Chapter 16

Choosers

This chapter discusses two Swing chooser components—`JFileChooser` and `JColorChooser`—that are used to select files and colors, respectively.

JFileChooser

File choosers, like option panes—see "JOptionPane" on page 815—are lightweight components that are meant to be placed in a dialog. Once an instance of `JFileChooser` has been instantiated, it can be added to a dialog. Additionally, the `JFileChooser` class provides methods that add an existing file chooser to a modal dialog and display the dialog. The methods return an `integer` value indicating whether the chooser's approve button was activated or the dialog was dismissed.

File choosers support three display modes: files only, directories only, and files and directories. Additionally file choosers support both single and multiple selection.[1]

1. Multiple selection is not fully supported in Swing 1.1 FCS

File choosers can be customized in a number of different ways, as Figure 16-1 illustrates. The top picture in Figure 16-1 shows the standard dialog displayed as a result of invoking `JFileChooser.showSaveDialog()`. The bottom picture in Figure 16-1 shows a file chooser with customized text for the dialog title, approve button and its tooltip, custom filters, custom icons, and an accessory component.

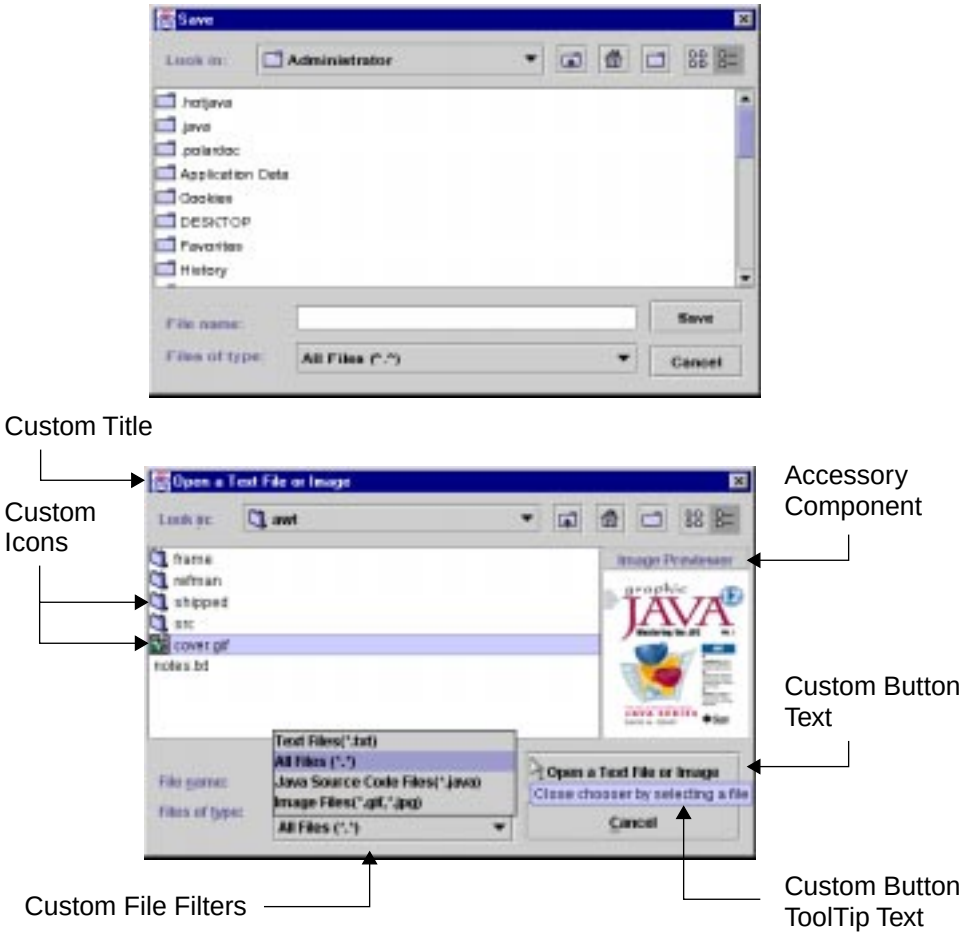


Figure 16-1 Customizing File Choosers

File chooser features are summarized in Feature Summary 16-1.

Feature Summary 16-1 JFileChooser

Open/Save/Custom Dialogs

`JFileChooser` provides three methods that display a *modal* dialog containing choosers: `showOpenDialog()`, `showSaveDialog()`, and `showDialog()`.[2] Each of the methods sets the dialog title and the string displayed in the file chooser's approve button and returns an `integer` value indicating whether the approve button was activated or the dialog was canceled.

Display Modes

`JFileChooser` supports three display modes: files only, directories only, and files and directories.

Multiple Selection

File choosers come with both single and multiple selection; however, multiple selection is not fully supported in Swing 1.1.

Approve Button

Three facets of a file chooser's approve button can be customized: the button's text, its tooltip text, and mnemonic.

File Filters

Specific files or types of files can be filtered from a file chooser. File choosers can have multiple filters, but only one is active at any given time.

File Views

Given a file, the icons and file names displayed in a file chooser are obtained from a file view—an object that extends the `FileView` class. File choosers can be fitted with custom file views.

2. The methods do not return until the dialogs are dismissed.

Accessory Component

A component can be installed in a file chooser for any number of purposes. Previewers for specific types of files—such as image previewers—represent one use for accessory components.

Figure 16-2 shows an application that displays a file dialog. After the dialog is dismissed, a message dialog communicates the selected file (or lack thereof). The top picture in Figure 16-2 shows the application as it appears initially, and the middle picture shows the file chooser displayed as a result of activation of the application's button. The bottom picture shows the message dialog displayed after a file has been selected.

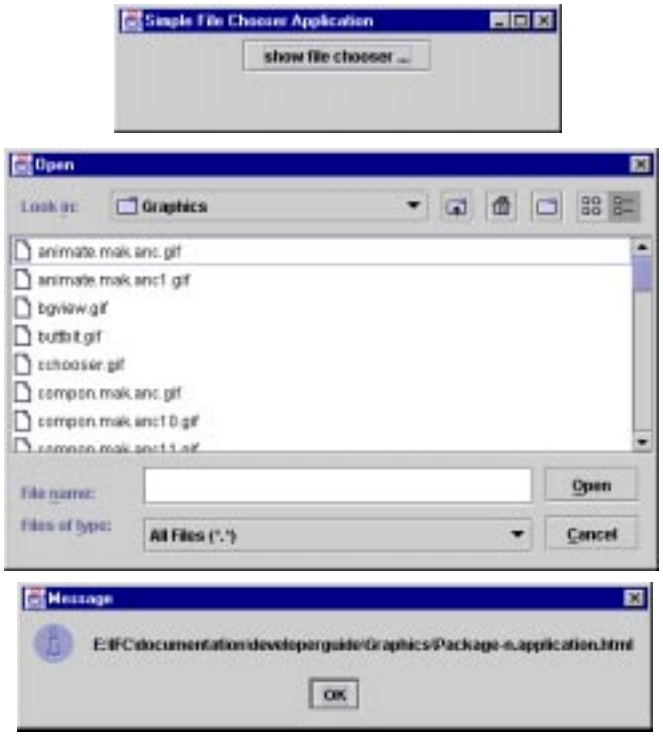


Figure 16-2 A Simple File Chooser Example

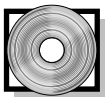
The application creates a button that is added to the application frame's content pane. The application also creates a file chooser with the `JFileChooser` no-argument constructor.

When the button contained in the application is activated, `JFileChooser.showOpenDialog()` is invoked to display the file chooser in a modal dialog. Because the dialog is modal, the call to `showOpenDialog()` does not return until the dialog is dismissed.

After the dialog is dismissed, `JFileChooser.getSelectedFile()` obtains a reference to the file that was selected in the file chooser. Subsequently, a message dialog is displayed that indicates whether the chooser's approve button was activated or the dialog was canceled, depending upon the value returned from `JFileChooser.showOpenDialog()`.

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int state = chooser.showOpenDialog(null);
                File file = chooser.getSelectedFile();

                if(file != null &&
                   state == JFileChooser.APPROVE_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, file.getPath());
                }
                else if(state == JFileChooser.CANCEL_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, "Canceled");
                }
            }
        });
    }
```

The application shown in Figure 16-2 is listed in its entirety in Example 16-1.



Example 16-1 A Simple File Chooser Example

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

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int state = chooser.showOpenDialog(null);
                File file = chooser.getSelectedFile();

                if(file != null &&
                   state == JFileChooser.APPROVE_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, file.getPath());
                }
                else if(state == JFileChooser.CANCEL_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, "Canceled");
                }
            }
        });
    }
    public static void main(String args[]) {
        JFrame f = new Test();
        f.setBounds(300,300,350,100);
        f.setVisible(true);

        f.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

Swing Tip ...

Check for null Selected Files

If a file chooser's approve button is activated without a file being selected, the file returned from JFileChooser.getSelectedFile() will be null. As a result, before a file returned from JFileChooser.getSelectedFile() is used, a check should be made to ensure that a file has actually been selected.

For example, the application listed in Example 16-1 makes the following check:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int state = chooser.showOpenDialog(null);
        File file = chooser.getSelectedFile();

        if(file != null && state == JFileChooser.APPROVE_OPTION)
            JOptionPane.showMessageDialog(null, file.getPath());
        ...
    ...
}
```

In other words, just because a JFileChooser.show...() method returns JFileChooser.APPROVE_OPTION does not necessarily mean that a file was selected.

File Chooser Types

The `JFileChooser` class supports three types of preconfigured file choosers and associated dialogs—custom, open, and save—that are differentiated by the dialog title and the text used for the file chooser's approve button.

`JFileChooser` dialog types are listed in Table 16-1, along with the `JFileChooser` instance methods used to show the dialogs.

Table 16-1 JFileChooser Dialog Types

Dialog Type	Method Used to Show Dialog	Dialog Title / Approve Button Text
Custom	int showDialog(Component parent, String approveButtonText)	Specified by second argument
Open	int showOpenDialog(Component parent)	Open
Save	int showSaveDialog(Component parent)	Save

The methods listed in Table 16-1 display a modal dialog containing a file chooser and return an `integer` value after the dialog has been dismissed, signifying whether the chooser's approve button was activated or the dialog was canceled. The dialog is centered over the parent component that is passed to the methods. If the methods are passed a `null` value for the parent component, the dialog is centered on the screen.

The following constants are returned from the methods listed in Table 16-1.

- `JFileChooser.APPROVE_OPTION`
- `JFileChooser.CANCEL_OPTION`

The first method listed in Table 16-1 is passed a string that is used as the text for the file chooser's approve button and the dialog's title. The last two methods listed in Table 16-1 set the dialog title and the file chooser's approve button text to "Open" and "Save", respectively.

Figure 16-3 shows an application that uses the methods listed above to display a file chooser in a modal dialog. The application contains a combo box that allows the type of file chooser to be displayed and a button that displays the dialog. The top pictures in Figure 16-3 show a "Save" file chooser, and the middle pictures show an "Open" file chooser. The bottom pictures show a custom file chooser whose approve button text and dialog title are specified with an input dialog.

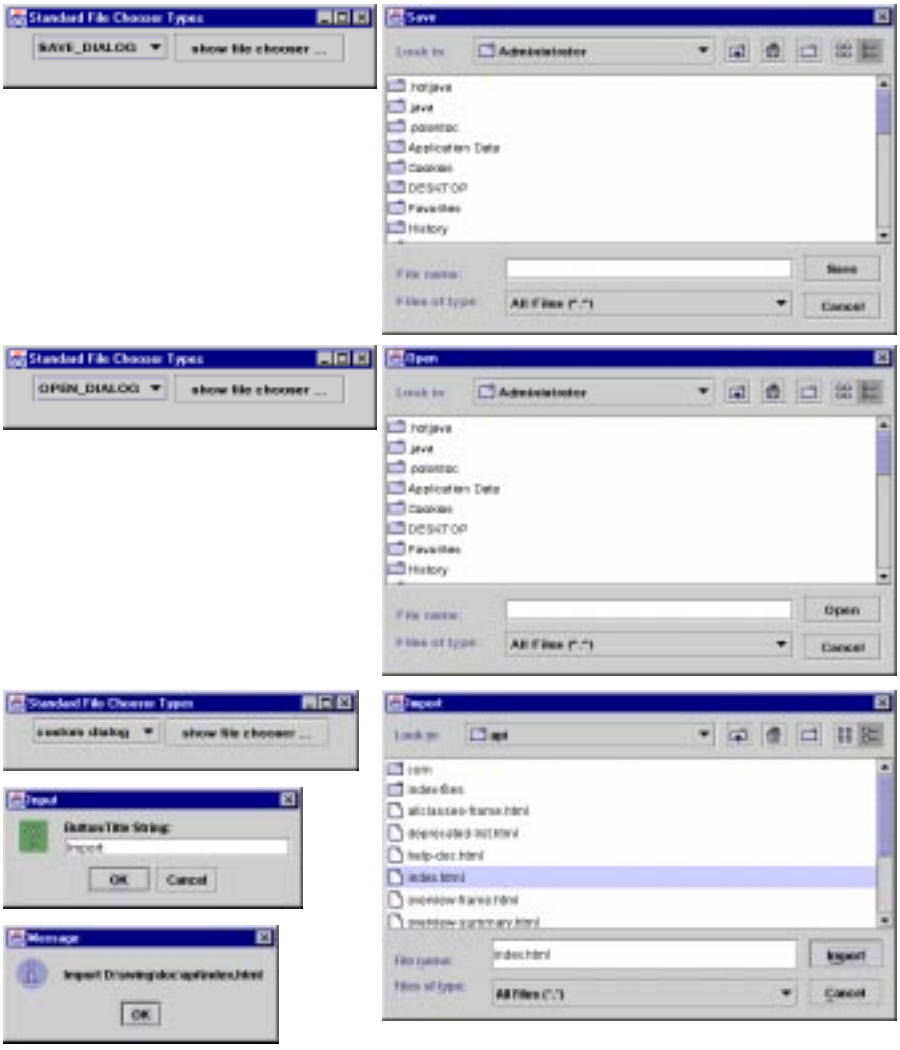


Figure 16-3 Default File Chooser Types

The application creates a file chooser, a combo box, and a button; the combo box and button are subsequently added to the application frame's content pane. When the button is activated, the file chooser is displayed in either an open, save, or custom dialog, depending upon the value selected from the combo box. After the dialog has been dismissed, a message dialog is displayed that indicates whether a file was selected or the dialog was canceled.

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JComboBox comboBox = new JComboBox();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Standard File Chooser Types");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(comboBox);
        contentPane.add(button);

        comboBox.addItem("OPEN_DIALOG");
        comboBox.addItem("SAVE_DIALOG");
        comboBox.addItem("custom dialog");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String message = "CANCELED";
                int state = showChooser(
                          (String)comboBox.getSelectedItem());
                File file = chooser.getSelectedFile();

                if(file != null &&
                    state == JFileChooser.APPROVE_OPTION) {
                    message = chooser.getApproveButtonText() +
                              " " + file.getPath();
                }
                JOptionPane.showMessageDialog(null, message);
            }
        });
    }
```

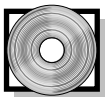
The application's `showChooser` method is passed the string selected in the combo box and displays an appropriate file chooser dialog. If custom dialog was selected from the combo box, an input dialog is displayed to obtain the string used for the file chooser's approve button text and the dialog's title.

```
    private int showChooser(String s) {
        int state;

        if(s.equals("OPEN_DIALOG")) {
            state = chooser.showOpenDialog(null);
        }
        else if(s.equals("SAVE_DIALOG")) {
            state = chooser.showSaveDialog(null);
        }
        else { // custom dialog
```


```
            String string = JOptionPane.showInputDialog(
                                    null,
                                    "Button/Title String:");

            chooser.setApproveButtonMnemonic(string.charAt(1));
            state = chooser.showDialog(Test.this, string);
        }
        return state;
    }
```

The application shown in Figure 16-3 is listed in its entirety in Example 16-2.

Example 16-2 Default File Chooser Types



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

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JComboBox comboBox = new JComboBox();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Standard File Chooser Types");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(comboBox);
        contentPane.add(button);

        comboBox.addItem("OPEN_DIALOG");
        comboBox.addItem("SAVE_DIALOG");
        comboBox.addItem("custom dialog");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String message = "CANCELED";
                int state = showChooser(
                          (String)comboBox.getSelectedItem());
                File file = chooser.getSelectedFile();

                if(file != null &&
                    state == JFileChooser.APPROVE_OPTION) {
                    message = chooser.getApproveButtonText() +
                              " " + file.getPath();
                }
                JOptionPane.showMessageDialog(null, message);
            }
        });
```

```
    }
    private int showChooser(String s) {
        int state;

        if(s.equals("OPEN_DIALOG")) {
            state = chooser.showOpenDialog(null);
        }
        else if(s.equals("SAVE_DIALOG")) {
            state = chooser.showSaveDialog(null);
        }
        else { // custom dialog
            String string = JOptionPane.showInputDialog(
                                    null,
                                    "Button/Title String:");

            chooser.setApproveButtonMnemonic(string.charAt(1));
            state = chooser.showDialog(Test.this, string);
        }
        return state;
    }
    public static void main(String args[]) {
        JFrame f = new Test();
        f.setBounds(300,300,350,100);
        f.setVisible(true);

        f.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

Accessory Components

File choosers can accommodate an accessory component, as illustrated in Figure 16-4. The accessory component is placed to the right of the file listing area[3] and is sized as tall as the file listing area and as wide as the accessory component's preferred width. Accessory components can be used for any purpose, the most common of which is a previewer that allows the content of files to be displayed as files are selected in the file chooser.

3. Actual placement is look-and-feel dependent

Figure 16-4 shows a file chooser equipped with an image previewer accessory component. When a GIF or JPEG file is selected, the image previewer displays a scaled version of the image.

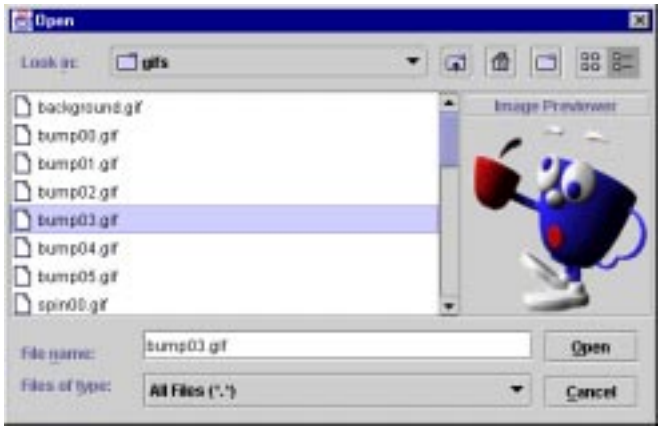


Figure 16-4 An Image Previewer Accessory

The application shown in Figure 16-4 implements an `ImagePreviewer` class that extends `JLabel`. When a file is selected in the file chooser, `ImagePreviewer.configure()` is passed a reference to the selected file. The image displayed in the previewer is an image icon created with the selected file's path.

After the image icon is created, its image is reset to a scaled version of the image by invocation of `Image.getScaledInstance()`. The scaled version of the image is used to create another image icon that is used as the label's icon.

```
class ImagePreviewer extends JLabel {
   public void configure(File f) {
      Dimension size = getSize();
      Insets insets = getInsets();

      // used to create full-size image
      ImageIcon icon = new ImageIcon(f.getPath());

      // set label's icon to new image icon
      // with scaled image
      setIcon(
       new ImageIcon(icon.getImage().getScaledInstance(
                size.width - insets.left - insets.right,
```

```
                    size.height - insets.top - insets.bottom,
                    Image.SCALE_SMOOTH)));
            ...
        }
        ...
    }
```

Note: The `configure` method could create the original image by hand, thereby eliminating the need for a second image icon, but the implementation presented here is more readable.

The image previewer is placed in an instance of `PreviewPanel`, which is an extension of `JPanel`. The `PreviewPanel` class sets its preferred size to a dimension of `(150,0)` because the preferred height of accessory components is ignored.[4]

The `PreviewPanel` class also sets its border to an etched border and adds the previewer as its center component. A label that identifies the previewer is specified as the preview panel's north component.

```
class PreviewPanel extends JPanel {
    public PreviewPanel() {
        JLabel label = new JLabel("Image Previewer",
                                  SwingConstants.CENTER);
        setPreferredSize(new Dimension(150,0));
        setBorder(BorderFactory.createEtchedBorder());

        setLayout(new BorderLayout());

        label.setBorder(BorderFactory.createEtchedBorder());
        add(label, BorderLayout.NORTH);
        add(previewer, BorderLayout.CENTER);
    }
    ...
}
```

The application creates a button, file chooser, image previewer, and preview panel. The button is added to the application frame's content pane, and the preview panel is specified as the file chooser's accessory component by invocation of `JFileChooser.setAccessory()`.

```
public class Test extends JFrame {
```

4. Results may vary with nonstandard look and feels.


```
JFileChooser   chooser = new JFileChooser();
ImagePreviewer previewer = new ImagePreviewer();
PreviewPanel   previewPanel = new PreviewPanel();
...

public Test() {
    super("Image Previewer Accessory");

    Container contentPane = getContentPane();
    JButton button = new JButton("Select A File");

    contentPane.setLayout(new FlowLayout());
    contentPane.add(button);

    chooser.setAccessory(previewPanel);
    ...
```

An action listener that invokes `JFileChooser.showOpenDialog()` is added to the button. Subsequently, a message dialog is displayed indicating whether a file was selected or the dialog was canceled.

```
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int state = chooser.showOpenDialog(null);
        File file = chooser.getSelectedFile();
        String s = "CANCELED";

        if(file != null &&
            state == JFileChooser.APPROVE_OPTION) {
            s = "File Selected: " + file.getPath();
        }
        JOptionPane.showMessageDialog(null, s);
    }
});
...
```

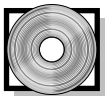
A property change listener that reacts to file selections is added to the file chooser. If the selected file has a suffix of "`.gif`" or "`.jpg`", the previewer is configured by invocation of `ImagePreviewer.configure()`.

```
...
chooser.addPropertyChangeListener(
                        new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        if(e.getPropertyName().equals(
```

```
                    JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                    File f = (File)e.getNewValue();
                    String s = f.getPath(), suffix = null;
                    int i = s.lastIndexOf('.');

                    if(i > 0 &&  i < s.length() - 1)
                        suffix = s.substring(i+1).toLowerCase();

                    if(suffix.equals("gif") ||
                       suffix.equals("jpg"))
                        previewer.configure(f);
                }
            }
        });
    }
    ...
}
```

The application shown in Figure 16-4 is listed in its entirety in Example 16-3.

Example 16-3 An Image Previewer Accessory



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

public class Test extends JFrame {
    JFileChooser   chooser = new JFileChooser();
    ImagePreviewer previewer = new ImagePreviewer();
    PreviewPanel   previewPanel = new PreviewPanel();

    class PreviewPanel extends JPanel {
        public PreviewPanel() {
            JLabel label = new JLabel("Image Previewer",
                                      SwingConstants.CENTER);
            setPreferredSize(new Dimension(150,0));
            setBorder(BorderFactory.createEtchedBorder());

            setLayout(new BorderLayout());

            label.setBorder(BorderFactory.createEtchedBorder());
            add(label, BorderLayout.NORTH);
            add(previewer, BorderLayout.CENTER);
        }
    }
    public Test() {
        super("Image Previewer Accessory");
```

```
    Container contentPane = getContentPane();
    JButtonbutton = new JButton("Select A File");

    contentPane.setLayout(new FlowLayout());
    contentPane.add(button);

    chooser.setAccessory(previewPanel);


    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            int state = chooser.showOpenDialog(null);
            File file = chooser.getSelectedFile();
            String s = "CANCELED";

            if(file != null &&
                state == JFileChooser.APPROVE_OPTION) {
                s = "File Selected: " + file.getPath();
            }
            JOptionPane.showMessageDialog(null, s);
        }
    });

    chooser.addPropertyChangeListener(
                            new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent e) {
            if(e.getPropertyName().equals(
               JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                File f = (File)e.getNewValue();
                String s = f.getPath(), suffix = null;
                int i = s.lastIndexOf('.');

            if(i > 0 &&  i < s.length() - 1)
                suffix = s.substring(i+1).toLowerCase();

                if(suffix.equals("gif") ||
                   suffix.equals("jpg"))
                    previewer.configure(f);
            }
        }
    });
}
public static void main(String a[]) {
    JFrame f = new Test();
    f.setBounds(300, 300, 300, 75);
    f.setVisible(true);

    f.setDefaultCloseOperation(
                          WindowConstants.DISPOSE_ON_CLOSE);

    f.addWindowListener(new WindowAdapter() {
        public void windowClosed(WindowEvent e) {
```

```
                System.exit(0);
            }
        });
    }
}
class ImagePreviewer extends JLabel {
    public void configure(File f) {
        Dimension size = getSize();
        Insets insets = getInsets();
        ImageIcon icon = new ImageIcon(f.getPath());

        setIcon(new ImageIcon(icon.getImage().getScaledInstance(
                    size.width - insets.left - insets.right,
                    size.height - insets.top - insets.bottom,
                    Image.SCALE_SMOOTH)));
    }
}
```

Filtering File Types

Files that fit a certain criteria, such as files that end with "`.gif`", can be filtered out of a file chooser with a file filter. Any number of filters can be associated with a single file chooser, but only one of the filters can be active at any given time.

By default, file choosers are fitted with a single “accept all” filter, that, as its name suggests, accepts all files. The default filter can be replaced with a custom filter by invocation of `JFileChooser.setFileFilter()`, which replaces the current filter.

File filters are implemented by extension of the abstract `swing.filechooser.FileFilter` class, which is summarized in Class Summary 15-5.

Class Summary 15-5 fileFilter

Extends: java.lang.Object

Constructors

public FileFilter()

The no-argument constructor is compiler generated.

Methods

public abstract boolean accept(File)
public abstract String getDescription()

The `accept` method returns a `boolean` value indicating whether a particular file is accepted by a filter. The `getDescription` method returns a short description of the types of files that a filter accepts. The description is displayed in the file filter combo box.[5]

The file chooser shown in Figure 16-5 has a text file filter that accepts only files ending in "`.txt`". The file chooser is also fitted with a text previewer, which is listed in Example 16-4 but not discussed.

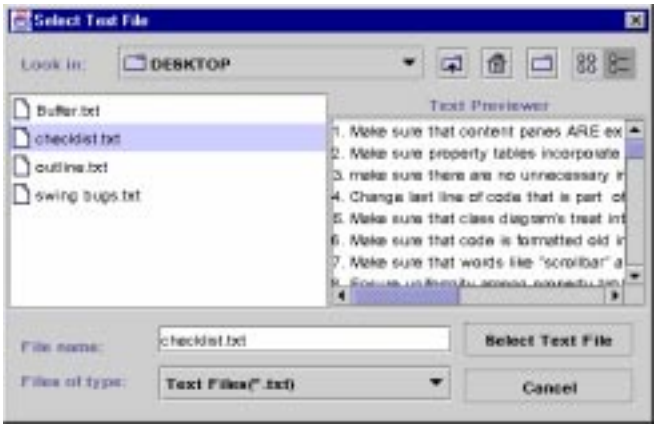


Figure 16-5 A Text File Filter and Previewer

5. How descriptions are displayed is look-and-feel dependent.

The file chooser shown in Figure 16-5 is created by the application listed below, which sets the file filter for the chooser with the `JFileChooser.setFileFilter` method.

```
public class Test extends JFrame {
    JFileChooser   chooser = new JFileChooser();
    TextPreviewer  previewer = new TextPreviewer();
    PreviewPanel   previewPanel = new PreviewPanel();

    ...
    public Test() {
        super("Filtering Files");

        Container contentPane = getContentPane();
        JButtonbutton = new JButton("Select A File");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        chooser.setAccessory(previewPanel);
        chooser.setFileFilter(new TextFilter());

    ...
    }
}
```

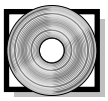
The `TextFilter` class extends `FileFilter` and accepts only files that end in "`.txt`". The description is used in the file chooser's file filter combo box, as can be seen from Figure 16-5.

```
class TextFilter
              extends javax.swing.filechooser.FileFilter {
     public boolean accept(File f) {
        boolean accept = f.isDirectory();

        if( ! accept) {
            String suffix = getSuffix(f);

            if(suffix != null)
                accept = suffix.equals("txt");
        }
        return accept;
    }
    public String getDescription() {
        return "Text Files(*.txt)";
    }
    private String getSuffix(File f) {
        String s = f.getPath(), suffix = null;
```


```
        int i = s.lastIndexOf('.');

        if(i > 0 &&  i < s.length() - 1)
        suffix = s.substring(i+1).toLowerCase();

       return suffix;
    }
}
```

The application shown in Figure 16-5 is listed in its entirety in Example 16-4.

Example 16-4 A Text File Filter and Previewer



```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.URL;

public class Test extends JFrame {
    JFileChooser   chooser = new JFileChooser();
    TextPreviewer  previewer = new TextPreviewer();
    PreviewPanel   previewPanel = new PreviewPanel();

    class PreviewPanel extends JPanel {
        public PreviewPanel() {
            JLabel label = new JLabel("Text Previewer",
                                      SwingConstants.CENTER);
            setPreferredSize(new Dimension(350,0));
            setBorder(BorderFactory.createEtchedBorder());

            setLayout(new BorderLayout());

            label.setBorder(BorderFactory.createEtchedBorder());
            add(label, BorderLayout.NORTH);
            add(previewer, BorderLayout.CENTER);
        }
    }
    public Test() {
        super("Filtering Files");

        Container contentPane = getContentPane();
        JButtonbutton = new JButton("Select A File");

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                int state = chooser.showOpenDialog(null);
                File file = chooser.getSelectedFile();
                String s = "CANCELED";

                if(file != null &&
                   state == JFileChooser.APPROVE_OPTION) {
                    s = "File Selected: " + file.getPath();
                }
                JOptionPane.showMessageDialog(null, s);
            }

        });

        chooser.setAccessory(previewPanel);
        chooser.setFileFilter(new TextFilter());

        chooser.addPropertyChangeListener(
                                new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent e) {
                if(e.getPropertyName().equals(
                   JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
                    previewer.configure((File)e.getNewValue());
                }
            }
        });
    }
    public static void main(String a[]) {
        JFrame f = new Test();
        f.setBounds(300, 300, 300, 75);
        f.setVisible(true);

        f.setDefaultCloseOperation(
                            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
class TextFilter
             extends javax.swing.filechooser.FileFilter {
     public boolean accept(File f) {
        boolean accept = f.isDirectory();

        if( ! accept) {
            String suffix = getSuffix(f);

            if(suffix != null)
                accept = suffix.equals(“txt”);
        }
```

```
        return accept;
    }
    public String getDescription() {
        return "Text Files(*.txt)";
    }
    private String getSuffix(File f) {
        String s = f.getPath(), suffix = null;
         int i = s.lastIndexOf('.');

         if(i > 0 &&  i < s.length() - 1)
         suffix = s.substring(i+1).toLowerCase();

        return suffix;
    }
}
class TextPreviewer extends JComponent {
    private JTextArea textArea = new JTextArea();
    private JScrollPane scrollPane = new JScrollPane(textArea);

    public TextPreviewer() {
        textArea.setEditable(false);

        setBorder(BorderFactory.createEtchedBorder());
        setLayout(new BorderLayout());
        add(scrollPane, BorderLayout.CENTER);
    }
    public void configure(File file) {
        textArea.setText(contentsOfFile(file));

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JViewport vp = scrollPane.getViewport();

                vp.setViewPosition(new Point(0,0));
            }
        });
    }
    public static String contentsOfFile(File file) {
        String s = new String();
        char[] buff = new char[50000];
        InputStream is;
        InputStreamReader reader;
        URL url;

        try {
            reader = new FileReader(file);

            int nch;

            while ((
                nch = reader.read(buff, 0, buff.length)) != -1) {
                s = s + new String(buff, 0, nch);
```

```
            }
        }
        catch (java.io.IOException ex) {
            s = "Could not load file";
        }
        return s;
    }
}
```

Choosable Filters

As illustrated in "Filtering File Types" on page 936, `JFileChooser.setFileFilter()` replaces the default file filter with the filter that is passed to the method. If it is desirable to equip a file chooser with multiple filters, the `JFileChooser.addChoosableFilter` method can be used to add a filter to the current list of filters. The current list of filters can be accessed, and filters can be removed with the `JFileChooser` methods `getChoosableFilters()` and `removeChoosableFilter()`, respectively.

Figure 16-6 shows a file chooser with three filters, one for text files, another for Java source code files, and the default filter that accepts all files. The file chooser is also equipped with a text previewer, which is listed in Example 16-5 but not discussed.[6]

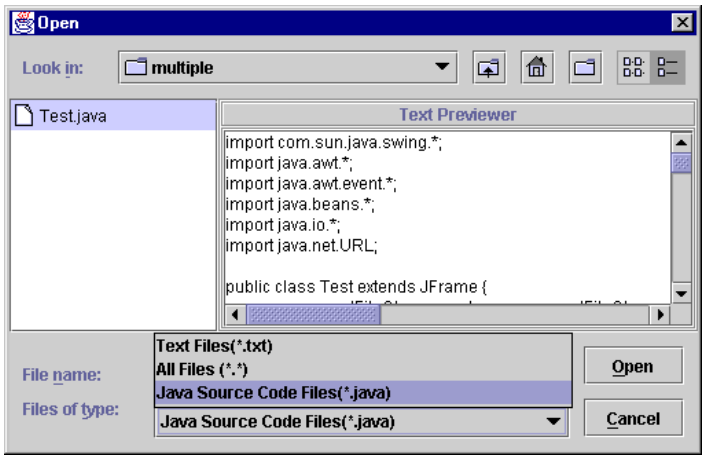


Figure 16-6 Multiple File Filters

6. See "Accessory Components" on page 930 for a discussion of accessory components.

The application that shows the file chooser in Figure 16-6 creates an instance of `JFileChooser` and invokes `JFileChooser.addChoosableFileFilter` to add the text and Java code filters.

```
public Test() {
    JFileChooser   chooser = new JFileChooser();
    TextPreviewer  previewer = new TextPreviewer();
    PreviewPanel   previewPanel = new PreviewPanel();

    super("Filtering Files");

    Container contentPane = getContentPane();
    JButtonbutton = new JButton("Select A File");

    contentPane.setLayout(new FlowLayout());
    contentPane.add(button);

    chooser.setAccessory(previewPanel);
    chooser.addChoosableFileFilter(new TextFilter());
    chooser.addChoosableFileFilter(new JavaCodeFilter());
    ...
}
```

Because both filters determine the acceptability of files according to file name suffix, the filters extend a filter that is suffix aware. The `SuffixAwareFilter` is an abstract filter that accepts all directories and provides a method that returns a file name suffix, given a file. The `SuffixAwareFilter` is abstract because it leaves the `getDescription` method for subclasses to implement.

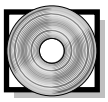
```
...
abstract class SuffixAwareFilter
          extends javax.swing.filechooser.FileFilter {
public String getSuffix(File f) {
    String s = f.getPath(), suffix = null;
     int i = s.lastIndexOf('.');

     if(i > 0 &&  i < s.length() - 1)
     suffix = s.substring(i+1).toLowerCase();

    return suffix;
}
public boolean accept(File f) {
    return f.isDirectory();
}
}
...
```

Both `TextFilter` and `JavaCodeFilter` will accept a file if it is accepted by the `SuffixAwareFilter` superclass (which means the file is a directory) or the file's suffix matches "`.txt`" or "`.java`", respectively.

```
...
class JavaCodeFilter extends SuffixAwareFilter {
    public boolean accept(File f) {
       boolean accept = super.accept(f);

       if( ! accept) {
           String suffix = getSuffix(f);

           if(suffix != null)
              accept = super.accept(f) || suffix.equals("java");
       }
       return accept;
    }
    public String getDescription() {
       return "Java Source Code Files(*.java)";
    }
}
class TextFilter extends SuffixAwareFilter {
    public boolean accept(File f) {
       String suffix = getSuffix(f);

       if(suffix != null)
           return super.accept(f) || suffix.equals("txt");

       return false;
    }
    public String getDescription() {
       return "Text Files(*.txt)";
    }
}
```

The application shown in Figure 16-6 is listed in its entirety in Example 16-5.

Example 16-5 Multiple File Filters



```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.URL;

public class Test extends JFrame {
    JFileChooser   chooser = new JFileChooser();
    TextPreviewer  previewer = new TextPreviewer();
```

```
PreviewPanel   previewPanel = new PreviewPanel();

class PreviewPanel extends JPanel {
    public PreviewPanel() {
        JLabel label = new JLabel("Text Previewer",
                                  SwingConstants.CENTER);
        setPreferredSize(new Dimension(350,0));
        setBorder(BorderFactory.createEtchedBorder());

        setLayout(new BorderLayout());

        label.setBorder(BorderFactory.createEtchedBorder());
        add(label, BorderLayout.NORTH);
        add(previewer, BorderLayout.CENTER);
    }
}
public Test() {
    super("Filtering Files");

    Container contentPane = getContentPane();
    JButtonbutton = new JButton("Select A File");

    contentPane.setLayout(new FlowLayout());
    contentPane.add(button);

    chooser.setAccessory(previewPanel);
    chooser.addChoosableFileFilter(new TextFilter());
    chooser.addChoosableFileFilter(new JavaCodeFilter());

    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            int state = chooser.showOpenDialog(null);
            String s = "CANCELED";

            if(state == JFileChooser.APPROVE_OPTION) {
                s = "File Selected: " +
                     chooser.getSelectedFile().getPath();
            }
            JOptionPane.showMessageDialog(null, s);
        }
    });

    chooser.addPropertyChangeListener(
                           new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent e) {
            if(e.getPropertyName().equals(
               JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)) {
               previewer.configure((File)e.getNewValue());
            }
        }
    });
}
public static void main(String a[]) {
```

```
        JFrame f = new Test();
        f.setBounds(300, 300, 300, 75);
        f.setVisible(true);

        f.setDefaultCloseOperation(
                            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
abstract class SuffixAwareFilter
             extends javax.swing.filechooser.FileFilter {
    public String getSuffix(File f) {
        String s = f.getPath(), suffix = null;
         int i = s.lastIndexOf('.');

         if(i > 0 &&  i < s.length() - 1)
         suffix = s.substring(i+1).toLowerCase();

        return suffix;
    }
    public boolean accept(File f) {
        return f.isDirectory();
    }
}
class JavaCodeFilter extends SuffixAwareFilter {
     public boolean accept(File f) {
        boolean accept = super.accept(f);

        if( ! accept) {
            String suffix = getSuffix(f);

            if(suffix != null)
                accept = super.accept(f) || suffix.equals("java");
        }
        return accept;
    }
    public String getDescription() {
        return "Java Source Code Files(*.java)";
    }
}
class TextFilter extends SuffixAwareFilter {
     public boolean accept(File f) {
        String suffix = getSuffix(f);

        if(suffix != null)
            return super.accept(f) || suffix.equals("txt");
```

```
        return false;
    }
    public String getDescription() {
        return "Text Files(*.txt)";
    }
}
class TextPreviewer extends JComponent {
    private JTextArea textArea = new JTextArea();
    private JScrollPane scrollPane = new JScrollPane(textArea);

    public TextPreviewer() {
        textArea.setEditable(false);

        setBorder(BorderFactory.createEtchedBorder());
        setLayout(new BorderLayout());
        add(scrollPane, BorderLayout.CENTER);
    }
    public void configure(File file) {
        textArea.setText(contentsOfFile(file));

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JViewport vp = scrollPane.getViewport();

                vp.setViewPosition(new Point(0,0));
            }
        });
    }
    static String contentsOfFile(File file) {
        String s = new String();
        char[] buff = new char[50000];
        InputStream is;
        InputStreamReader reader;
        URL url;

        try {
            reader = new FileReader(file);

            int nch;

            while ((
                nch = reader.read(buff, 0, buff.length)) != -1) {
                s = s + new String(buff, 0, nch);
            }
        }
        catch (java.io.IOException ex) {
            s = "Could not load file";
        }
        return s;
    }
}
```

Swing Tip ...

Choosable File Filters

File filters for file choosers can be accessed by the following methods:

- void setFileFilter(swing.filechooser.FileFilter)
- void addChoosableFileFilter(swing.filechooser.FileFilter)
- void removeChoosableFileFilter(swing.filechooser.FileFilter)
- swing.filechooser.FileFilter getChoosableFileFilters()

The use of the word choosable is unfortunate,[1] because it implies that the filter set by setFileFilter() is somehow not choosable. In actuality, all of the methods listed above pertain to a file chooser's choosable file filters.

1. Or the omission of the word in `setFileFilter`, depending upon your perspective

File Views

File choosers obtain the icons and file names they display from an object that is an extension of the abstract `swing.filechooser.FileView` class. The default file view is implemented by the `BasicFileChooserUI.BasicFileView` class, but it can be wholly or partially replaced if a file view is explicitly set with the `JFileChooser.setFileView` method.

If a file chooser is explicitly fitted with a file view that returns `null` from any of its methods, the default file view is queried for the information. For example, if a file chooser has a file view that returns `null` from the `getName` method, the name is obtained from an instance of `BasicFileView`. Therefore, default file view behavior can be partially overridden.

The `swing.filechooser.FileView` class is summarized in Class Summary 15-6.

Class Summary 15-6 FileView

Extends: java.lang.Object

Constructors

public FileView()

The `FileView` constructor is compiler generated, because no constructors are implemented by the `FileView` class.

Methods

Icon / Name / Is Traversable

public abstract Icon getIcon(File)
public abstract String getName(File)
public abstract Boolean isTraversable(File)

The first two methods listed above return an icon and a string respectively, given a file. The icon and string represent files in a file chooser.

The `isTraversable` method indicates whether a file (typically a directory) can be opened. For example, implementing an extension of `FileView` that returns `new Boolean(false)` from `isTraversable` will not allow a directory to be opened; instead, double-clicking on a directory will select the directory instead of opening it.

It should be pointed out that `isTraversable` returns a `Boolean` object and not a `boolean` value because returning `null` from `FileView` methods causes the associated file chooser to get the corresponding value from the default file view.[7]

public abstract String getDescription(File)
public abstract String getTypeDescription(File)

The `getDescription` and `getTypeDescription` methods return a description of a specific file and a description of a file type, respectively. For example, for a file named `FileChooserExample.java`, the `getDescription` method might return "an example of using file choosers" and the `getTypeDescription` might return "a Java source code file."

As of Swing1.1 FCS, the `getDescription` and `getTypeDescription` methods are not used within Swing. The methods are meant for look and feels that wish to provide additional information about files in a file chooser.

The file chooser shown in Figure 16-7 is fitted with a custom extension of the `FileView` class that provides alternative icons for files and directories. Files ending in "`.gif`", "`.bmp`," and "`.jpg`" are represented by a different icon than the one used for other types of files.

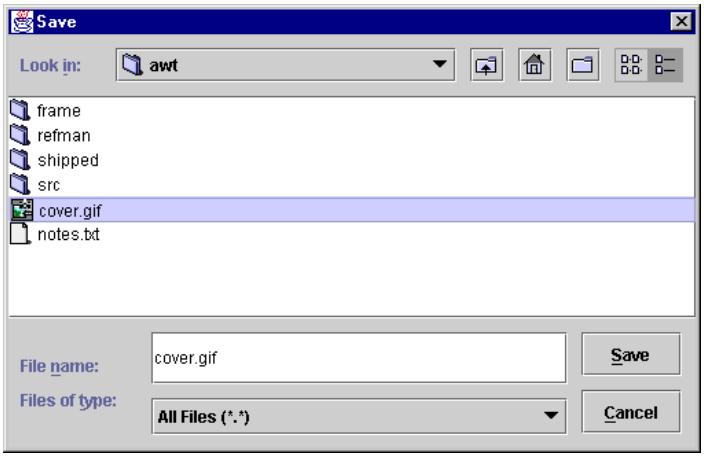


Figure 16-7 A Custom File View

7. `boolean` is an intrinsic type and therefore not an object.

The application that shows the file chooser shown in Figure 16-7 creates an instance of `CustomFileView` that is subsequently specified as the file chooser's file view.

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Custom File View Example");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        chooser.setFileView(new CustomFileView());
        ...
    }
    ...
}
```

The `CustomFileView` class implements the `getName`, `getDescription`, and `getTypeDescription` to return `null`, which causes the values to be obtained from the file chooser's look and feel.

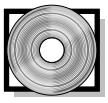
The `CustomFileView` method implements the `getIcon` and `isTraversable` methods. `CustomFileView.getIcon()` returns an icon that is appropriate for the type of file, and `isTraversable` method returns `true` except for a file—`D:\file.txt`—and a directory, `D:\books`.

```
class CustomFileView extends FileView {
    private Icon fileIcon = new ImageIcon("file.gif"),
                 directoryIcon = new ImageIcon("folder.gif"),
                 imageIcon = new ImageIcon("photo.jpg");

    public String getName(File f) { return null; }
    public String getDescription(File f) { return null; }
    public String getTypeDescription(File f) { return null; }

    public Icon getIcon(File f) {
        Icon icon = null;

        if(isImage(f))   icon = imageIcon;
        else if(f.isDirectory()) icon = directoryIcon;
        else             icon = fileIcon;

        return icon;
```

```
    }
    public Boolean isTraversable(File f) {
        Boolean b;

        if(f.getPath().equals("D:\\file.txt")) {
            b = new Boolean(false);
        }
        else if(f.getPath().equals("D:\\books")) {
            b = new Boolean(false);
        }
        return b == null ? new Boolean(true) : b;
    }
    private boolean isImage(File f) {
        String suffix = getSuffix(f);
        boolean isImage = false;

        if(suffix != null) {
            isImage = suffix.equals(“gif”) ||
                      suffix.equals(“bmp”) ||
                      suffix.equals(“jpg”);
        }
        return isImage;
    }
    private String getSuffix(File file) {
        String filestr = file.getPath(), suffix = null;
        int i = filestr.lastIndexOf(‘.’);

        if(i > 0 && i < filestr.length()) {
            suffix = filestr.substring(i+1).toLowerCase();
        }
        return suffix;
    }
}
```

The application shown in Figure 16-7 is listed in its entirety in Example 16-6.

Example 16-6 A Custom File View



```
import javax.swing.*;
import javax.swing.filechooser.FileView;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton(“show file chooser ...”);

    public Test() {
        super(“Custom File View Example”);
```

```
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        chooser.setFileView(new CustomFileView());

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int state = chooser.showSaveDialog(null);
                File file = chooser.getSelectedFile();
                String s = "CANCELED";

                if(state == JFileChooser.APPROVE_OPTION)
                    s = "File: " + file.getPath();

                JOptionPane.showMessageDialog(null, s);
            }
        });
    }
    public static void main(String args[]) {
        JFrame f = new Test();
        f.setBounds(300,300,350,100);
        f.setVisible(true);

        f.setDefaultCloseOperation(
                            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
class CustomFileView extends FileView {
    private Icon fileIcon = new ImageIcon("file.gif"),
                 directoryIcon = new ImageIcon("folder.gif"),
                 imageIcon = new ImageIcon("photo.jpg");

    public String getName(File f) { return null; }
    public String getDescription(File f) { return null; }
    public String getTypeDescription(File f) { return null; }

    public Icon getIcon(File f) {
        Icon icon = null;

        if(isImage(f))  icon = imageIcon;
        else if(f.isDirectory()) icon = directoryIcon;
        else                icon = fileIcon;

        return icon;
```

```
    }
    public Boolean isTraversable(File f) {
        Boolean b;

        if(f.getPath().equals("D:\\file.txt")) {
            b = new Boolean(false);
        }
        else if(f.getPath().equals("D:\\books")) {
            b = new Boolean(false);
        }
        return b == null ? new Boolean(true) : b;
    }
    private boolean isImage(File f) {
        String suffix = getSuffix(f);
        boolean isImage = false;

        if(suffix != null) {
            isImage = suffix.equals("gif") ||
                      suffix.equals("bmp") ||
                      suffix.equals("jpg");
        }
        return isImage;
    }
    private String getSuffix(File file) {
        String filestr = file.getPath(), suffix = null;
        int i = filestr.lastIndexOf('.');

        if(i > 0 && i < filestr.length()) {
            suffix = filestr.substring(i+1).toLowerCase();
        }
        return suffix;
    }
}
```

Multiple Selection

Instances of `JFileChooser` allow multiple files to be selected; however, for Swing 1.1 FCS, although multiple files can be selected in a file chooser, the files selected cannot be accessed.

The file chooser shown in Figure 16-8 allows multiple selection.

The file chooser shown in Figure 16-8 has multiple selection enabled by a call to `JFileChooser.setMultiSelectionEnabled(true)`. After the file chooser has been dismissed, the selected files are retrieved with `JFileChooser.getSelectedFiles()`, which under Swing 1.1 FCS always returns `null`.

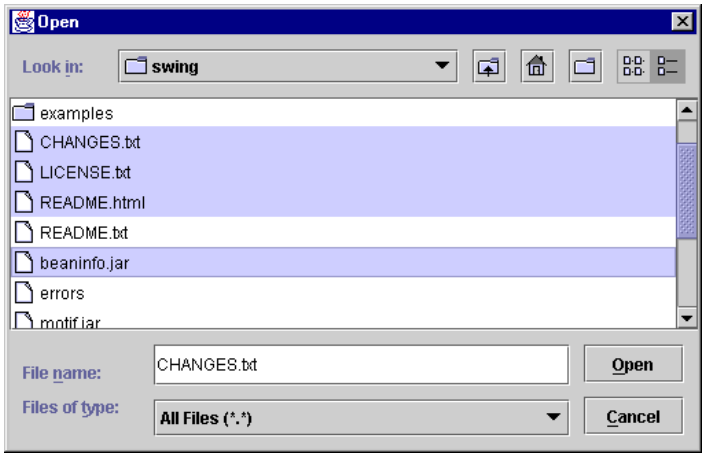


Figure 16-8 Selecting Multiple Files

The applet shown in Figure 16-8 is listed in its entirety in Example 16-7.

Example 16-7 Multiple Selection for File Chooser

```
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import javax.swing.*;
import java.beans.*;

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        chooser.setMultiSelectionEnabled(true);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int state = chooser.showOpenDialog(null);
                File[] files = chooser.getSelectedFiles();
                String[] filenames = getFilenames(files);
```

```
                if(filenames != null &&
                   state == JFileChooser.APPROVE_OPTION) {
                    JOptionPane.showMessageDialog(null,filenames);
                }
                else if(state == JFileChooser.CANCEL_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, "Canceled");
                }
                else if(state == JFileChooser.ERROR_OPTION) {
                    JOptionPane.showMessageDialog(
                                        null, "Error!");
                }
            }
        });
    }
    private String[] getFilenames(File[] files) {
        String[] filenames = null;
        int numFiles = files.length;

        if(files.length > 0) {
            filenames = new String[numFiles];

            for(int i=0; i < numFiles; ++i) {
                filenames[i] = files[i].getPath();
                System.out.println(filenames[i]);
            }
        }
        return filenames;
    }
    public static void main(String args[]) {
        JFrame f = new Test();
        f.setBounds(300,300,350,100);
        f.setVisible(true);

        f.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

The `JFileChooser` class is summarized in Component Summary 16-1.

Component Summary 16-1 JFileChooser

Model(s)	——
UI Delegate(s)	javax.swing.plaf.basic.BasicFileChooserUI
Renderer(s)	——
Editor(s)	——
Events Fired	PropertyChangeEvents, ActionEvents
Replacement For	java.awt.FileDialog

Class Diagrams

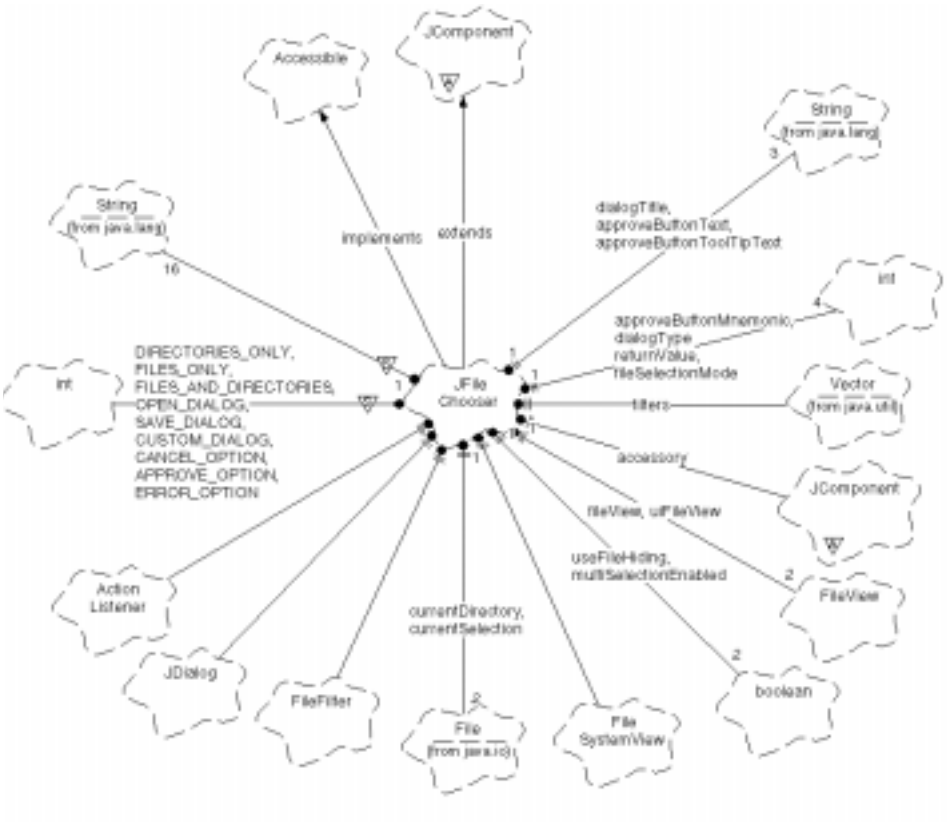


Figure 16-9 JFileChooser Class Diagram

`JFileChooser` extends `JComponent` and implements the `Accessible` interface. `JFileChooser` maintains `private` references to its accessory component, file filters, and file views. `JFileChooser` maintains private references to objects that represent `JFileChooser` properties, which are discussed in "JFileChooser Properties" on page 958.

`JFileChooser` also maintains `public static integer` values representing the three file chooser modes: `DIRECTORIES_ONLY`, `FILES_ONLY`, and `FILES_AND_DIRECTORIES`, in addition to dialog types and selection options. Additionally, `JFileChooser` maintains `public static` strings representing property names, open and save dialogs, etc. See "JFileChooser Class Summaries" on page 968 for more information concerning the `public` string constants defined by the `JFileChooser` class.

JFileChooser Properties

The properties maintained by the `JFileChooser` class are listed in Table 16-2.

Table 16-2 JFileChooser Properties

Property Name	Data Type	Property Type[1]	Access[2]	Default[3]
acceptAllFileFilter	FileFilter	S	G	L&F
accessory	JComponent	B	SG	null
approveButton-Mnemonic	int	B	SG	0
approveButton-Text	String	B	SG	——
approveButton-ToolTipText	String	B	SG	null
choosableFile-Filters	FileFilter[]	B	G	see discussion below
currentDirectory	File	B	CSG	——
description	String	S	G	——
dialogTitle	String	B	SG	L&F
dialogType	int	B	SG	OPEN_DIALOG
fileFilter		B	SG	L&F
fileHidingEnabled	boolean	B	SG	true
fileSelectionMode	int	B	SG	FILES_ONLY
fileSystemView	FileSystemView	B	CSG	null

Table 16-2 JFileChooser Properties (Continued)

Property Name	Data Type	Property Type[1]	Access[2]	Default[3]
fileView	FileView	B	SG	null
icon	Icon	S	G	L&F
multiSelection-Enabled	boolean	B	SG	false
name	String	S	G	——
selectedFile	File	B	SG	null
selectedFiles	File[]	B	SG	null
traversable	boolean	S	G	——
typeDescription	String	S	G	——

1. B = bound (fires PropertyChangeEvent) / C = constrained Of/ I = indexed / S = simple / Ch = fires ChangeEvent
2. C = settable at construction time / G = getter method / S = setter method
3. L&F = look-and-feel dependent

`acceptAllFileFilter` — Represents the default file filter that accepts all files. The property is maintained by the file chooser's UI delegate, making it look-and-feel dependent. It also is read-only and is accessible with the `JFileChooser.getAcceptAllFilter` method.

See "Filtering File Types" on page 936 for more information concerning the `acceptAllFileFilter` property and file filters in general.

`accessory` — An instance of `JComponent` that can be used for multiple purposes, the most common of which are previewers for different file types. See "Accessory Components" on page 930 for more information concerning file chooser accessory components.

`approveButtonMnemonic` — Represents the mnemonic associated with a file chooser's approve button. The mnemonic is set with the `JFileChooser.setApproveButtonMnemonic` method. See "Button Mnemonics" on page 415 for more information concerning button mnemonics in general.

`approveButtonText` — File choosers are equipped with two buttons, one for approving the selected file and another for canceling the file chooser dialog. The text associated with the approve button can be explicitly set.

Setting the approve button text sets the `dialogType` property to `CUSTOM_DIALOG`. Calling `JFileChooser.showOpenDialog()` or

`JFileChooser.showSaveDialog()` sets the approve button text to a look-and-feel-specific string.

`approveButtonToolTipText` — A file chooser's approve button can have a tooltip associated with it. The default tooltip is look-and-feel dependent and can be overridden with the `JFileChooser.setApproveButtonText` method.

`choosableFileFilters` — A file chooser can have any number of choosable file filters associated with it. Choosable file filters can be added to and removed from a file chooser with the `JFileChooser addChoosableFileFilter` and `removeChoosableFileFilter` methods, respectively.

The set of choosable file filters always includes the "accept all" file filter. See the `acceptAllFilter` property for more information concerning the "accept all" file filter.

`currentDirectory` — An instance of `java.io.File` that represents the directory currently displayed in a file chooser. The `currentDirectory` property can be set at construction time or any time thereafter.

If the `currentDirectory` property is specified as a file instead of a directory, the current directory is set to the first traversable parent directory of the file specified.

`dialogTitle` — Is look-and-feel dependent and represents the title of the dialog in which a file chooser is displayed when the following `JFileChooser` methods are used: `showOpenDialog()`, `showSaveDialog()`, and `showDialog()`.

The `dialogTitle` property has no effect if a file chooser is manually placed in a dialog. See Example 16-8 on page 967 for an example of manually placing a file chooser in a dialog.

`dialogType` — The `dialogType` property can be assigned one of the following `JFileChooser` constants:

- `JFileChooser.OPEN_DIALOG`
- `JFileChooser.SAVE_DIALOG`
- `JFileChooser.CUSTOM_DIALOG`

The `dialogType` property sets a file chooser's approve button text if the property is set to either `OPEN_DIALOG` or `CLOSE_DIALOG`. See the `dialogType` property for more information concerning the constants listed above.

`fileFilter` — File filters for a file chooser can be set in one of two ways: with `JFileChooser.addChoosableFileFilter()` or with `JFileChooser.setFileFilter()`, both of which are passed an extension of the abstract `swing.filechooser.FileFilter` class.

`JFileChooser.setFileFilter()` replaces the current file filter(s), whereas `addChoosableFileFilter` adds a filter to the current list of file filters.

`fileHidingEnabled` — A `boolean` value that determines whether or not hidden files are displayed in `JFileChooser`.

`fileSelectionMode` — The `fileSelectionMode` property can be set to one of the following values:

- `JFileChooser.FILES_ONLY`
- `JFileChooser.FILES_AND_DIRECTORIES`
- `JFileChooser.DIRECTORIES_ONLY`

The `fileSelectionMode` property determines whether files and/or directories are displayed in a file chooser.

`fileSystemView` — An instance of the `swing.filechooser.FileSystemView` class, which is summarized in Class Summary 15-7.

Class Summary 15-7 FileSystemView

Extends: java.lang.Object

Constructors

---

public FileSystemView()

The constructor is compiler generated.

Methods

---

public static FileSystemView getFileSystemView()
public File createFileObject(File, String)

public File createFileObject(String)
public abstract File createNewFolder(File) throws IOException
public File[] getFiles(File, boolean)
public File getHomeDirectory()
public File getParentDirectory(File)
public abstract File[] getRoots()
public abstract boolean isHiddenFile(File)
public abstract boolean isRoot(File)

`FileSystemView` provides platform-specific file information, such as root partitions and file type information, that is not available from the JDK1.1 `java.io.File` API.

The `JFileChooser` `fileSystemView` property will rarely be explicitly set by developers.

`fileView` — An object whose class is an extension of the `abstract swing.filechooser.FileView` class. File views provide information pertaining to the manner in which files are represented in a file chooser. By default, the information provided by a file view is obtained from a file chooser's look and feel; however, a file chooser's file view can be explicitly set.

See "FileView" on page 949 for more information concerning file views.

`multiSelectionEnabled` — File choosers support multiple selection, and the `multiSelectionEnabled` property controls whether a file chooser allows single or multiple selection. By default, file choosers allow single selection.

If a file chooser supports only single selection, the selected file is obtained by invocation of `JFileChooser.getSelectedFile()`—see the `selectedFile` property. If a file chooser supports multiple selection, the selected files are obtained by invocation of `JFileChooser.getSelectedFiles()`—see the `selectedFiles` property.

`selectedFile` — Represents the currently selected file in a file chooser. The selected file can be set either by user manipulation of a file chooser or programmatically with the `JFileChooser.setSelectedFile` method.

`selectedFiles` — If a file chooser has multiple selection enabled—see the `multiSelectionEnabled` property—the selected files are obtained by invocation of `JFileChooser.getSelectedFiles()`.

The selected files in a file chooser can be set programmatically with the `JFileChooser.setSelectedFiles` method, which is passed an array of `java.io.File` instances.

`description`
`name`
`icon`
`typeDescription` — The properties listed above are actually maintained by a file chooser's file view. All of the properties listed above are read-only, and the values are obtained from the file chooser's file view.

JFileChooser Events

The `JFileChooser` class fires two types of events: action events and property change events. Action events are fired by file choosers whenever their Approve or Cancel buttons are activated.

Property change events are fired whenever a file chooser's bound properties are modified—see "JFileChooser Properties" on page 958 for a list of bound `JFileChooser` properties.

Action Events

By default, the dialogs displayed as a result of invoking `showOpenDialog()`, `showSaveDialog()`, or `showDialog()` are modal, and therefore references to the selected file(s) can be obtained with a line of code after the call to one of the methods listed above, as the code fragment below illustrates.

```
// code fragment

int state = chooser.showOpenDialog();  // shows modal dialog
file = chooser.getSelectedFile();  // invoked after dialog
                                   // dialog is dismissed
```

In the code fragment listed above, the second line of code is not executed until the dialog is dismissed, and therefore the file returned from `getSelectedFile()` represents the file selected from the file chooser.

On the other hand, non-modal dialogs do not suspend execution of the thread that displays the dialog. As a result, a reference to selected file(s) for a file chooser displayed in a non-modal dialog cannot be obtained in the same fashion as illustrated in the code fragments above. For example, consider the following code fragment:

```
// code fragment

JDialog dialog = new JDialog(null, title, false);

// add chooser to dialog, set dialog title, etc.

dialog.setVisible(true);           // show non-modal dialog
file = chooser.getSelectedFile();  // dialog has not
                                   // been dismissed
```

In the code fragment listed above, the dialog is non-modal, which means the line of code that invokes `JFileChooser.getSelectedFile()` is executed before the dialog is dismissed. As a result, the file returned from `getSelectedFile()` in the code fragment above will return the last file selected from the file chooser or `null` if it is the first time the chooser has been displayed.

An alternative method for reacting to the dismissal of a file chooser dialog is to register an action listener with the file chooser. When the Approve or Cancel buttons in a file chooser are activated, the file chooser fires an action event. As a result, action listeners can be used to react to selections from—or cancellations of—a file chooser.

The file chooser shown in Figure 16-10 is contained in a non-modal dialog. The application that displays the dialog illustrates reacting to selections made from a file chooser.

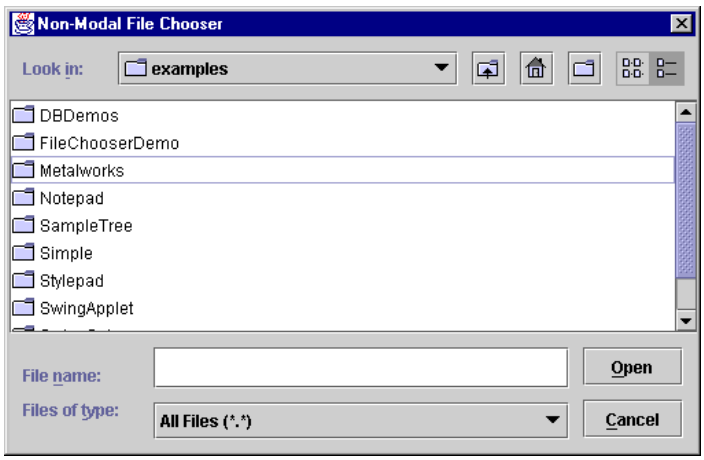


Figure 16-10 JFileChooser Action Events

The application contains a button whose activation results in the display of the dialog shown in Figure 16-10. The application creates a file chooser and a button. The button is added to the applet's content pane, and an action listener is added to the button that creates a non-modal dialog, adds the file chooser to the dialog, and displays the dialog.

The dialog's title is obtained by first invoking `JFileChooser.getDialogTitle()`. If the dialog title has been explicitly set, the call to `JFileChooser.getDialogTitle()` will return the title; otherwise, the method will return `null`. If the dialog title has not been explicitly set, the title is obtained from the chooser's UI delegate.

The dialog is created with a `null` frame, which results in the dialog being centered on the screen when it is displayed. The third argument to the `JDialog` constructor determines the modality of the dialog; in this case, a `false` value results in a non-modal dialog.

```
public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JDialog dialog;
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String title = chooser.getDialogTitle();

                if(title == null)
                    chooser.getUI().getDialogTitle(chooser);

                dialog = new JDialog((Frame)null, title, false);

                Container dialogContentPane =
                            dialog.getContentPane();

                dialogContentPane.setLayout(new BorderLayout());
                dialogContentPane.add(chooser,
                            BorderLayout.CENTER);

                dialog.setTitle("Non-Modal File Chooser");

                dialog.pack();
```

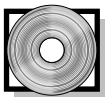
```
            dialog.setLocationRelativeTo(Test.this);

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

An action listener is also added to the chooser itself. The listener uses the action command associated with the action event to determine if a selection was made from the file chooser or if the dialog containing the chooser was canceled. If a selection was made from the file chooser, the selected file is obtained by invocation of `JFileChooser.getSelectedFile`. Subsequently, a message dialog is displayed, and the visibility of the dialog containing the chooser dialog is set to `false`.

```
    ...
    chooser.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String state = (String)e.getActionCommand();
            File file = chooser.getSelectedFile();

            if(file != null &&
                state.equals(JFileChooser.APPROVE_SELECTION)) {
                JOptionPane.showMessageDialog(
                                    null, file.getPath());
            }
            else if(
                state.equals(JFileChooser.CANCEL_SELECTION)) {
                JOptionPane.showMessageDialog(
                                    null, "Canceled");
            }
            // JFileChooser action listeners are notified
            // when either the approve button or
            // cancel button is activated
            dialog.setVisible(false);
        }
    });
}
```

The applet shown in Figure 16-10 is listed in its entirety in Example 16-8.

Example 16-8 JFileChooser Action Events



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

public class Test extends JFrame {
    JFileChooser chooser = new JFileChooser();
    JDialog dialog;
    JButton button = new JButton("show file chooser ...");

    public Test() {
        super("Simple File Chooser Application");
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String title = chooser.getDialogTitle();

                if(title == null)
                    chooser.getUI().getDialogTitle(chooser);

                dialog = new JDialog(null, title, false);

                Container dialogContentPane =
                            dialog.getContentPane();

                dialogContentPane.setLayout(new BorderLayout());
                    dialogContentPane.add(chooser,
                            BorderLayout.CENTER);

                dialog.setTitle("Non-Modal File Chooser");

                dialog.pack();
                dialog.setLocationRelativeTo(Test.this);

                dialog.setVisible(true);
            }
        });
        chooser.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String state = (String)e.getActionCommand();
                File file = chooser.getSelectedFile();
```

```
                if(file != null &&
                    state.equals(JFileChooser.APPROVE_SELECTION)) {
                    JOptionPane.showMessageDialog(
                                        null, file.getPath());
                }
                else if(
                    state.equals(JFileChooser.CANCEL_SELECTION)) {
                    JOptionPane.showMessageDialog(
                                        null, "Canceled");
                }
                // JFileChooser action listeners are notified
                // when either the approve button or
                // cancel button is activated
                dialog.setVisible(false);
            }
        });
    }
    public static void main(String args[]) {
        JFrame f = new Test();
        f.setBounds(300,300,350,100);
        f.setVisible(true);

        f.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);

        f.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

JFileChooser Class Summaries

The `public` and `protected` variables and methods for `JFileChooser` are listed in Class Summary 15-8.

Class Summary 15-8 JFileChooser

Extends: JComponent

Implements: javax.accessibility.Accessible

Constants

public static final String ACCESSORY_CHANGED_PROPERTY

public static final String
APPROVE_BUTTON_MNEMONIC_CHANGED_PROPERTY

public static final String APPROVE_BUTTON_TEXT_CHANGED_PROPERTY
public static final String
APPROVE_BUTTON_TOOL_TIP_TEXT_CHANGED_PROPERTY

public static final String CHOOSABLE_FILE_FILTER_CHANGED_PROPERTY
public static final String DIALOG_TYPE_CHANGED_PROPERTY
public static final String DIRECTORY_CHANGED_PROPERTY
public static final String FILE_FILTER_CHANGED_PROPERTY
public static final String FILE_HIDING_CHANGED_PROPERTY
public static final String FILE_SELECTION_MODE_CHANGED_PROPERTY
public static final String FILE_SYSTEM_VIEW_CHANGED_PROPERTY
public static final String FILE_VIEW_CHANGED_PROPERTY
public static final String MULTI_SELECTION_ENABLED_CHANGED_PROPERTY
public static final String SELECTED_FILE_CHANGED_PROPERTY

The constants listed above identify the properties maintained by the `JFileChooser` class, as in the code fragment listed below:

```
// code fragment

chooser.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        if(e.getPropertyName().equals(
                        JFileChooser.ACCESSORY_CHANGED_PROPERTY)
        // react to accessory change
    }
});
```

Dialog Types

public static final int CUSTOM_DIALOG
public static final int OPEN_DIALOG
public static final int SAVE_DIALOG

The constants listed above are mostly used internally by `JFileChooser` but are also passed to the `JFileChooser.setDialogType` method. If file choosers are displayed with the `JFileChooser` methods `showOpenDialog()`, `showSaveDialog()`, or `showDialog()`, dialog type is automatically set to `OPEN_DIALOG`, `SAVE_DIALOG`, and `CUSTOM_DIALOG`, respectively. If a file chooser is displayed in a dialog without the use of `JFileChooser.show...` methods, the dialog type will default to `OPEN_DIALOG`.

Because dialog type corresponds to a file chooser's approve button text, it may be desirable to explicitly set dialog type with either `OPEN_DIALOG` or `SAVE_DIALOG` when file choosers are displayed by means other than the `JFileChooser.show...` methods. (Setting dialog type to `CUSTOM_DIALOG` does not affect a file chooser's approve button text.)

Modes

---

public static final int DIRECTORIES_ONLY
public static final int FILES_AND_DIRECTORIES
public static final int FILES_ONLY

The constants listed above represent the three modes supported by the `JFileChooser` class. The constants are passed to the `JFileChooser.setFileSelectionMode` method.

Options / Approve-Cancel

---

public static final int APPROVE_OPTION
public static final int CANCEL_OPTION
public static final int ERROR_OPTION

public static final String APPROVE_SELECTION
public static final String CANCEL_SELECTION

The first group of constants listed above are returned from `JFileChooser.showDialog()`, `JFileChooser.showOpenDialog()`, and `JFileChooser.showSaveDialog()`. If the file chooser's approve button is activated, the methods return `APPROVE_OPTION`, whereas `CANCEL_OPTION` is returned in response to an activation of the Cancel button. The `ERROR_OPTION` is reserved for use when an error occurs while a file chooser dialog is displayed. As of Swing1.1, `JFileChooser.ERROR_OPTION` is not used within Swing.

The second group of constants are used as action command strings for action events fired by file choosers as a result of either Approve or Cancel button activation.

Constructors

---

public JFileChooser()
public JFileChooser(FileSystemView)
public JFileChooser(File currentDirectory)
public JFileChooser(File currentDirectory, FileSystemView)
public JFileChooser(String currentDirectoryPath)
public JFileChooser(String currentDirectoryPath, FileSystemView)

protected void setup(FileSystemView)

The no-argument constructor creates a file chooser with the user's home directory as the initially displayed directory.

Files passed to `JFileChooser` constructors determine the initially displayed directory. If the file is not a directory, the initially displayed directory is set to the file's first ancestor directory that is traversable.

`FileSystemView` objects passed to `JFileChooser` constructors are used to obtain file information. See "JFileChooser Properties" on page 958 for a discussion of the `JFileChooser` `fileSystemView` property.

The `protected` `setup` method sets the file system view, updates the UI delegate, and updates the file chooser's UI delegate.

Methods

Accessory Component

---

public JComponent getAccessory()
public void setAccessory(JComponent)

The methods listed above are accessors for a file chooser's accessory component. The `setAccessory` method fires a property change event if the new accessory component is different from the old.

Approve Button

---

public int getApproveButtonMnemonic()
public String getApproveButtonText()
public String getApproveButtonToolTipText()

public void setApproveButtonMnemonic(char)
public void setApproveButtonMnemonic(int)
public void setApproveButtonText(String)
public void setApproveButtonToolTipText(String)

A file chooser's approve button can be customized in three ways: the button's text, its tooltip, and its mnemonic. The methods listed above are accessors for the button's properties.

All of the setter methods listed above fire property change events if the value they are passed is different from the file chooser's current value. The approve button mnemonic can be set with either an `integer` or a `character`.

The `setApproveButtonToolTipText` method sets the dialog type to `JFileChooser.CUSTOM_DIALOG`.

Boolean Properties

public void setFileHidingEnabled(boolean)
public boolean isFileHidingEnabled()

public boolean isDirectorySelectionEnabled()
public boolean isFileSelectionEnabled()

File hiding corresponds to whether hidden files (such as files that begin with '.' under UNIX) are displayed in a file chooser.

The last two methods listed above can be used to determine whether directory or file selection is enabled.

Dialogs

public int showDialog(Component parent, String approveButtonText)
public int showOpenDialog(Component parent)
public int showSaveDialog(Component parent)

public String getDialogTitle()
public int getDialogType()

public void setDialogTitle(String)
public void setDialogType(int)

The first group of methods listed above display a modal dialog containing a file chooser on whose behalf the method is invoked. The `integer` value returned by the methods is one of the following constants:

- `JFileChooser.CANCEL_OPTION`
- `JFileChooser.APPROVE_OPTION`

Dialog title and type are accessed by the last four methods listed above. The `setDialog...` methods take effect only if a file chooser is displayed by one of the `JFileChooser.show...` methods.

Files and Directories

public void changeToParentDirectory()
public void rescanCurrentDirectory()

public File getCurrentDirectory()
public void setCurrentDirectory(File)

The `changeToParentDirectory` method delegates to the file chooser's file system view, and the `rescanCurrentDirectory` delegates to the UI delegate.

The last two methods listed above are accessors for a file chooser's current directory. If a file is passed to `setCurrentDirectory` that is not a directory, the current directory is set to the first traversable parent directory of the specified file.

File Filters

public FileFilter getAcceptAllFileFilter()
public FileFilter getFileFilter()
public void setFileFilter(FileFilter)

public void addChoosableFileFilter(FileFilter)
public boolean removeChoosableFileFilter(FileFilter)

public FileFilter[] getChoosableFileFilters()
public void resetChoosableFileFilters()

By default, file choosers are equipped with a single "accept all" filter that accepts all files. The "accept all" filter can be accessed by the `getAcceptAllFileFilter` method. The `getFileFilter` and `setFileFilter` methods can be used to get and set the currently selected filter.

File choosers can have more than one filter, although only one can be active at any given time. Filters can be added to and removed from a file chooser with the `addChoosableFileFilter` and `removeChoosableFileFilter` methods.

The `getChoosableFileFilters` method returns all of the filters associated with a file chooser, and the `resetChoosableFileFilters` method resets the list of filters to the "accept all" filter.

FileViews

---

public FileView getFileView()
public void setFileView(FileView)

public boolean accept(File)
public String getDescription(File)
public Icon getIcon(File)
public String getName(File)
public String getTypeDescription(File)
public boolean isTraversable(File)

File choosers have a file view object that extends the abstract `swing.filechooser.FileView` class, and `JFileChooser` provides accessors for the file view.

The second group of methods delegate directly to a file chooser's file view.

FileSystemView

---

public FileSystemView getFileSystemView()
public void setFileSystemView(FileSystemView)

File choosers delegate operating-system-dependent functionality to an object that extends the `abstract swing.filechooser.FileSystemView` class. The methods listed above are accessors for a file chooser's file system view.

Listeners

---

public void addActionListener(ActionListener)
public void removeActionListener(ActionListener)

protected void fireActionPerformed(String)

File choosers fire action events when their buttons are activated. As a result, `JFileChooser` provides methods for adding and removing action listeners. The `protected fireActionPerformed` method fires an action event to all registered action listeners.

Programmatic Manipulation

---

public void approveSelection()
public void cancelSelection()
public void ensureFileIsVisible(File)

File choosers can be manipulated programmatically by the methods listed above. Invoking `approveSelection()` mimics the activation of a file chooser's approve button, and the `cancelSelection` method mimics the activation of a file chooser's cancel button. There is no way to distinguish whether a selection or cancellation was initiated programmatically or by a user gesture.

The `ensureFileIsVisible` method scrolls the specified file into view.

Note: `ensureFileIsVisible()` *does not work for all look and feels as of Swing 1.1 FCS.*

File Selection Mode

---

public void setFileSelectionMode(int)
public int getFileSelectionMode()

File choosers support three selection modes, as defined by the `integer` constants listed below:

- `DIRECTORIES_ONLY`
- `FILES_AND_DIRECTORIES`
- `FILES_ONLY`

The constants are passed to the `setFileSelectionMode` method. The current selection mode can be obtained by `getFileSelectionMode()`.

Multiple Selection and Selected Files

---

public void setMultiSelectionEnabled(boolean)
public boolean isMultiSelectionEnabled()

public File getSelectedFile()
public void setSelectedFile(File)

public File[] getSelectedFiles()
public void setSelectedFiles(File[])

File choosers support both single and multiple selection. The first two methods listed above are accessors for the `boolean multiSelectionEnabled` property.

The last four methods listed above are accessors for the selected file(s); the first two methods are used for file choosers with single selection, and the last two methods are used for file choosers with multiple selection.

Note: *Multiple selection is not fully supported in Swing 1.1 FCS.*

Accessibility / Pluggable Look and Feel

---

public AccessibleContext getAccessibleContext()
public FileChooserUI getUI()
public String getUIClassID()
public void updateUI()

The methods listed above can be found in most extensions of `JComponent`. Swing lightweight components can return the class name of their UI delegate and an accessibility context that contains accessibility information for the component. The `updateUI` method is invoked when the component is fitted with a UI delegate.

AWT Compatibility

The main difference between the AWT's `FileDialog` and Swing's `JFileChooser`—other than the fact that Swing file choosers are much more capable—is the fact that the AWT's file dialogs deal in strings, whereas `JFileChooser` deals with files.

Table 16-3 lists the `public` methods implemented by the `java.awt.FileDialog` class and their `JFileChooser` equivalents.

Table 16-3 java.awt.FileDialog Methods & JFileChooser Equivalents

java.awt.FileDialog Methods	JFileChooser Equivalent
String getDirectory()	File getCurrentDirectory()
String getFile()	File getSelectedFile()
FilenameFilter getFilenameFilter()	FileFilter getFileFilter()
int getMode()	int getFileSelectionMode()
void setDirectory(String)	void setCurrentDirectory(File)
void setFile(String)	void setSelectedFile(File)
void setFilenameFilter(FilenameFilter)	void setFileFilter(FileFilter)
void setMode(int)	void setFileSelectionMode(int)

JColorChooser

Color choosers, represented by the `JColorChooser` class, are components that allow a color to be selected. A color chooser is composed of two separate areas: a set of color chooser panels displayed in a tabbed pane[8] and a preview panel that visually communicates the selected color. Both areas can be customized by replacing the default panels.

Like file choosers and option panes, color choosers are typically displayed in a dialog, but since color choosers are components, they can be contained in any AWT or Swing container.

8. there is more than one chooser panel.

`JColorChooser` features are listed in Feature Summary 16-1.

Feature Summary 16-1 JColorChooser

Color Chooser Panels

By default, color choosers come with three panels that allow a color chooser to be selected: `Swatches`, `HSB`, and `RGB`. Any of the default panels can be removed, and custom panels can be added.

Preview Panel

A color chooser's selected color is communicated through a preview panel. The default preview panel can be replaced with a custom version.

Prefabricated Dialogs

`JColorChooser` provides two `static` methods: `createDialog()` and `showDialog()` that create dialogs containing color choosers. The former creates and returns a dialog containing the color chooser it is passed; the latter creates both color chooser and dialog and shows the dialog. Both dialogs are modal.

The applet shown in Figure 16-11 contains a color chooser. By default, color choosers are equipped with three chooser panels contained in a tabbed pane, each of which is shown in Figure 16-11.

The applet shown in Figure 16-11 creates a color chooser with the `JColorChooser` no-argument constructor, and the chooser is added to the applet's content pane.

Color choosers have a selection model that is an implementation of the `swing.colorchooser.ColorSelectionModel` interface. The applet shown in Figure 16-11 adds a change listener to the color chooser's selection model to determine when a color has been selected. The listener responds to selections by updating the applet's status area.

The applet shown in Figure 16-11 is listed in Example 16-9.

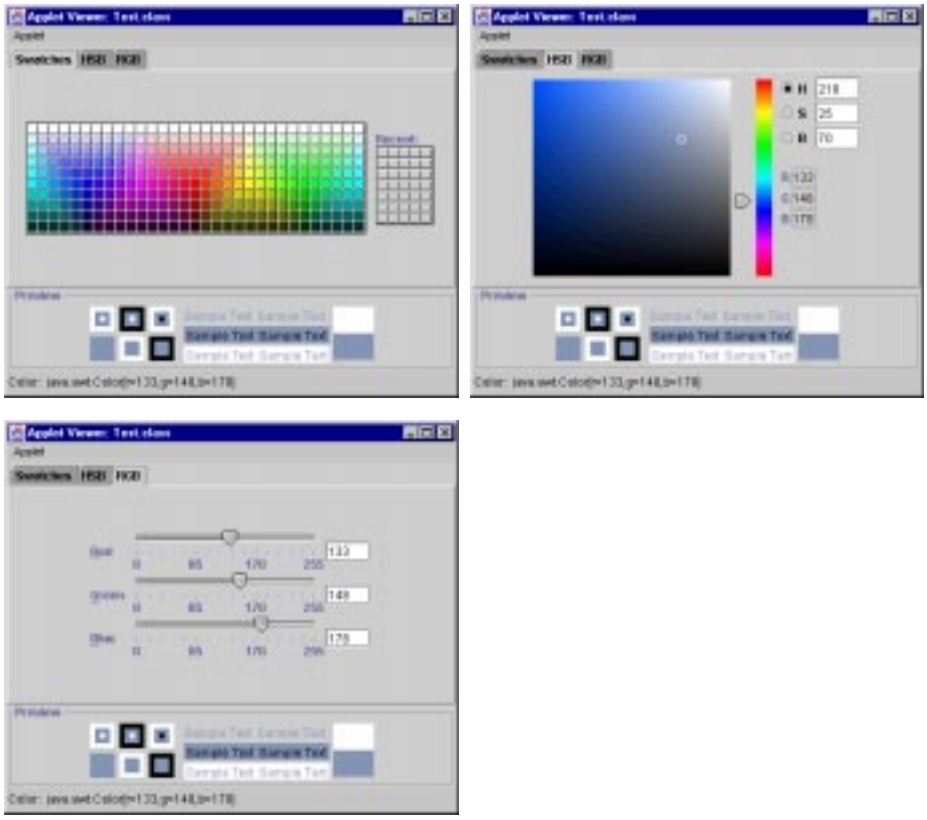
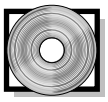


Figure 16-11 A Color Chooser Displayed in an Applet



Example 16-9 A Color Chooser Displayed in an Applet

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

public class Test extends JApplet {
    JColorChooser chooser = new JColorChooser();
    ColorSelectionModel model = chooser.getSelectionModel();
```

```
    public void init() {
        getContentPane().add(chooser, BorderLayout.CENTER);

        model.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                showStatus("Color:  " + chooser.getColor());
            }
        });
    }
}
```

Displaying Color Choosers In Dialogs

Color choosers can be displayed in a dialog in one of two ways. The `static JColorChooser.showDialog` method creates a color chooser that is placed in a newly created dialog every time the method is invoked.

The `JColorChooser.createDialog` method is passed an existing color chooser that is also placed in a newly created dialog every time the method is invoked.

The following sections illustrate the use of the methods described above.

Showing Color Chooser Dialogs

The applet shown in Figure 16-12 contains a grid of `ColorPatch` objects that are contained in a `Palette`; both the `ColorPatch` and `Palette` classes are implemented by the applet. The upper-left picture in Figure 16-12 shows the applet as it appears initially. The upper-right picture shows the color chooser displayed as a result of clicking on the upper-left color patch. The lower-left picture shows a message dialog that the applet displays after the color chooser has been dismissed. The lower-right picture shows the applet after white has been selected from the color chooser. The selected color is set as the background color for the color patch in which the mouse pressed event occurred.

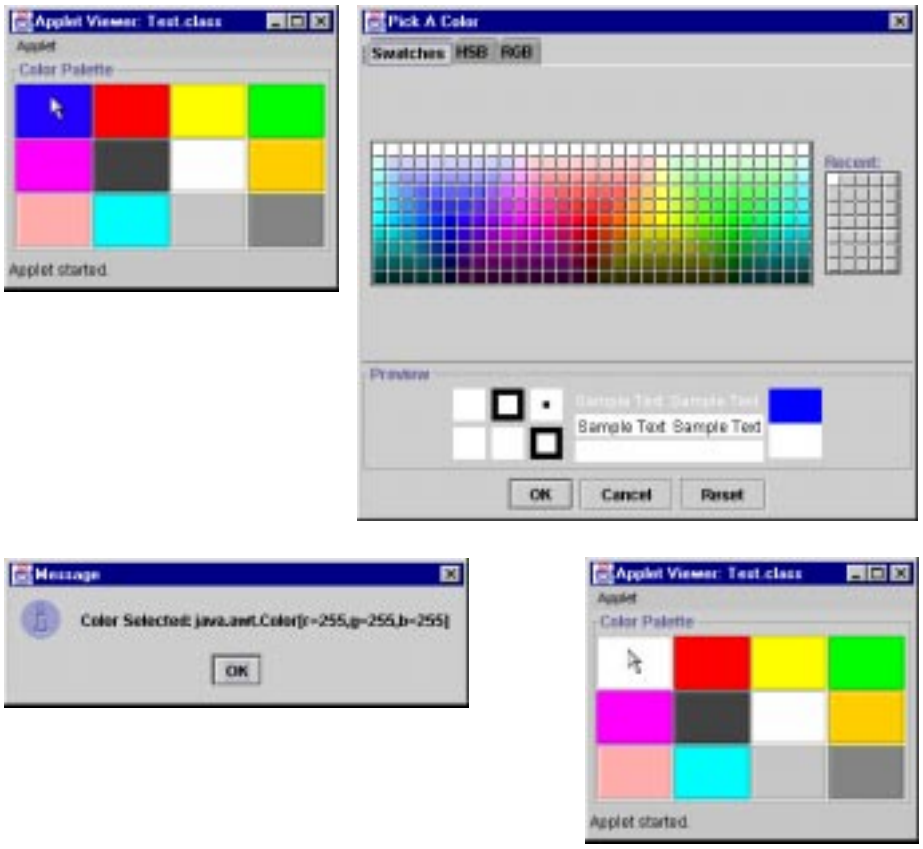


Figure 16-12 Using JFileChooser.showDialog()

The applet creates an instance of `Palette`, which is subsequently added to the applet's content pane.

```
public class Test extends JApplet {
    public void init() {
        getContentPane().add(new Palette(), BorderLayout.CENTER);
    }
}
```

The `Palette` class is a simple extension of `JPanel` that contains a grid of `ColorPatch` objects representing a set of default colors.


```
class Palette extends JPanel {
    private Color[] defaultColors = new Color[] {
        Color.blue, Color.red, Color.yellow, Color.green,
        Color.magenta, Color.darkGray, Color.white, Color.orange,
        Color.pink, Color.cyan, Color.lightGray, Color.gray,
    };

    public Palette() {
        int columns = 3;

        setBorder(
            BorderFactory.createTitledBorder("Color Palette"));

        setLayout(new GridLayout(columns,0,1,1));

        for(int i=0; i < defaultColors.length; ++i)
            add(new ColorPatch(defaultColors[i]));
    }
}
```

`ColorPatch` class is the most interesting class the applet implements. `ColorPatch` is also an extension of `JPanel` with a mouse listener that invokes `JColorChooser.showDialog` when a mouse pressed event is detected.

`JColorChooser.showDialog` creates a color chooser and a modal dialog, adds the chooser to the dialog, displays the dialog, and returns a reference to the color selected from the chooser. If the dialog is canceled—by activating the color chooser's Cancel button, pressing the `Esc` key, or clicking the dialog's close button—the `showDialog` method returns a `null` reference.

The mouse listener for the `ColorPatch` class displays a message dialog with a message corresponding to whether a color was selected or the dialog was dismissed. If a color is selected, the listener sets the background color for the color patch that launched the dialog to the selected color and repaints the color patch.

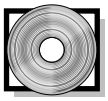
```
class ColorPatch extends JPanel {
    JApplet applet;
    Color selectedColor;

    public ColorPatch(Color color) {
        // add border and set background color ...

        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                selectedColor = JColorChooser.showDialog(
                                ColorPatch.this,  // parent comp
                                "Pick A Color",   // dialog title
                                getBackground()); // initial color
```

```
                if(selectedColor == null) {
                    JOptionPane.showMessageDialog(ColorPatch.this,
                        "ColorChooser Canceled");
                }
                else {
                    setBackground(selectedColor);
                    repaint();

                    JOptionPane.showMessageDialog(ColorPatch.this,
                        "Color Selected: " + selectedColor);
                }
            }
        });
    }
}
```

`JColorChooser.showDialog()` takes three arguments: a component over which the dialog is centered, a title for the dialog, and the color chooser's initial color.

The applet shown in Figure 16-12 is listed in its entirety in Example 16-10.

Example 16-10 A Color Chooser Displayed in a Dialog



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

public class Test extends JApplet {
    public void init() {
        getContentPane().add(new Palette(), BorderLayout.CENTER);
    }
}
class Palette extends JPanel {
    private Color[] defaultColors = new Color[] {
        Color.blue, Color.red, Color.yellow, Color.green,
        Color.magenta, Color.darkGray, Color.white, Color.orange,
        Color.pink, Color.cyan, Color.lightGray, Color.gray,
    };

    public Palette() {
        int columns = 3;

        setBorder(
            BorderFactory.createTitledBorder("Color Palette"));

        setLayout(new GridLayout(columns,0,1,1));

        for(int i=0; i < defaultColors.length; ++i)
```


```
            add(new ColorPatch(defaultColors[i]));
        }
    }
    class ColorPatch extends JPanel {
        JApplet applet;
        Color selectedColor;

        public ColorPatch(Color color) {
            setBorder(BorderFactory.createEtchedBorder());
            setBackground(color);

            addMouseListener(new MouseAdapter() {
                public void mousePressed(MouseEvent e) {
                    selectedColor = JColorChooser.showDialog(
                                    ColorPatch.this,  // parent comp
                                    "Pick A Color",   // dialog title
                                    getBackground()); // initial color

                    if(selectedColor == null) {
                        JOptionPane.showMessageDialog(ColorPatch.this,
                            "ColorChooser Canceled");
                    }
                    else {
                        setBackground(selectedColor);
                        repaint();

                        JOptionPane.showMessageDialog(ColorPatch.this,
                            "Color Selected: " + selectedColor);
                    }
                }
            });
        }
    }
```

Creating Color Chooser Dialogs

As an alternative to `JColorChooser.showDialog()`, the `JColorChooser` class provides a `static` `createDialog` method that creates a dialog, given a color chooser. The `JColorChooser.createDialog` method is passed six arguments:

- A component over which the dialog will be centered
- A string representing dialog's title
- A `boolean` value representing the modality of the dialog
- A color chooser that is placed in the dialog
- Two action listeners that are notified when the color chooser's buttons are activated

The `ColorPatch` class listed below is similar to the one listed in Example 16-10 on page 984, except that `JFileChooser.createDialog()` is used instead of `JFileChooser.showDialog()`. The `ColorPatch` class listed below contains a `static` color chooser that is shared by all instances of the `ColorPatch` class. Color choosers are somewhat expensive to create; as a result, creating a color chooser for each color patch in the palette would result in a perceptible performance penalty.

```
class ColorPatch extends JPanel {
    JApplet applet;
    static JColorChooser chooser = new JColorChooser();
    JDialog dialog;

    public ColorPatch(Color color) {
        setBorder(BorderFactory.createEtchedBorder());
        setBackground(color);

        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if(dialog == null)
                    dialog = JColorChooser.createDialog(
                              ColorPatch.this,  // parent comp
                              "Pick A Color",   // dialog title
                              false,      // modality
                              chooser,
                              new OkListener(),
                              new CancelListener());

                chooser.setColor(getBackground());
                dialog.setVisible(true);
            }
        });
    }
    class OkListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            Color color = chooser.getColor();
            setBackground(color);
            repaint();

            JOptionPane.showMessageDialog(chooser,
                "Color Selected: " + chooser.getColor());
        }
    }
    class CancelListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null,
                "ColorChooser Canceled");

        }
    }
}
```

After the mouse listener associated with the `ColorPatch` class creates the dialog, the chooser's color is set to the background color of the patch and the dialog is made visible.

The action listeners passed to the `JFileChooser.createDialog` method show a message dialog in a similar fashion to the mouse listener associated with the `ColorPatch` class listed in Example 16-10. If the color chooser's button is activated, the selected color is obtained with the `JColorChooser.getColor` method, the background of the color patch is set, and the color patch is repainted.

The applet that uses the `ColorPatch` class listed above is identical to the applet listed in Example 16-10, with the exception of the `ColorPatch` class itself. The applet is not listed; however, the listing is contained on the CD in the back of the book.

Swing Tip ...

Speed vs. Convenience

The easiest way to display a color chooser in a dialog is to use the JColorChooser.showDialog method; however, there is a fairly steep price to pay for the convenience. Every time showDialog() is invoked, a newly created color chooser is displayed in a newly created dialog.

Although it requires a little more effort, manually constructing a color chooser and a dialog can result in a significant performance boost. If the applications discussed in "Showing Color Chooser Dialogs" on page 981 and "Creating Color Chooser Dialogs" on page 985 are run simultaneously, a perceptible performance difference can be seen from the time a mouse click occurs in a color patch and the color chooser dialog is displayed.

Customizing Color Choosers

By default, color choosers are fitted with three color chooser panels contained in a tabbed pane and a preview panel, as illustrated in Figure 16-1 on page 920. Although the default color chooser panels and preview panel are likely to be sufficient for most color choosers, both the color choosers panels and preview panel can be replaced with custom versions, as the following sections illustrate.

Preview Panels

When a color is selected in a color chooser, the default preview panel's foreground color is set to the selected color and the default preview panel is repainted.

A color chooser's preview component can be customized by specifying the preview component `JColorChooser.setPreviewPanel(JComponent)`. However, although the component specified by the `setPreviewPanel` method is substituted for the default preview panel, custom preview panels are not updated when a color is selected due to a bug in `JColorChooser` in Swing 1.1 FCS.

Note: *The following application does not work properly under Swing 1.1 FCS because of the bug cited in the previous paragraph. It should work when a subsequent release of Swing fixes the bug.*

The color chooser shown in Figure 16-13 contains a custom preview panel.

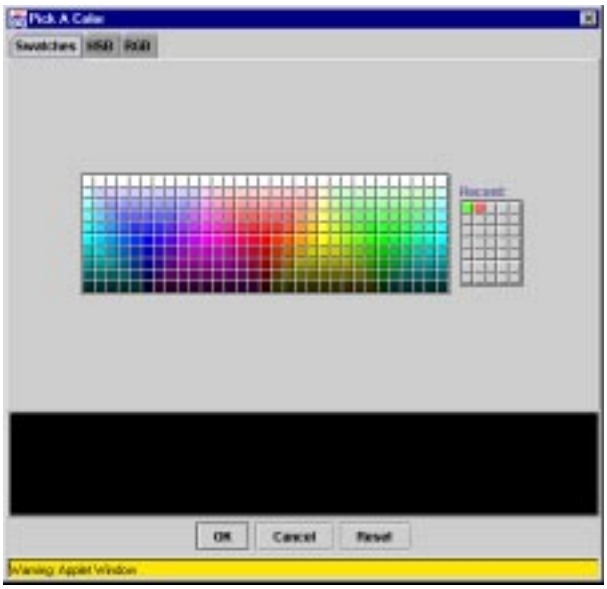
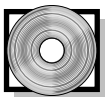


Figure 16-13 A Custom Preview in a Color Chooser

The application that shows the color chooser shown in Figure 16-13 is listed in Example 16-11.

Example 16-11 A Custom Preview in a Color Chooser



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

public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser();
    private JButton button = new JButton("Show Color Chooser");
    private JDialog dialog;

    public void init() {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button, BorderLayout.CENTER);

        chooser.setPreviewPanel(new PreviewPanel());

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {

                if(dialog == null)
                    dialog= JColorChooser.createDialog(
                                Test.this,  // parent comp
                                "Pick A Color",   // dialog title
                                false,      // modality
                                chooser,
                                null, null);

                dialog.setVisible(true);
            }
        });
    }
    class PreviewPanel extends JPanel {
        public PreviewPanel() {
            setPreferredSize(new Dimension(0,100));
            setBorder(BorderFactory.createRaisedBevelBorder());
        }
        public void paintComponent(Graphics g) {
            Dimension size = getSize();

            g.setColor(getForeground());
            g.fillRect(0,0,size.width,size.height);
        }
    }
}
```

The application sets the preview panel for its color chooser by invoking `JColorChooser.setPreviewPanel()`, which is passed an instance of `PreviewPanel`.

The `PreviewPanel` class extends `JPanel` and specifies a preferred height of 100 pixels. The preferred width for color chooser preview panels is ignored, so it is set to 0. The preview panel is fitted with a raised bevel border obtained from the Swing border factory.

When a color is selected in a color chooser, the foreground color of the preview panel is set to the selected color and the preview panel is repainted. The `PreviewPanel` class overrides `paintComponent` to fill the panel with the foreground color. Unfortunately, only the default preview panel is updated when a color is selected; as a result, the application listed in Figure 16-13 does not work as advertised.

Color Chooser Panels

The set of color chooser panels contained in a color chooser can be modified with `JColorChooser.setChooserPanels()` or individual panels can be added or removed with the `addChooserPanel` and `removeChooserPanel` methods. This section discusses the former.

The color chooser panels displayed in a color chooser are objects that extend the `swing.colorchooser.AbstractColorChooserPanel` class, which is summarized in Class Summary 15-9.

Class Summary 15-9 AbstractColorChooserPanel

Extends: JPanel

Constructors

---

public AbstractColorChooserPanel()

The `AbstractColorChooserPanel` class does not implement any constructors, and therefore the no-argument constructor is compiler generated.

Methods

Color/Color Selection Model / Installing Chooser Panel / Painting

---

public ColorSelectionModel getColorSelectionModel()
protected Color getColorFromModel()

public void installChooserPanel(JColorChooser)
public void uninstallChooserPanel(JColorChooser)

public void paint(Graphics)

The first two methods listed above are convenience methods that return the selection model associated with the panel's color chooser, and the color maintained by the selection model, respectively.

The `installChooserPanel` method is invoked when a color chooser is instantiated. The method invokes `buildChooser()` and `updateChooser()` and adds a change listener to the color chooser's selection model. The listener works in concert with the `paint` method so that `updateChooser()` is invoked when the panel is painted if changes have been made to the chooser's selection model before the chooser is displayed.

The `uninstallChooserPanel` method removes the listener that was added to the chooser's selection model by the `installChooserPanel` method.

Building and Updating Chooser

---

protected abstract void buildChooser()
public abstract void updateChooser()

The `buildChooser` method is invoked when a color chooser's chooser panels are set with the `JColorChooser.setChooserPanels` method. The `buildChooser` method is expected to create the components contained in the panel and to add the components to the panel.

The `updateChooser` method is also invoked when color chooser panels are set with the `JColorChooser.setChooserPanels` method. Additionally, `updateChooser()` is invoked when a change is made to the chooser's color selection model.

The `buildChooser` and `updateChooser` methods are abstract and therefore must be implemented by concrete extensions of `AbstractColorChooserPanel`.

Display Names and Icons

---

public abstract String getDisplayName()
public abstract Icon getLargeDisplayIcon()
public abstract Icon getSmallDisplayIcon()

The three methods listed above return information about a color chooser panel. As of Swing 1.1 FCS / 1.2 JDK, only the `getDisplayName` method is used for the string displayed in the tab associated with a panel. For example, the default display names for the default chooser panels are "Swatches", "HSB", and "RBG", as can be seen from Figure 16-11 on page 980. The icons returned from `getLargeDisplayIcon()` and `getSmallDisplayIcon()` are not currently used within Swing.

The applet shown in Figure 16-14 contains a button whose activation results in a color chooser being displayed in a dialog. The default chooser panels in the color chooser are replaced by an extension of the `AbstractColorChooserPanel` class. It should be noted that the chooser panel contained in the color chooser shown in Figure 16-14 is short on usability and long on simplicity for the sake of illustration.

The applet creates a color chooser and an array of objects that implement the `AbstractColorChooserPanel` class. The array contains a single instance of `ListPanel`, which is implemented by the applet. Notice that the `ListPanel` instance is not contained in a tabbed pane because only one chooser panel is contained in the array.

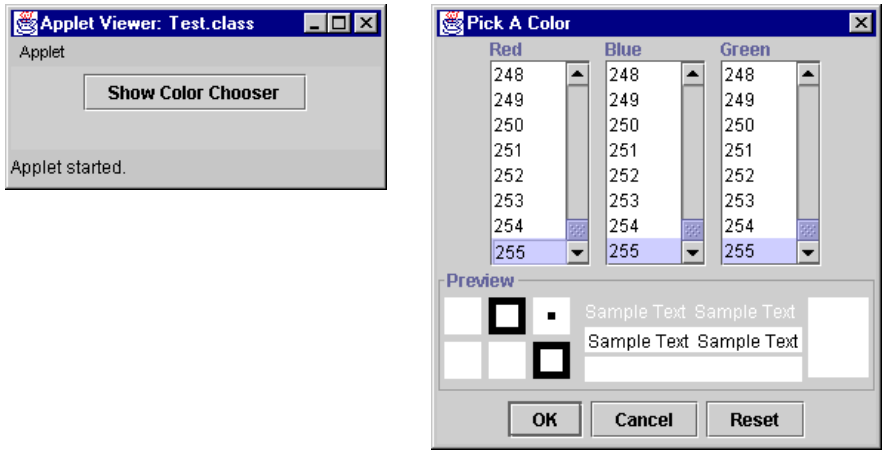


Figure 16-14 Replacing Chooser Panels

The applet's `init` method invokes `JColorChooser.setChooserPanels()`, passing the array of color chooser panels.

```
public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser();
    private AbstractColorChooserPanel colorPanels[] =
           new AbstractColorChooserPanel[] {
               new ListPanel(),
           };
    private JButton button = new JButton("Show Color Chooser");
    private JDialog dialog;

    public void init() {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button, BorderLayout.CENTER);

        chooser.setChooserPanels(colorPanels);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(dialog == null)
                    dialog= JColorChooser.createDialog(
                              Test.this,  // parent comp
                              "Pick A Color",   // dialog title
                              false,      // modality
                              chooser,
```

```
                                null, null);

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

The `ListPanel` class extends abstract `ColorChooserPanel` class and implements the `ListSelectionListener` interface. Two panels are created, one for the "Red", "Green", and "Blue" labels and another for their associated lists. Additionally, three default list models are created for each of the lists, and a `boolean` `isAdjustingValue` is set to `false`.

```
...
class ListPanel extends AbstractColorChooserPanel
                        implements ListSelectionListener {
    private JPanel labelPanel = new JPanel(),
                   listPanel = new JPanel();

    private JList redList = new JList(), blueList = new JList(),
                  greenList = new JList();

    private DefaultListModel redModel = new DefaultListModel(),
                        blueModel = new DefaultListModel(),
                        greenModel = new DefaultListModel();

    private boolean isAdjusting = false;
    ...
```

The `buildChooser` method populates each of the lists with strings representing values from 0 to 256. The lists are added to the list panel, the labels are added to the label panel, and the list and label panels are added to the chooser panel.

Each of the lists has the chooser panel specified as a list selection listener. When a value changes in a list, the chooser panel's `valueChanged` method retrieves the values from the lists and updates the chooser's selection model accordingly.

```
    ...
    protected void buildChooser() {
        redList.setFixedCellWidth(50);
        greenList.setFixedCellWidth(50);
        blueList.setFixedCellWidth(50);
```


```
        for(int i=0; i < 256; ++i) {
            redModel.addElement(Integer.toString(i));
            greenModel.addElement(Integer.toString(i));
            blueModel.addElement(Integer.toString(i));
        }

        redList.setModel(redModel);
        greenList.setModel(greenModel);
        blueList.setModel(blueModel);

        listPanel.setLayout(new GridLayout(0,3,10,0));

        listPanel.add(new JScrollPane(redList));
        listPanel.add(new JScrollPane(blueList));
        listPanel.add(new JScrollPane(greenList));

        labelPanel.setLayout(new GridLayout(0,3,10,0));

        labelPanel.add(new JLabel("Red"));
        labelPanel.add(new JLabel("Blue"));
        labelPanel.add(new JLabel("Green"));

        setLayout(new BorderLayout());
        add(labelPanel, BorderLayout.NORTH);
        add(listPanel, BorderLayout.CENTER);

        redList.addListSelectionListener(this);
        greenList.addListSelectionListener(this);
        blueList.addListSelectionListener(this);
    }
    public void valueChanged(ListSelectionEvent e) {
        int r = redList.getSelectedIndex(),
            b = blueList.getSelectedIndex(),
            g = greenList.getSelectedIndex();

        if(r != -1 && g != -1 && b != -1)
            getColorSelectionModel().setSelectedColor(
                                        new Color(r,g,b));
    }
    ...
```

The `updateChooser` method updates the selected values in the lists, depending upon the current color from the chooser's selection model.

The `isAdjusting` `boolean` variable is used in order to avoid an infinite loop when `JList.setSelectedIndex()` is called. The gory details are enumerated below.

A call to `JList.setSelectedIndex()` causes the list to fire a list selection event, resulting in a call to `ListPanel.valueChanged()`—recall that `ListPanel` adds itself as a listener for each list. `valueChanged()` updates the

chooser's selection model, causing `ListPanel.updateChooser()` to be invoked, and `updateChooser` invokes `JList.setSelectedIndex()`.

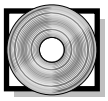
```
...
public void updateChooser() {
    if( ! isAdjusting) {
        isAdjusting = true;

        Color color = getColorFromModel();
        int r = color.getRed(), g = color.getGreen(),
            b = color.getBlue();

        redList.setSelectedIndex(r);
        redList.ensureIndexIsVisible(r);

        blueList.setSelectedIndex(b);
        blueList.ensureIndexIsVisible(b);

        greenList.setSelectedIndex(g);
        greenList.ensureIndexIsVisible(g);

        isAdjusting = false;
    }
}
...
```

The `getDisplayName`, `getSmallDisplayIcon`, and `getLargeDisplayIcon` methods must be implemented by the `ListPanel` class because they are defined as abstract in `AbstractColorChooserPanel`. The `get...Icon` methods return `null` references because the icons are currently not used within Swing. The `getDisplayName` method returns a non-`null` string that would normally be used as the string displayed in the tab associated with the chooser panel. However, because the `ListPanel` instance is the only chooser panel displayed in the color chooser, the string returned from `getDisplayName` is not used.

```
    ...
    public String getDisplayName() {
        return "lists";
    }
     public Icon getSmallDisplayIcon() {
        return null;
    }
     public Icon getLargeDisplayIcon() {
        return null;
    }
}
```

The applet shown in Figure 16-14 is listed in its entirety in Example 16-12.

Example 16-12 Implementing a Custom Color Chooser Panel



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

public class Test extends JApplet {
    private JColorChooser chooser = new JColorChooser();
    private AbstractColorChooserPanel colorPanels[] =
           new AbstractColorChooserPanel[] {
              new ListPanel(),
           };
    private JButton button = new JButton("Show Color Chooser");
    private JDialog dialog;

    public void init() {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(button, BorderLayout.CENTER);

        chooser.setChooserPanels(colorPanels);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(dialog == null)
                    dialog= JColorChooser.createDialog(
                              Test.this,  // parent comp
                              "Pick A Color",   // dialog title
                              false,      // modality
                              chooser,
                              null, null);

                dialog.setVisible(true);
            }
        });
    }
}
class ListPanel extends AbstractColorChooserPanel
              implements ListSelectionListener {
    private JPanel labelPanel = new JPanel(),
                   listPanel = new JPanel();

    private JList redList = new JList(), blueList = new JList(),
                  greenList = new JList();

    private DefaultListModel redModel = new DefaultListModel(),
                             blueModel = new DefaultListModel(),
```

```
                        greenModel = new DefaultListModel();

    private boolean isAdjusting = false;

    public void updateChooser() {
       if( ! isAdjusting) {
          isAdjusting = true;

          Color color = getColorFromModel();
          int r = color.getRed(), g = color.getGreen(),
              b = color.getBlue();

          redList.setSelectedIndex(r);
          redList.ensureIndexIsVisible(r);

          blueList.setSelectedIndex(b);
          blueList.ensureIndexIsVisible(b);

          greenList.setSelectedIndex(g);
          greenList.ensureIndexIsVisible(g);

          isAdjusting = false;
       }
    }
     protected void buildChooser() {
       redList.setFixedCellWidth(50);
       greenList.setFixedCellWidth(50);
       blueList.setFixedCellWidth(50);

       for(int i=0; i < 256; ++i) {
          redModel.addElement(Integer.toString(i));
          greenModel.addElement(Integer.toString(i));
          blueModel.addElement(Integer.toString(i));
       }

       redList.setModel(redModel);
       greenList.setModel(greenModel);
       blueList.setModel(blueModel);

       listPanel.setLayout(new GridLayout(0,3,10,0));

       listPanel.add(new JScrollPane(redList));
       listPanel.add(new JScrollPane(blueList));
       listPanel.add(new JScrollPane(greenList));

       labelPanel.setLayout(new GridLayout(0,3,10,0));

       labelPanel.add(new JLabel("Red"));
       labelPanel.add(new JLabel("Blue"));
       labelPanel.add(new JLabel("Green"));
```

```
        setLayout(new BorderLayout());
        add(labelPanel, BorderLayout.NORTH);
        add(listPanel, BorderLayout.CENTER);

        redList.addListSelectionListener(this);
        greenList.addListSelectionListener(this);
        blueList.addListSelectionListener(this);
    }
    public void valueChanged(ListSelectionEvent e) {
        int r = redList.getSelectedIndex(),
            b = blueList.getSelectedIndex(),
            g = greenList.getSelectedIndex();

        if(r != -1 && g != -1 && b != -1)
            getColorSelectionModel().setSelectedColor(
                                        new Color(r,g,b));
    }
     public String getDisplayName() {
        return "display name";
    }
     public Icon getSmallDisplayIcon() {
        return null;
    }
     public Icon getLargeDisplayIcon() {
        return null;
    }
}
```

The `JColorChooser` class is summarized in Component Summary 16-2.

Component Summary 16-2 JColorChooser

Model(s)	javax.swing.colorchooser.ColorSelectionModel
UI Delegate(s)	javax.swing.plaf.basic.BasicColorChooserUI
Renderer(s)	——
Editor(s)	——
Events Fired	ActionEvents / ChangeEvents / PropertyChangeEvents
Replacement For	——

Class Diagrams

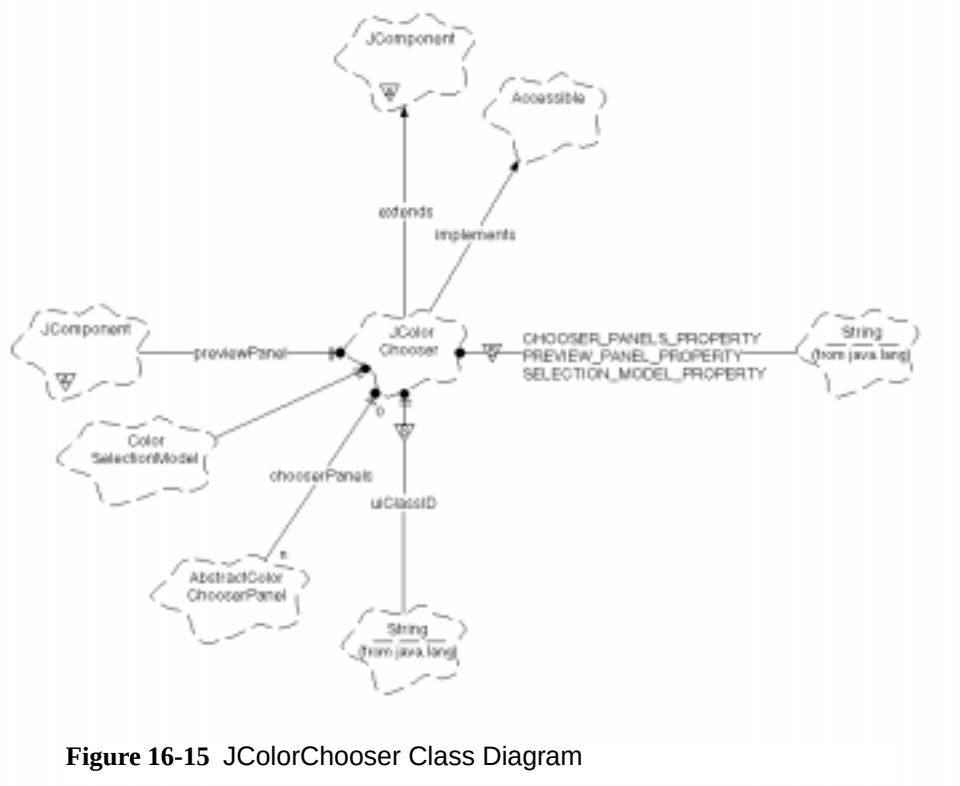


Figure 16-15 JColorChooser Class Diagram

`JColorChooser` extends `JComponent` and implements the `Accessible` interface. `JColorChooser` maintains `protected` references to its preview panel—an instance of `JComponent` and an array of color chooser panels—objects that implement the `AbstractColorChooserPanel` class. Additionally, the `JColorChooser` class defines a set of `public static` strings used to represent the properties associated with instances of `JColorChooser`.

JColorChooser Properties

The properties maintained by the `JColorChooser` class are listed in Table 16-4.

Table 16-4 JColorChooser Properties

Property Name	Data Type	Property Type[1]	Access[3]	Default[4]
chooserPanels	AbstractColorChooserPanel[]	B	SG	L&F
color	Color	S[2]	CSG	L&F
previewPanel	JComponent	B	SG	L&F
selectionModel	ColorSelectionModel	B	CSG	L&F

1. B = bound (fires PropertyChangeEvent) / C = constrained/ I = indexed / S = simple / Ch = fires ChangeEvent / LD = fires ListDataEvent
2. selection model fires a change event when color is changed
3. C = settable at construction time / G = getter method / S = setter method
4. L&F = look-and-feel dependent

`chooserPanels` — The panels displayed in a color chooser that are used to select a color.

`color` — The currently selected color in a color chooser.

`previewPanel` — A panel that visually communicates the currently selected color. The default preview panel can be replaced with the `JColorChooser.setPreviewPanel` method. See "Preview Panels" on page 988 for more information concerning replacing the default preview panel.

`selectionModel` — A selection model that implements the `swing.colorchooser.ColorSelectionModel` interface. Color chooser selection models fire change events when the currently selected color is modified.

JColorChooser Events

The most commonly handled events for color choosers are action events fired when the non-modal color chooser dialog created by `JColorChooser.createDialog()` is dismissed. Listeners cannot be specified for the buttons contained in the modal color chooser dialogs created with `JColorChooser.showDialog()`.

Significant events fired by the classes that comprise color choosers are listed in Table 16-5.

Table 16-5 Color Chooser Events

Event	Fired by	Triggers/Handling
Action	JColorChooser	*trigger*: OK/Cancel buttons
Change	DefaultColor SelectionModel	*trigger*: color selection, *handling*: UI delegate sets preview panel's foreground color
Property Change	JColorChooser	*handling*: UI delegate reacts to preview panel and chooser panel changes

The default color selection model for color choosers fires change events when the selected color is modified. The file chooser's UI delegate reacts to change events from the selection model by setting the preview panel's foreground color to the selected color and repainting the panel.

Color choosers fire property change events when their bound properties are modified. To react to changes in a color chooser's `color` property, a change listener must be registered with the chooser's selection model, as illustrated in Example 16-9 on page 980.

JColorChooser Class Summaries

The `public` and `protected` variables and methods for `JColorChooser` are listed in Class Summary 15-10.

Class Summary 15-10 JColorChooser

Constants

public static final String CHOOSER_PANELS_PROPERTY
public static final String PREVIEW_PANEL_PROPERTY
public static final String SELECTION_MODEL_PROPERTY

The constants defined by the `JColorChooser` class identify color chooser properties.

Constructors

---

public JColorChooser()

public JColorChooser(Color)
public JColorChooser(ColorSelectionModel)

The `JColorChooser` class provides the three constructors listed above. The no-argument constructor creates a color chooser with an initial color of white. The initial color can also be specified explicitly with the last two constructors listed above by a color or color selection model, respectively.

Methods

Creating and Showing Color Chooser Dialogs

---

public static JDialog createDialog(Component, String, boolean, JColorChooser,
ActionListener, ActionListener)
public static Color showDialog(Component, String, Color)

The `static` methods listed above can be used to create or show a dialog containing an instance of `JColorChooser`. The methods are discussed in "Displaying Color Choosers In Dialogs" on page 981.

Chooser Panels

---

public void addChooserPanel(AbstractColorChooserPanel)
public AbstractColorChooserPanel removeChooserPanel(AbstractColorChooserPanel)

public AbstractColorChooserPanel[] getChooserPanels()
public void setChooserPanels(AbstractColorChooserPanel[])

Color choosers allow total control over the color chooser panels that they display. The first two methods listed above can be used to add and remove panels from the array of color chooser panels maintained by instances of `JColorChooser`.

The last two methods listed above can be used to access the entire array of color chooser panels and to set the array, respectively. See "Color Chooser Panels" on page 990 for an example of replacing the color chooser panels that reside in a color chooser.

Color and Color Selection Model

---

public Color getColor()
public void setColor(int colorBits)
public void setColor(int red, int green, int blue)
public void setColor(Color)

public ColorSelectionModel getSelectionModel()
public void setSelectionModel(ColorSelectionModel)

The selected color in a color chooser can be explicitly controlled by the `setColor` methods listed above. A color can be specified by an `integer` value whose low-order byte specifies the blue component, followed by the green component in the next byte, and the red component in the high-order byte.

Setting a color chooser's color causes its selection model to fire a change event. See Example 16-9 on page 980 for an example of to a change in a color chooser's color.

`JColorChoosers` have a color selection model that implements the `swing.colorchooser.ColorSelectionModel` interface. A color chooser's selection model can be obtained or set after construction with the last two methods listed above.

Preview Panels

---

public JComponent getPreviewPanel()
public void setPreviewPanel(JComponent)

Color choosers display a preview panel that visually communicates the currently selected color. The standard preview panel can be replaced with the last method listed above.

Accessibility / Pluggable Look and Feel

---

public AccessibleContext getAccessibleContext()
public ColorChooserUI getUI()
public String getUIClassID()
public void setUI(ColorChooserUI)
public void updateUI()

The methods listed above can be found in most extensions of `JComponent`. Swing lightweight components can return the class name of their UI delegate and an accessible context that contains accessibility information for the component. The `updateUI` method is invoked when the component is fitted with a UI delegate.

AWT Compatibility

The AWT does not provide a component analogous to `JColorChooser`.

Parting Shots

Swing's file and color choosers are highly configurable components that are vital to modern graphical user interfaces. The file chooser can be customized in a number of different ways including accessory components and file filters. Both choosers are easily displayed in dialogs.

As of Swing 1.1 FCS, multiple selection for file choosers is not yet implemented and preview panels cannot be set for color choosers due to bugs.