFormat.getImageableX(),
                    pageFormat.getImageableY());
      disableDoubleBuffering(componentToBePrinted);
      componentToBePrinted.paint(g2d);
      enableDoubleBuffering(componentToBePrinted);
      return(PAGE_EXISTS);
    }
  }

  /** The speed and quality of printing suffers dramatically if
   *  any of the containers have double buffering turned on,
   *  so this turns it off globally.  This step is only
   *  required in JDK 1.2.
   */

  public static void disableDoubleBuffering(Component c) {
    RepaintManager currentManager =
                      RepaintManager.currentManager(c);
    currentManager.setDoubleBufferingEnabled(false);
  }

  /** Reenables double buffering globally. This step is only
   *  required in JDK 1.2.
   */

  public static void enableDoubleBuffering(Component c) {
    RepaintManager currentManager =
                      RepaintManager.currentManager(c);
    currentManager.setDoubleBufferingEnabled(true);
  }
}
```

Listing 15.20 adds print capabilities to a `JFrame` by including a button that simply calls

`PrintUtilities.printComponent` when selected. The frame includes a `DrawingPanel`, Listing 15.21, with a custom `paintComponent` method to draw the text "Java 2D" in a shadow effect, as shown in Figure 15-14. See Chapter 10 (Java 2D: Graphics in Java 2) for information about performing transformations and drawing on the `Graphics2D` object.

Figure 15-14. Selecting the `JButton` invokes the print service to send the `Graphics2D` object to the printer.



Listing 15.20 `PrintExample.java`

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

/** An example of a printable window in Java 1.2. The key point
 *  here is that <B>any</B> component is printable in Java 1.2.
 *  However, you have to be careful to turn off double buffering
 *  globally (not just for the top--level window).
 *  See the PrintUtilities class for the printComponent method
 *  that lets you print an arbitrary component with a single
 *  function call.
 */

public class PrintExample extends JFrame
                          implements ActionListener {
  public static void main(String[] args) {
    new PrintExample();
  }

  public PrintExample() {
    super("Printing Swing Components in JDK 1.2");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    JButton printButton = new JButton("Print");
```

```
    printButton.addActionListener(this);
    JPanel buttonPanel = new JPanel();
    buttonPanel.setBackground(Color.white);
    buttonPanel.add(printButton);
    content.add(buttonPanel, BorderLayout.SOUTH);
    DrawingPanel drawingPanel = new DrawingPanel();
    content.add(drawingPanel, BorderLayout.CENTER);
    pack();
    setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    PrintUtilities.printComponent(this);
  }
}
```

Listing 15.21 `DrawingPanel.java`

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

/** A window with a custom paintComponent method.
 *  Illustrates that you can make a general--purpose method
 *  that can print any component, regardless of whether
 *  that component performs custom drawing.
 *  See the PrintUtilities class for the printComponent method
 *  that lets you print an arbitrary component with a single
 *  function call.
 */

public class DrawingPanel extends JPanel {
  private int fontSize = 90;
  private String message = "Java 2D";
  private int messageWidth;

  public DrawingPanel() {
    setBackground(Color.white);
    Font font = new Font("Serif", Font.PLAIN, fontSize);
    setFont(font);
    FontMetrics metrics = getFontMetrics(font);
    messageWidth = metrics.stringWidth(message);
    int width = messageWidth*5/3;
    int height = fontSize*3;
    setPreferredSize(new Dimension(width, height));
  }

  /** Draws a black string with a tall angled "shadow"
   *  of the string behind it.
   */

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
```

```
    int x = messageWidth/10;
    int y = fontSize*5/2;
    g2d.translate(x, y);
    g2d.setPaint(Color.lightGray);
    AffineTransform origTransform = g2d.getTransform();
    g2d.shear(--0.95, 0);
    g2d.scale(1, 3);
    g2d.drawString(message, 0, 0);
    g2d.setTransform(origTransform);
    g2d.setPaint(Color.black);
    g2d.drawString(message, 0, 0);
  }
}
```

Printing in JDK 1.3

The burden of disabling and enabling double buffering during the printing process in JDK 1.2 was lifted in JDK 1.3. Now, `JComponent` provides its own `print` method to facilitate printing. The `print` method simply sets an internal boolean flag, `IS_PRINTING`, to `true` and calls `paint`. The `paint` method in `JComponent` was rewritten to check the flag and, if set to disable double buffering before calling three internal `protected` methods (`printComponent`, `printBorder`, and `printChildren`) to print the components.

Now, when printing in JDK 1.3, use

```
componentToBePrinted.print(g2d);
```

instead of

```
setDoubleBufferEnabled(false);
componentToBePrinted.paint(g2d);
setDoubleBufferEnabled(true);
```

to facilitate printing.

Core Approach

For printing in JDK 1.3, call the component's `print(Graphics2d g2d)` method. Disabling and enabling double buffering during the print process is no longer required.

In Listing 15.22 we provide a new utility, `PrintUtilities2`, for printing in JDK 1.3. This class inherits from `PrintUtilities` (Listing 15.19) and overrides the `print` method to call the component's `print` method instead of the component's `paint` method. To use this class for printing, call

```
PrintUtilities2.printComponent(componentToBePrinted);
```

where `componentToBePrinted` is the Swing component to print.

Listing 15.22 `PrintUtilities2.java`

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

```
/** A simple utility class for printing an arbitrary
 *  component in JDK 1.3. The class relies on the
 *  fact that in JDK 1.3 the JComponent class overrides
 *  print (in Container) to automatically set a flag
 *  that disables double buffering before the component
 *  is painted. If the printing flag is set, paint calls
 *  printComponent, printBorder, and printChildren.
 *
 *  To print a component, just pass the component to
 *  PrintUtilities2.printComponent(componentToBePrinted).
 */

public class PrintUtilities2 extends PrintUtilities {

  public static void printComponent(Component c) {
    new PrintUtilities2(c).print();
  }

  public PrintUtilities2(Component componentToBePrinted) {
    super(componentToBePrinted);

  }

  // General print routine for JDK 1.3. Use PrintUtilities1
  // for printing in JDK 1.2.
  public int print(Graphics g, PageFormat pageFormat,
                   int pageIndex) {
    if (pageIndex > 0) {
      return(NO_SUCH_PAGE);
    } else {
      Graphics2D g2d = (Graphics2D)g;
      g2d.translate(pageFormat.getImageableX(),
                    pageFormat.getImageableY());
      componentToBePrinted.print(g2d);
      return(PAGE_EXISTS);
    }
  }
}
```

15.6 Swing Threads

Swing follows the familiar AWT event model and uses a single-threaded design for updating components. Recall from Chapter 11 (Handling Mouse and Keyboard Events) that painting of components and handling of listeners are both processed in a single event dispatch thread. As a consequence of the event model, two general rules apply for robust GUI design:

1. If tasks in the event handling method require considerable CPU time, then execute these time-intensive tasks in a separate thread. Freeing the event dispatch thread to process other queued events yields a more responsive user interface.

2. Make changes to the state of a realized (visible) Swing component only within the event dispatch thread and not within a user-defined thread.

If you do not follow these design rules when modifying a Swing component, your GUI could be

unresponsive to user interaction or race conditions could occur for the state of a component.

To illustrate potential design problems, we consider a simple application that transfers a file from a client machine to a remote server. Suppose that the GUI design includes a Start button to initiate the file transfer and a label that shows the status of the file transfer (i.e., "Transferring *filename* …"). An initial, albeit poor, design for the `actionPerformed` method of the Start button is shown below:

```
// Poorly designed event handler for Start button.
public void actionPerformed(ActionEvent event){

  // Change to label is ok here because it is executed
  // on the event dispatch thread.
  label.setText("Transferring " + filename);

  // Transfer file to server - time intensive.
  transferFile(filename);

  // Ok to change label here also, executed on
  // event dispatch thread.
  label.setText("Transfer completed.");
}
```

Here, changing the text on the label is perfectly legal because the `setText` method is executed within the event dispatch thread. However, the results are not what one might expect. Each call to `setText` places a `PropertyChangeEvent` on the event queue, and only *after* the file is transferred and the `actionPerformed` method is completed are the change events dispatched from the queue. The initial update to the label does not occur until *after* the file is transferred. The second change to the label immediately occurs and the user never sees the "Transferring …" label message. Furthermore, because the file transfer is executing in the event dispatch thread, user interaction with any other components in the user interface is effectively blocked until the current `actionPerformed` method completes. This is definitely not the desired behavior!

One possible solution to the dilemma is to perform the file transfer and final label update in a separate user thread and free the event dispatch thread to process the queued events. Once the thread completes the file transfer, the thread can communicate the completion to the `JLabel`. Consider the improved `actionPerformed` method,

```
// Improved event handler. Time-intensive task is
// moved to a separate thread.
public void actionPerformed(ActionEvent event) {
  // Change to label is ok here since it is executed
  // on the event thread.
  label.setText("Transferring " + filename);

  // Transferring the file is time intensive, so the
  // task is performed in a separate thread to permit
  // processing of other events on the event queue.
  Thread t = new FileTransfer(filename, label);
  t.start();
}
```

and supporting threaded class, `FileTransfer`,

```
// Improperly designed thread class - update
// of Swing component is not thread safe.
public class FileTransfer extends Thread {
```

```
  private String filename;
  private JLabel = label;

  public FileTransfer(String filename, JLabel label) {
    this.filename = filename;
    this.label = label;
  }

  public void run() {
    // Transfer file to server. Lengthy process.
    doTransfer(...);

    // Update of label is not ok. Update is not
    // executed on the event dispatch thread.
    label.setText("Transfer complete.");
  }
}
```

This design certainly frees up the event dispatch thread to handle other events on the queue. However, because the `FileTransfer` thread directly modifies the state of the `JLabel`, the second design rule is violated—update Swing components only in the event dispatch thread. If you don't follow this rule, then a race condition could occur when the event dispatch thread is relying on one value for the label and the user-defined thread changes the label to a different value. Depending on the scheduling of the threads on the CPU, two different outcomes could be observed for multiple runs of the program.

To resolve this problem, Swing provides two methods, `invokeLater` and `invokeAndWait`, to place a `Runnable` object on the event queue for updating Swing components. Once the `Runnable` object is dispatched from the front of the event queue, the object's `run` method is executed in the event dispatch thread. The idea here is to place the label update in the `run` method of the `Runnable` object. An example of creating a `Runnable` object is illustrated shortly. For more information on multithreading and `Runnable` objects, see Chapter 16 (Concurrent Programming with Java Threads).

Core Approach

To properly access Swing components from a thread other than the event dispatch thread, place the code in a `Runnable` object and then use one of the static `SwingUtilities` methods, `invokeLater` or `invokeAndWait`, to place the `Runnable` object on the event queue.

SwingUtilities Methods

The `SwingUtilities` class provides two methods to queue a `Runnable` object on the event dispatch thread. Changes to a Swing component from a user thread should be wrapped (placed) in the `run` method of the `Runnable` object.

public static void invokeLater(Runnable object)

The `invokeLater` method places the `Runnable` object on the event queue and immediately returns. Once the runnable object reaches the front of the event queue, the object's `run` method is executed on the event dispatch thread. You can call `invokeLater` from within the event dispatch thread and any user thread.

public static void invokeAndWait(Runnable object) throws InterruptedException, InvocationTargetException

> The `invokeAndWait` method places the `Runnable` object on the event queue and blocks (waits) until the object's `run` method completes execution. Use this method when the Swing component must be updated before the thread proceeds to the next operation or when information from the component must be first retrieved before the thread proceeds. You cannot call `invokeAndWait` from the event dispatch thread because doing so produces as a deadlock condition. Use `isEventDispatchThread` to determine if the current thread is the event dispatch thread. The possibly thrown `InvocationTargetException` is located in the `java.lang.reflect` package.

Next, we illustrate the general approach for creating a `Runnable` object to update a Swing component on the event dispatch thread. Consider the situation described earlier where a file is transferred to a remote server and a label is updated on the user interface to indicate the status of the file transfer. As shown earlier, transferring the file in the event dispatch thread cripples the responsiveness of the event handler. Moving the task to a user-defined thread and modifying the Swing label from a user-defined thread violates the fundamental design rule of only modifying Swing components during execution in the event dispatch thread. Failure to follow this rule could produce a race condition. Below we provide our final design for the `actionPerformed` method and `FileTransfer` class. In this final design, we update the label in a thread-safe manner.

```
// Correctly designed event handler. Time-intensive
// task is moved to a separate thread and modification
// of the Swing component from the thread is performed
// in a Runnable object placed on the event queue.

public void actionPerformed(ActionEvent event) {

    // Time-intensive task is moved to a separate thread.
    Thread t = new FileTransfer(filename, label);
    t.start();
  }
}
```

In the `FileTransfer` constructor, a reference to the `JLabel` is provided so that the `Runnable` object can update the label. Two `Runnable` objects are created, one to update the label before the file transfer and one to update the label after the file transfer. Here, each `Runnable` object is actually defined as an anonymous class. See Chapter 11 (Handling Mouse and Keyboard Events) for further examples on creating and using anonymous classes.

```
// Final version of FileTransfer. Modification of the
// label is thread safe.
public class FileTransfer extends Thread {
  private String filename;
  private JLabel label;

  public FileTransfer(String filename, JLabel label) {
    this.filename = filename;
    this.label = label;
  }
  public void run() {

    try {
      // Place the runnable object to update the label
      // on the event queue. The invokeAndWait method
      // will block until the label is updated.
      SwingUtilities.invokeAndWait(
        new Runnable() {
```

```
            public void run() {
              label.setText("Transferring " + filename);
            }
          });
      } catch(InvocationTargetException ite) {
      } catch(InterruptedException ie) { }

      // Transfer file to server. Lengthy process.
      doTransfer(...);

      // Perform the final update to the label from
      // within the runnable object. Use invokeLater;
      // blocking is not necessary.
      SwingUtilities.invokeLater(
         new Runnable() {
           public void run() {
             label.setText("Transfer completed");
           }
         });
  }
}
```

In this design we use `invokeAndWait` to place the first `Runnable` object on the event queue, ensuring that the label is updated to "Transferring *file*" before we proceed with the actual file transfer. After the file transfer is completed, another `Runnable` object is created and placed on the event queue by `invokeLater`. Here, we are not so concerned that the label is updated before we proceed. Eventually, the object will reach the front of the event queue and the `run` method will execute, changing the label to "Transfer completed."

As we've shown, you need to pay special attention to updating Swing components from threads other than the event dispatch thread. The need to move tasks to separate threads actually occurs quite often, for example, when querying databases, communicating with remote objects by means of RMI, accessing Web servers, and transferring files. Always remember that if you modify a component from a user-defined thread, use either `invokeLater` or `invokeAndWait` to change the component.

15.7 Summary

Now you have the skillset to create outstanding graphical user interfaces with the advanced Swing components. In this chapter, we've only shown you the basic functions of each component. You'll need to spend time developing your own Swing interfaces to appreciate the full capabilities of these advanced components.

A `JList`, `JTree`, and `JTable` all have default models for accessing and modifying their data (as well as firing change events), so you can get started right away or you can write your own custom models to interact with the data and visual presentation directly. Technically, you can display almost any Swing component in a list, tree, or table, but you must provide a custom cell renderer to display the component correctly.

Once you've created a Swing interface, printing the component simply requires you to provide the class with a `print` method. But the `print` method is not required to be in the components class, so it's often useful to write a separate utility class to handle the printing. Finally, if you modify a Swing component, you must do so in a thread-safe manner. Swing provides both `invokeAndWait` and `invokeLater` to ensure that modification of the component is executed in the event dispatch thread.

CONTENTS