



CHAPTER 11

Text components and undo

11.1 Text components overview	292	11.5 Using Formats and InputVerifier	312
11.2 Using the basic text components	304	11.6 Formatted Spinner example	319
11.3 JFormattedTextField	306	11.7 Undo/redo	321
11.4 Basic JFormattedTextField example	310		

11.1 TEXT COMPONENTS OVERVIEW

This chapter summarizes the most basic and commonly used text component features, and it introduces the `undo` package. In the next chapter we'll develop a basic `JTextArea` application to demonstrate the use of menus and toolbars. In chapter 19, we'll discuss the inner workings of text components in much more detail. In chapter 20, we'll develop an extensive `JTextPane` html editor application with powerful font, style, paragraph, find and replace, and spell-checking dialogs.

11.1.1 JTextComponent

abstract class `javax.swing.text.JTextComponent`

The `JTextComponent` class serves as the superclass of each Swing text component. All text component functionality is defined by this class, along with the plethora of supporting classes and interfaces provided in the `text` package. The text components themselves are members of the `javax.swing` package: `JTextField`, `JPasswordField`, `JTextArea`, `JEditorPane`, and `JTextPane`.

NOTE We have purposely left out most of the details behind text components in this chapter so we could provide only the information that you will most likely need on a regular basis. If, after reading this chapter, you would like a more thorough understanding of how text components work, and how to customize them or take advantage of some of the more advanced features, see chapters 19 and 20.

`JTextComponent` is an abstract subclass of `JComponent`, and it implements the `Scrollable` interface (see chapter 7). Each multi-line text component is designed to be placed in a `JScrollPane`.

Textual content is maintained in instances of the `javax.swing.text.Document` interface, which acts as the text component model. The text package includes two concrete `Document` implementations: `PlainDocument` and `StyledDocument`. `PlainDocument` allows one font and one color, and it is limited to character content. `StyledDocument` is much more complex, allowing multiple fonts, colors, embedded images and components, and various sets of hierarchically resolving textual attributes. `JTextField`, `JPasswordField`, and `JTextArea` each use a `PlainDocument` model. `JEditorPane` and `JTextPane` use a `StyledDocument` model. We can retrieve a text component's `Document` with `getDocument()`, and assign one with `setDocument()`. We can also attach `DocumentListeners` to a document to listen for changes in that document's content (this is much different than a key listener because all document events are dispatched *after* a change has been made).

We can assign and retrieve the color of a text component's `Caret` with `setCaretColor()` and `getCaretColor()`. We can also assign and retrieve the current `Caret` position in a text component with `setCaretPosition()` and `getCaretPosition()`.

JAVA 1.4 In Java 1.4 the new `NavigationFilter` class has been added in the `javax.swing.text` package. By installing an instance of `NavigationFilter` on a text component, using the new `setNavigationFilter()` method, you can control and restrict caret movement. `NavigationFilter` is most commonly used in combination with an instance of `JFormattedTextField.AbstractFormatter`. See section 11.3.

The `disabledColor` property assigns a font color to be used in the disabled state. The `foreground` and `background` properties inherited from `JComponent` also apply; the foreground color is used as the font color when a text component is enabled, and the background color is used as the background for the whole text component. The `font` property specifies the font to render the text in. The `font` property and the foreground and background color properties do not overpower any attributes assigned to styled text components such as `JEditorPane` and `JTextPane`.

All text components maintain information about their current selection. We can retrieve the currently selected text as a `String` with `getSelectedText()`, and we can assign and retrieve specific background and foreground colors to use for selected text with `setSelectionBackground()/getSelectionBackground()` and `setSelectionForeground()/getSelectionForeground()` respectively.

`JTextComponent` also maintains a bound `focusAccelerator` property, which is a `char` that is used to transfer focus to a text component when the corresponding key is pressed simultaneously with the ALT key. This works internally by calling `requestFocus()` on the text component, and it will occur as long as the top-level window containing the given text compo-

ment is currently active. We can assign/retrieve this character with `setFocusAccelerator()`/`getFocusAccelerator()`, and we can turn this functionality off by assigning `'\0'`.

The `read()` and `write()` methods provide convenient ways to read and write text documents. The `read()` method takes a `java.io.Reader` and an `Object` that describes the `Reader` stream, and it creates a new document model appropriate to the given text component containing the obtained character data. The `write()` method stores the content of the document model in a given `java.io.Writer` stream.

WARNING We can customize any text component's document model. However, it is important to realize that whenever the `read()` method is invoked, a new document will be created. Unless this method is overridden, a custom document that had been previously assigned with `setDocument()` will be lost whenever `read()` is invoked, because the current document will be replaced by a default instance.

11.1.2 JTextField

class javax.swing.JTextField

`JTextField` is a single-line text component that uses a `PlainDocument` model. The `horizontalAlignment` property specifies text justification within the text field. We can assign/retrieve this property with `setHorizontalAlignment()/getHorizontalAlignment()`. Acceptable values are `JTextField.LEFT`, `JTextField.CENTER`, and `JTextField.RIGHT`.

There are several `JTextField` constructors, two of which allow us to specify a number of columns. We can also assign/retrieve this number, the `columns` property, with `setColumns()/getColumns()`. Specifying a certain number of columns will set up a text field's preferred size to accommodate at least an equivalent number of characters. However, a text field might not receive its preferred size due to the current layout manager. Also, the width of a column is the width of the character 'm' in the current font. Unless a monospaced font is used, this width will be greater than most other characters.

The following example creates 14 `JTextFields` with a varying number of columns. Each field contains a number of ms equal to its number of columns.

Example 11.1

JTextFieldTest.java

```
see \Chapter11\1

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

public class JTextFieldTest extends JFrame
{
    public JTextFieldTest() {
        super("JTextField Test");

        getContentPane().setLayout(new FlowLayout());

        JTextField textField1 = new JTextField("m",1);
        JTextField textField2 = new JTextField("mm",2);
        JTextField textField3 = new JTextField("mmm",3);
```

```

    JTextField textField4 = new JTextField("mmmm", 4);
    JTextField textField5 = new JTextField("mmmmm", 5);
    JTextField textField6 = new JTextField("mmmmm", 6);
    JTextField textField7 = new JTextField("mmmmmm", 7);
    JTextField textField8 = new JTextField("mmmmmmmm", 8);
    JTextField textField9 = new JTextField("mmmmmmmmmm", 9);
    JTextField textField10 = new JTextField("mmmmmmmmmm", 10);
    JTextField textField11 = new JTextField("mmmmmmmmmmmm", 11);
    JTextField textField12 = new JTextField("mmmmmmmmmmmm", 12);
    JTextField textField13 = new JTextField("mmmmmmmmmmmmmm", 13);
    JTextField textField14 = new JTextField("mmmmmmmmmmmmmm", 14);

    getContentPane().add(textField1);
    getContentPane().add(textField2);
    getContentPane().add(textField3);
    getContentPane().add(textField4);
    getContentPane().add(textField5);
    getContentPane().add(textField6);
    getContentPane().add(textField7);
    getContentPane().add(textField8);
    getContentPane().add(textField9);
    getContentPane().add(textField10);
    getContentPane().add(textField11);
    getContentPane().add(textField12);
    getContentPane().add(textField13);
    getContentPane().add(textField14);

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300,170);
    setVisible(true);
}

public static void main(String argv[]) {
    new JTextFieldTest();
}
}

```

Figure 11.1 illustrates the output. Notice that none of the text completely fits in its field. This happens because `JTextField` does not factor in the size of its border when calculating its preferred size, as we might expect. To work around this problem, though this is not an ideal solution, we can add one more column to each text field. The result is shown in figure 11.2. This solution is more appropriate when a fixed width font (monospaced) is being used. Figure 11.3 illustrates this last solution.

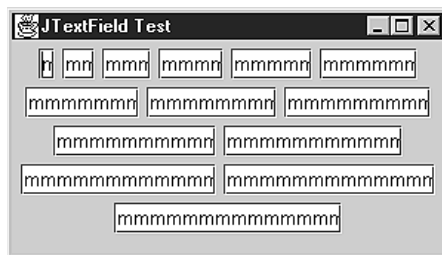


Figure 11.1
JTextFields using an equal number
of columns and "m" characters

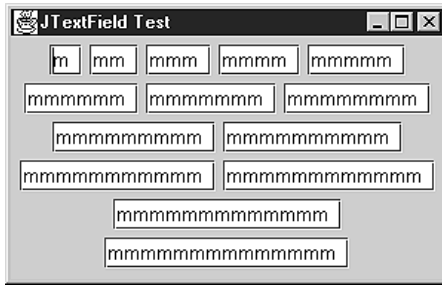


Figure 11.2
JTextFields using one more column
than the number of “m” characters

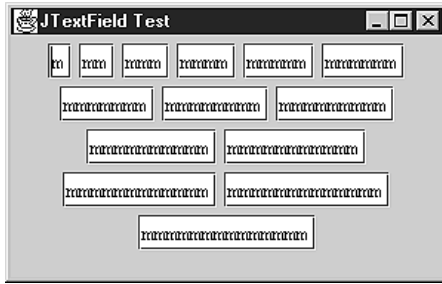


Figure 11.3
JTextFields using a monospaced font,
and one more column than the number
of “m” characters

NOTE Using a monospaced font is always more appropriate when a fixed character limit is desired.

JTextField also maintains a BoundedRangeModel (see chapter 13) as its horizontal-visibility property. This model is used to keep track of the amount of currently visible text. The minimum is 0 (the beginning of the document), and the maximum is equal to the width of the text field or the total length of the text in pixels (whichever is greater). The value is the current offset of the text displayed at the left edge of the field, and the extent is the width of the text field in pixels.

By default, a `KeyStroke` (see section 2.13.2) is established with the ENTER key that causes an `ActionEvent` to be fired. By simply adding an `ActionListener` to a `JTextField`, we will receive events whenever ENTER is pressed while that field has the current focus. This is very convenient functionality, but it may also get in the way of things. To remove this registered keystroke, do the following:

```
KeyStroke enter = KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0);
Keymap map = myJTextField.getKeymap();
map.removeKeyStrokeBinding(enter);
```

JTextField’s document model can be customized to allow only certain forms of input; this is done by extending `PlainDocument` and overriding the `insertString()` method. The following code shows a class that will only allow six or fewer digits to be entered. We can assign this document to a `JTextField` with the `setDocument()` method (see chapter 19 for more about working with Documents).

```
class SixDigitDocument extends PlainDocument
{
    public void insertString(int offset,
        String str, AttributeSet a)
```

```

throws BadLocationException {
    char[] insertChars = str.toCharArray();

    boolean valid = true;
    boolean fit = true;
    if (insertChars.length + getLength() <= 6) {
        for (int i = 0; i < insertChars.length; i++) {
            if (!Character.isDigit(insertChars[i])) {
                valid = false;
                break;
            }
        }
    }
    else
        fit = false;

    if (fit && valid)
        super.insertString(offset, str, a);
    else if (!fit)
        getToolkit().beep();
}
}

```

JAVA 1.4

In Java 1.4 the new `JFormattedTextField` component has been added to more easily allow the creation of customized input fields. We'll discuss this component along with several examples of its use in sections 11.4, 11.5, and 11.6.

Java 1.4 also includes a new `DocumentFilter` class in the `javax.swing.text` package. When an instance of `DocumentFilter` is installed on a `Document`, all invocations of `insertString()`, `remove()`, and `replace()` get forwarded on to the `DocumentFilter`. This allows clean encapsulation of all custom document mutation code. So, for instance, the `SixDigitDocument` code would be more appropriately built into a `DocumentFilter` subclass. In this way different filters can be applied to various documents without the need to change a given `Document` instance. To support `DocumentFilters`, `AbstractDocument` includes the new `setDocumentFilter()` and `getDocumentFilter()` methods. `DocumentFilter` is most commonly used in combination with an instance of `JFormattedTextField.AbstractFormatter`. See section 11.3.



Don't overly restrict input Filtering text fields during data entry is a powerful aid to usability. It helps prevent the user from making a mistake and it can speed operations by removing the need for validation and correction procedures. However, it is important not to overly restrict the allowable input. Make sure that all reasonable input is expected and accepted.

For example, with a phone number, allow “00 1 44 654 7777,” “00+1 44 654 7777,” and “00-1-1-654-7777,” as well as “00144654777.” Phone numbers can contain more than just numbers!

Another example involves dates. You should allow “04-06-99,” “04/06/99,” and “04:06:99,” as well as “040699.”

11.1.3 JPasswordField

class javax.swing.JPasswordField

`JPasswordField` is a fairly simple extension of `JTextField` that displays an echo character instead of the actual content that is placed in its model. This echo character defaults to `*`, and we can assign a different character with `setEchoChar()`.

Unlike other text components, we cannot retrieve the actual content of a `JPasswordField` with `getText()` (this method, along with `setText()`, has been deprecated in `JPasswordField`). Instead we must use the `getPassword()` method, which returns an array of `chars`. `JPasswordField` overrides the `JTextComponent` `copy()` and `cut()` methods to do nothing but emit a beep, for security reasons.

Figure 11.4 shows the `JTextFieldDemo` example of section 11.1.2. It uses `JPasswordField`s instead, and each is using a monospaced font.

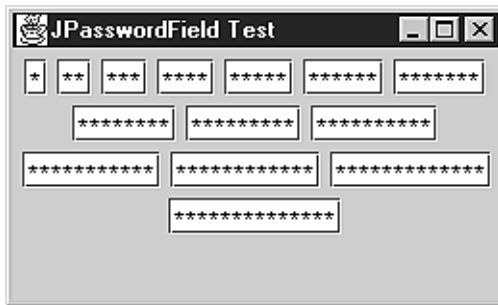


Figure 11.4
JPasswordFields using a mono-
spaced font, and one more column
than number of characters

11.1.4 JTextArea

class javax.swing.JTextArea

`JTextArea` allows multiple lines of text and, like `JTextField`, it uses a `PlainDocument` model. As we discussed earlier, `JTextArea` cannot display multiple fonts or font colors. `JTextArea` can perform line wrapping and, when line wrapping is enabled we can specify whether lines break on word boundaries. To enable/disable line wrapping we set the `lineWrap` property with `setLineWrap()`. To enable/disable wrapping on boundaries (which will only have an effect when `lineWrap` is set to `true`) we set the `wrapStyleWord` property using `setWrapStyleWord()`. Both `lineWrap` and `wrapStyleWord` are bound properties.

`JTextArea` overrides `isManagingFocus()` (see section 2.12) to return `true`, indicating that the `FocusManager` will not transfer focus out of a `JTextArea` when the `TAB` key is pressed. Instead, a tab is inserted into the document (the number of spaces in the tab is equal to `tabSize`). We can assign/retrieve the tab size with `setTabSize()/getTabSize()` respectively. `tabSize` is also a bound property.

There are several ways to add text to a `JTextArea`'s document. We can pass this text in to one of the constructors, append it to the end of the document using the `append()` method, insert a string at a given character offset using the `insert()` method, or replace a given range of text with the `replaceRange()` method. As with any text component, we can also set the

text with the `JTextComponent setText()` method, and we can add and remove text directly from its `Document` (see chapter 19 for more details about the `Document` interface).

`JTextArea` maintains `lineCount` and `rows` properties which can easily be confused. The `rows` property specifies how many rows of text `JTextArea` is actually displaying. This may change whenever a text area is resized. The `lineCount` property specifies how many lines of text the document contains. Each line consists of a set of characters ending in a line break (`\n`). We can retrieve the character offset of the end of a given line with `getLineEndOffset()`, the character offset of the beginning of a given line with `getLineStartOffset()`, and the line number that contains a given offset with `getLineOfOffset()`.

The `rowHeight` and `columnWidth` properties are determined by the height and width of the current font. The width of one column is equal to the width of the “m” character in the current font. We cannot assign new values to the properties, but we can override the `getColumnWidth()` and `getRowHeight()` methods in a subclass to return any value we like. We can explicitly set the number of rows and columns a text area contains with `setRows()` and `setColumns()`, and the `getRows()` and `getColumns()` methods will only return these explicitly assigned values (not the current row and column count, as we might assume at first glance).

Unless `JTextArea` is placed in a `JScrollPane` or a container using a layout manager which enforces a certain size, it will resize itself dynamically depending on the amount of text that is entered. This behavior is rarely desired.

11.1.5 JEditorPane

class javax.swing.JEditorPane

`JEditorPane` is a multi-line text component capable of displaying and editing various different types of content. Swing provides support for HTML and RTF, but there is nothing stopping us from defining our own content type, or implementing support for an alternate format.

NOTE Swing’s support for HTML and RTF is located in the `javax.swing.text.html` and `javax.swing.text.rtf` packages.

Support for different content is accomplished in part through the use of custom `EditorKit` objects. `JEditorPane`’s `contentType` property is a `String` that represents the type of document the editor pane is currently set up to display. The `EditorKit` maintains this value which, for `DefaultEditorKit`, defaults to “text/plain.” `HTMLEditorKit` and `RTFEditorKit` have `contentType` values of “text/html” and “text/rtf”, respectively (see chapter 19 for more about `EditorKits`).

In chapter 9 we built a simple web browser using a non-editable `JEditorPane` by passing a URL to its constructor. When it’s in non-editable mode, `JEditorPane` displays HTML pretty much as we might expect, although it has a long way to go to match Netscape. By allowing editing, `JEditorPane` will display an HTML document with many of its tags specially rendered, as shown in figure 11.5 (compare this to figure 9.4).

`JEditorPane` is smart enough to use an appropriate `EditorKit`, if one is available, to display a document passed to it. When it’s displaying an HTML document, `JEditorPane` can fire `HyperlinkEvents` (which are defined in the `javax.swing.event` package). We can attach `HyperlinkListeners` to `JEditorPane` to listen for hyperlink invocations, as

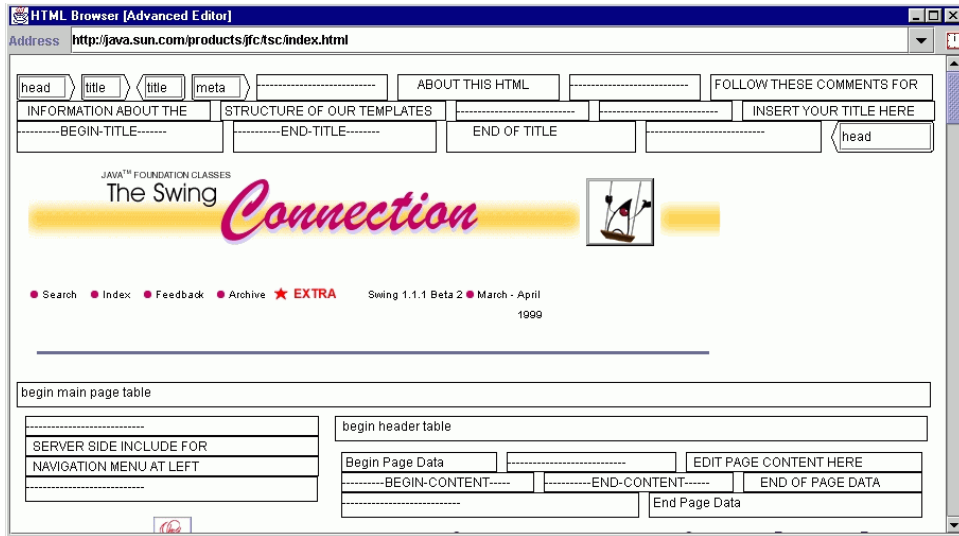


Figure 11.5 A `JEditorPane` displaying HTML in editable mode

demonstrated by the examples at the end of chapter 9. The following code shows how simple it is to construct an HTML browser using an active `HyperlinkListener`.

```

m_browser = new JEditorPane(
    new URL("http://java.sun.com/products/jfc/tsc/index.html"));
m_browser.setEditable(false);
m_browser.addHyperlinkListener( new HyperlinkListener() {
    public void hyperlinkUpdate(HyperlinkEvent e) {
        if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            URL url = e.getURL();
            if (url == null)
                return;
            try { m_browser.setPage(e.getURL); }
            catch (IOException e) { e.printStackTrace(); }
        }
    }
}

```

`JEditorPane` uses a `Hashtable` to store its editor kit/content type pairs. We can query this table and retrieve the editor kit associated with a particular content type, if there is one, using the `getEditorKitForContentType()` method. We can get the current editor kit with `getEditorKit()`, and the current content type with `getContentType()`. We can set the current content type with `setContentType()`, and if there is already a corresponding editor kit in `JEditorPane`'s hashtable, an appropriate editor kit will replace the current one. We can also assign an editor kit for a given content type using the `setEditorKitForContentType()` method (we will discuss `EditorKits`, and the ability to construct our own, in chapter 19).

`JEditorPane` uses a `DefaultStyledDocument` as its model. In HTML mode, an `HTMLDocument`, which extends `DefaultStyledDocument`, is used. `DefaultStyledDocument` is quite powerful, as it allows us to associate attributes with characters and paragraphs, and to apply logical styles (see chapter 19).

11.1.6 JTextPane

class `javax.swing.JTextPane`

`JTextPane` extends `JEditorPane` and thus inherits its abilities to display various types of content. The most significant functionalities `JTextPane` offers are the abilities to programmatically assign attributes to regions of its content, embed components and images within its document, and work with named sets of attributes called `Styles` (we will discuss `Styles` in chapters 19 and 20).

To assign attributes to a region of document content, we use an `AttributeSet` implementation. We will describe `AttributeSets` in detail in chapter 19, but we will tell you here that they contain a group of attributes such as font type, font style, font color, and paragraph and character properties. These attributes are assigned through the use of various static methods which are defined in the `StyleConstants` class, which we will also discuss further in chapter 19.

Example 11.2 demonstrates embedded icons, components, and stylized text. Figure 11.6 illustrates the output.



Figure 11.6 A `JTextPane` with inserted `ImageIcons`, text with attributes, and an active `JButton`

Example 11.2

JTextPaneDemo.java

see \Chapter11\2

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

public class JTextPaneDemo extends JFrame
{
    // Best to reuse attribute sets as much as possible.
    static SimpleAttributeSet ITALIC_GRAY = new SimpleAttributeSet();
    static SimpleAttributeSet BOLD_BLACK = new SimpleAttributeSet();
    static SimpleAttributeSet BLACK = new SimpleAttributeSet();

    static {
        StyleConstants.setForeground(ITALIC_GRAY, Color.gray);
        StyleConstants.setItalic(ITALIC_GRAY, true);
        StyleConstants.setFontFamily(ITALIC_GRAY, "Helvetica");
        StyleConstants.setFontSize(ITALIC_GRAY, 14);

        StyleConstants.setForeground(BOLD_BLACK, Color.black);
        StyleConstants.setBold(BOLD_BLACK, true);
        StyleConstants.setFontFamily(BOLD_BLACK, "Helvetica");
        StyleConstants.setFontSize(BOLD_BLACK, 14);

        StyleConstants.setForeground(BLACK, Color.black);
        StyleConstants.setFontFamily(BLACK, "Helvetica");
        StyleConstants.setFontSize(BLACK, 14);
    }

    JTextPane m_editor = new JTextPane();

    public JTextPaneDemo() {
        super("JTextPane Demo");

        JScrollPane scrollPane = new JScrollPane(m_editor);
        getContentPane().add(scrollPane, BorderLayout.CENTER);

        setEndSelection();
        m_editor.insertIcon(new ImageIcon("manning.gif"));
        insertText("\nHistory: Distant\n\n", BOLD_BLACK);

        setEndSelection();
        m_editor.insertIcon(new ImageIcon("Lee_fade.jpg"));
        insertText("                ", BLACK);
        setEndSelection();
        m_editor.insertIcon(new ImageIcon("Bace_fade.jpg"));

        insertText("\n        Lee Fitzpatrick        "
            + "
            + "Marjan Bace\n\n", ITALIC_GRAY);
    }
}
```

```

insertText("When we started doing business under " +
    "the Manning name, about 10 years ago, we were a very " +
    "different company. What we are now is the end result of " +
    "an evolutionary process in which accidental " +
    "events played as big a role, or bigger, as planning and " +
    "foresight.\n", BLACK);

setEndSelection();
JButton manningButton = new JButton("Visit Manning");
manningButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        m_editor.setEditable(false);
        try { m_editor.setPage("http://www.manning.com"); }
        catch (IOException ioe) { ioe.printStackTrace(); }
    }
});
m_editor.insertComponent(manningButton);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

setSize(500,450);
setVisible(true);
}

protected void insertText(String text, AttributeSet set) {
    try {
        m_editor.getDocument().insertString(
            m_editor.getDocument().getLength(), text, set);
    }
    catch (BadLocationException e) {
        e.printStackTrace();
    }
}

protected void setEndSelection() {
    m_editor.setSelectionStart(m_editor.getDocument().getLength());
    m_editor.setSelectionEnd(m_editor.getDocument().getLength());
}

public static void main(String argv[]) {
    new JTextPaneDemo();
}
}

```

As example 11.2 demonstrates, we can insert images and components with `JTextPane`'s `insertIcon()` and `insertComponent()` methods. These methods insert the given object by replacing the current selection. If there is no current selection, they will be placed at the beginning of the document. This is why we defined the `setEndSelection()` method in our example above to point the selection to the end of the document where we want to do insertions.

When inserting text, we cannot simply append it to the text pane itself. Instead we retrieve its document and call `insertString()`. To give attributes to inserted text we can construct `AttributeSet` implementations, and we can assign attributes to that set using the `StyleConstants` class. In the example above we do this by constructing three `SimpleAttributeSets` as static instances (so that they may be reused as much as possible).

As an extension of `JEditorPane`, `JTextPane` uses a `DefaultStyledDocument` for its model. Text panes use a special editor kit, `DefaultStyledEditorKit`, to manage their `Actions` and `Views`. `JTextPane` also supports the use of `Styles`, which are named collections of attributes. We will discuss styles, actions, and views as well as many other advanced features of `JTextPane` in chapters 19 and 20.

11.2 USING THE BASIC TEXT COMPONENTS

The following example demonstrates the use of the basic text components (`JTextField`, `JPasswordField`, and `JTextArea`) in a personal data dialog box.

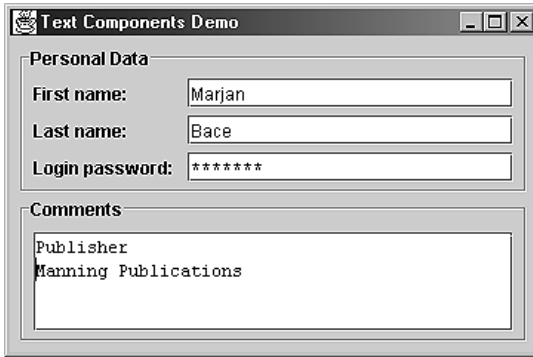


Figure 11.7
Basic text components demo;
a personal data dialog box

Example 11.3

TextDemo.java

```
see \Chapter11\3
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

import dl.*;

public class TextDemo extends JFrame {
    protected JTextField m_firstTxt;
    protected JTextField m_lastTxt;
    protected JPasswordField m_passwordTxt;
    protected JTextArea m_commentsTxt;

    public TextDemo() {
        super("Text Components Demo");
        Font monospaced = new Font("Monospaced", Font.PLAIN, 12);
        JPanel pp = new JPanel(new BorderLayout(0));

        JPanel p = new JPanel(new DialogLayout());
        p.setBorder(new JLabel("First name:"));
```

```

p.add(new JLabel("First name:"));
m_firstTxt = new JTextField(20);
p.add(m_firstTxt);

p.add(new JLabel("Last name:"));
m_lastTxt = new JTextField(20);
p.add(m_firstTxt);

p.add(new JLabel("Login password:"));
m_passwordTxt = new JPasswordField(20);
m_passwordTxt.setFont(monospaced);
p.add(m_passwordTxt);

p.setBorder(new CompoundBorder(
    new TitledBorder(new EtchedBorder(), "personal Data"),
    new EmptyBorder(1, 5, 3, 5)
));
pp.add(p, BorderLayout.NORTH);

m_commentsTxt = new JTextArea("", 4, 30);
m_commentsTxt.setFont(monospaced);
m_commentsTxt.setLineWrap(true);
m_commentsTxt.setWrapStyleWord(true);
p = new JPanel(new BorderLayout());
p.add(new JScrollPane(m_commentsTxt));
p.setBorder(new CompoundBorder(
    new TitledBorder(new EtchedBorder(), "comments"),
    new EmptyBorder(3, 5, 3, 5)
));
pp.add(p, BorderLayout.CENTER);

pp.setBorder(new EmptyBorder(5, 5, 5, 5));
getContentPane().add(pp);
pack();
}

public static void main(String[] args) {
    JFrame frame = new TextDemo();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}

```

1 Instructs the text area to wrap lines and words as more text

11.2.1 Understanding the Code

Class TextDemo

This class extends `JFrame` to implement the frame container for the following four text components used to input personal data:

- `JTextField m_firstTxt`: text field for the first name.
- `JTextField m_lastTxt`: text field for the last name.
- `JPasswordField m_passwordTxt`: password field.
- `JTextArea m_commentsTxt`: text area for comments.

The `DialogLayout` layout manager described in chapter 4 is used to lay out components in pairs: label on the left, text components on the right. (Note that you don't have to supply any additional constraints or parameters to this layout manager.)

- 1 The various settings applied to `JTextArea m_commentsTxt` instruct it to wrap text by lines and words rather than allow it to scroll horizontally as more text is entered.

11.2.2 Running the code

Figure 11.7 shows this demo in action. Note how text wraps in the comment box. Try commenting out the following lines individually and note the effects:

```
m_commentsTxt.setLineWrap(true);  
M_commentsTxt.setWrapStyleWord(true);
```

11.3 JFormattedTextField

class javax.swing.JFormattedTextField

`JFormattedTextField` is a new Swing component introduced in Java 1.4. This component extends `JTextField` and adds support for custom formatting.

The simplest way to use `JFormattedTextField` is to pass an instance of `java.text.Format` class to the component's constructor. This `Format` instance will be used to enforce the format of data input as a number, date, and so forth. Subclasses of `Format` include `DateFormat`, `NumberFormat`, and `MessageFormat` among others.

The formatting itself is handled by an instance of the inner `JFormattedTextField.AbstractFormatter` class which is normally obtained by an instance of the inner `JFormattedTextField.AbstractFormatterFactory` class. The default `JFormattedTextField` constructor installs a `DefaultFormatter` instance as its `JFormattedTextField.AbstractFormatter`. `DefaultFormatter`, `DefaultFormatter` and its subclasses, `MaskFormatter`, `InternationalFormatter`, `DateFormatter`, and `NumberFormatter` are described later in this section.

The `setFormatter()` method is protected, indicating that you should not set the `AbstractFormatter` directly. Rather, this should be done by setting the `AbstractFormatterFactory` with the `setFormatterFactory()` method. If you do not specify an `AbstractFormatter` using this method, or with the appropriate constructor, a concrete `AbstractFormatter` subclass will be used based on the `Class` of the current `JFormattedTextField` value. `DateFormatter` is used for `java.util.Date` values, `NumberFormatter` is used for `java.lang.Number` values, and for all other values `defaultFormatter` is used.

The `setValue()` method takes an `Object` as parameter and assigns it to the value property. It also sends this object to the `AbstractFormatter` instance to deal with appropriately in its `setValue()` method and assign to its value property. `JFormattedTextField` and its `AbstractFormatter` have separate value properties. During editing `AbstractFormatter`'s value is updated. This value is not pushed to `JFormattedTextField` until the `commitEdit()` method is called. This normally occurs when ENTER is pressed or after a focus change occurs.

The `getValue()` method returns an appropriate `Object` representing the current `JFormattedTextField` value. For instance, if a `DateFormatter` is in use a `Date` object will

be returned. This may not be the current value maintained by `AbstractFormatter`. To get the currently edited value the `commitEdit()` method must be invoked before `getValue()` is called.

The `invalidEdit()` method is invoked whenever the user inputs an invalid value, thus providing a way to give feedback to the user. The default implementation simply beeps. This method is normally invoked by `AbstractFormatter`'s `invalidEdit()` method, which is usually invoked whenever the user inputs an invalid character.

The `isValidEdit()` method returns a boolean value specifying whether or not the current field `JFormattedTextField` value is valid with respect to the current `AbstractFormatter` instance.

The `commitEdit()` method forces the current value in `AbstractFormatter` to be set as the current value of the `JFormattedTextField`. Most `AbstractFormatters` invoke this method when ENTER is pressed or a focus change occurs. This method allows us to force a commit programmatically. (Note that when editing a value in `JFormattedTextField`, until a commit occurs `JFormattedTextField`'s value is not updated. The value that is updated prior to a commit is `AbstractFormatter`'s value.)

The `setFocusLostBehavior()` method takes a parameter specifying what `JFormattedTextField`'s behavior should be when it loses the focus. The following `JFormattedTextField` constants are used for this method:

- `JFormattedTextField.REVERT`: revert to current value and ignore changes made to `AbstractFormatter`'s value.
- `JFormattedTextField.COMMIT`: try to commit the current `AbstractFormatter` value as the new `JFormattedTextField` value. This will only be successful if `AbstractFormatter` is able to format its current value as an appropriate return value from its `stringToValue()` method.
- `JFormattedTextField.COMMIT_OR_REVERT`: commit the current `AbstractFormatter` value as the new `JFormattedTextField` value only if `AbstractFormatter` is able to format its current value as an appropriate return value from its `stringToValue()` method. If not, `AbstractFormatter`'s value will revert to `JFormattedTextField`'s current value and ignore any changes.
- `JFormattedTextField.PERSIST`: leave the current `AbstractFormatter` value as is without committing or reverting.

Note that some `AbstractFormatters` may commit changes as they happen, versus when a focus change occurs. In these cases the assigned focus lost behavior will have no effect. (This happens when `DefaultFormatter`'s `commitsOnValidEdit` property is set to `true`.)

11.3.1 `JFormattedTextField.AbstractFormatter`

abstract class javax.swing.JFormattedTextField.AbstractFormatter

An instance of this class is used to install the actual custom formatting and caret movement functionality in a `JFormattedTextField`. Instances of `AbstractFormatter` have a `DocumentFilter` and `NavigationFilter` associated with them to restrict `getDocumentFilter()` and `getNavigationFilter()` methods to return custom filters as necessary.

WARNING `AbstractFormatter` normally installs a `DocumentFilter` on its `Document` instance and a `NavigationFilter` on itself. For this reason you should not install your own, otherwise the formatting and caret movement behavior enforced by `AbstractFormatter` will be overridden.

The `valueToString()` and `stringToValue()` methods are used to convert from `Object` to `String` and `String` to `Object`. Subclasses must override these methods so that `JFormattedTextField`'s `getValue()` and `setValue()` methods know how to behave. These methods throw `ParseException`s if a conversion does not occur successfully.

11.3.2 DefaultFormatter

class javax.swing.text.DefaultFormatter

This `AbstractFormatter` concrete subclass is used by default by `JFormattedTextField` when no formatter is specified. It is meant for formatting any type of `Object`. Formatting is done by calling the `toString()` method on the assigned value object.

In order for the value returned by the `stringToValue()` method to be of the appropriate object type, the class defining that object type must have a that takes a `String` constructor parameter.

The `getValueClass()` method returns the `Class` instance defining the allowed object type. The `setValueClass()` allows you to specify this.

The `setOverwriteMode()` method allows you to specify whether or not text will overwrite current text in the document when typed into `JFormattedTextField`. By default this is `true`.

The `setCommitsOnValidEdit()` method allows you to specify whether or not the current value should be committed and pushed to `JFormattedTextField` after each successful document modification. By default this is `false`.

The `getAllowsInvalid()` method specifies whether the `Format` instance should format the current text on every edit. This is the case if it returns `false`, the default.

11.3.3 MaskFormatter

class javax.swing.text.MaskFormatter

`MaskFormatter` is a subclass of `DefaultFormatter` that is designed to allow editing of custom formatted `Strings`. This formatting is controlled by a `String` mask that declares the valid character types that can appear in specific locations in the document.

The mask can be set as a `String` passed to the constructor or to the `setMask` method. The following characters are allowed, each of which represents a set of characters that will be allowed to be entered in the corresponding position of the document:

- #: represents any valid number character (validated by `Character.isDigit()`)
- \: escape character
- U: any character; lowercase letters are mapped to uppercase (validated by `Character.isLetter()`)
- L: any character; upper case letters are mapped to lowercase (validated by `Character.isLetter()`)

- A: any letter character or number (validated by `Character.isLetter()` or `Character.isDigit()`)
- ?: any letter character (validated by `Character.isLetter()`)
- *: any character
- H: any hex character (i.e., 0-9, a-f or A-F)

Any other characters not in this list that appear in a mask are assumed to be fixed and unchangeable. For example, the following mask will enforce the input of a U.S.-style phone number: `"(###)###-####"`.

The set of valid and invalid characters can be further refined with the `setValidCharacters()` and `setInvalidCharacters()` methods.

By default the placeholder character is a space `' '` representing a character location that needs to be filled in to complete the mask. The `setPlaceholderCharacter()` method provides a way to specify a different character. For instance, with the phone number mask and a `'_'` as the placeholder character, `JFormattedTextField`'s content would initially look like: `"(____) ____-____"`.

11.3.4 InternationalFormatter

class javax.swing.text.InternationalFormatter

`InternationalFormatter` extends `DefaultEditor` and uses a `Format` instance to handle conversion to and from a `String`. This formatter also allows specification of maximum and minimum allowed values with the `setMaximum()` and `setMinimum()` methods which take `Comparable` instances as parameters.

11.3.5 DateFormatter

class javax.swing.text.DateFormatter

`DateFormatter` is an `InternationalFormatter` subclass which uses a `java.text.DateFormat` instance as the `Format` used to handle conversion from `String` to `Date` and `Date` to `String`.

11.3.6 NumberFormatter

class javax.swing.text.NumberFormatter

`NumberFormatter` is an `InternationalFormatter` subclass which uses a `java.text.NumberFormat` instance as the `Format` used to handle conversion from `String` to `Number` and `Number` to `String`. Subclasses of `Number` include `Integer`, `Double`, `Float`, and so forth.

11.3.7 JFormattedTextField.AbstractFormatterFactory

abstract class javax.swing.JFormattedTextField.AbstractFormatterFactory

Instances of this class are used by `JFormattedTextField` to supply an appropriate `AbstractFormatter` instance. An `AbstractFormatterFactory` can supply a different `AbstractFormatter` depending on the state of the `JFormattedTextField`, or some other criteria. This behavior is customizable by implementing the `getFormatter()` method.

11.3.8 DefaultFormatterFactory

class javax.swing.text.DefaultFormatterFactory

This concrete subclass of `AbstractFormatterFactory` is used by default by `JFormattedTextField` when no formatter factory is specified. It allows specification of different formatters to use when `JFormattedTextField` is being edited (i.e., has the focus), just displayed (i.e., does not have the focus), when the value is null, and one for all other cases (the default formatter).

11.4 BASIC JFORMATTEDTEXTFIELD EXAMPLE

The following example demonstrates two `JFormattedTextFields` used for the input of a U.S. dollar amount and date. For the U.S. dollar amount field a locale-dependent currency format is used.



Figure 11.8
Basic `JFormattedTextField` example

Example 11.4

FTFDemo.java

```
see \Chapter11\4

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;

import dl.*;

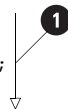
class FTFDemo extends JFrame {

    public FTFDemo() {
        super("Formatted Text Field");

        JPanel p = new JPanel(new DialogLayout2());
        p.setBorder(new EmptyBorder(10, 10, 10, 10));

        p.add(new JLabel("Dollar amount:"));
        NumberFormat formatMoney=
            NumberFormat.getCurrencyInstance(Locale.US);
```

Formatted text field
used for a US dollar
amount; a locale-specific
`NumberFormat` instance
is used to regulate



```

JFormattedTextField ftMoney = new
    JFormattedTextField(formatMoney);
ftMoney.setColumns(10);
ftMoney.setValue(new Double(100));
p.add(ftfMoney);

p.add(new JLabel("Transaction date:"));
DateFormat formatDate = new SimpleDateFormat("MM/dd/yyyy");
JFormattedTextField ftfDate = new JFormattedTextField(formatDate);
ftfDate.setColumns(10);
ftfDate.setValue(new Date());
p.add(ftfDate);

JButton btn = new JButton("OK");
p.add(btn);

getContentPane().add(p, BorderLayout.CENTER);
pack();
}

public static void main( String args[] ) {
    FTFDemo mainFrame = new FTFDemo();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.setVisible(true);
}
}

```

1 Formatted text field used for a US dollar amount; a locale-specific NumberFormat instance is used to regulate formatting

2 Formatted text field used for a date; a DateFormat instance is used to regulate formatting

11.4.1 Understanding the code

Class FTFDemo

This class extends `JFrame` to implement the frame container for two `JFormattedTextFields`:

- `JFormattedTextField ftMoney`: used to input a U.S. dollar amount. Constructor takes an instance of `NumberFormat` as parameter.
 - `JFormattedTextField ftDate`: used to input a date. Constructor takes an instance of `SimpleDateFormat` as parameter.
- 1** The `NumberFormat` instance is created with `NumberFormat`'s static `getCurrencyInstance()` method. This and other `Format` classes provide such static methods to return locale-specific `Format` instances.
 - 2** The `DateFormat` instance is easily created as an instance of `SimpleDateFormat`. `SimpleDateFormat` takes a `String` as its parameter representing how the date should be displayed. Specific characters such as "M", "d" and "Y" have specific meanings (see Javadoc writeup on `SimpleDateFormat` for a complete explanation).

11.4.2 Running the code

Figure 11.8 shows our `JFormattedTextfield` demo in action. Note that actual formatting and validation occurs when a field loses its focus. If a field is improperly formatted, it will revert to its last valid formatted value when it loses focus. Try tweaking the code to experiment with the `setFocusLostBehavior()` method and note how the various focus lost behaviors work.

11.5 USING FORMATS AND INPUTVERIFIER

This example builds on the personal data input dialog concept in section 11.3 to demonstrate how to develop custom formats for use by `JFormattedTextField` and how to use `MaskFormatter` to format and verify input. This example also demonstrates the use of the new `InputVerifier` class (added in Java 1.3) to control focus transfer between text fields based on whether or not data input is correct.

11.5.1 InputVerifier

abstract class javax.swing.InputVerifier

Instances of `InputVerifier` are attached to a `JComponent` through its new `setInputVerifier()` method. Before focus is transferred away from that component, the attached `InputVerifier`'s `shouldYieldFocus()` method is called to determine whether or not the focus transfer should be allowed to occur. If this method returns `true` the focus transfer should proceed, indicating that the currently focused component is in a valid state. If this method returns `false` the focus transfer should not proceed, indicating that the currently focused component is not in a valid state. This can be particularly useful when dealing with text fields and components involving textual input, as example 11.5 shows below.

Note that `InputVerifier` has two methods, `shouldYieldFocus()` and `verify()`. When building an `InputVerifier` subclass only the `verify()` method need be implemented, as it is the only abstract method. The `shouldYieldFocus()` method automatically calls the `verify()` method to perform the check.



Figure 11.9
Example demonstrating the use of custom Formats with `JFormattedTextField`, and the use of `InputVerifier` to control focus transfer based on content validation

Example 11.5

TextDemo.java

see \Chapter11\5

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.text.*;

import dl.*;

public class TextDemo extends JFrame {
    protected JFormattedTextField m_firstTxt;
    protected JFormattedTextField m_lastTxt;
    protected JFormattedTextField m_phoneTxt;
    protected JFormattedTextField m_faxTxt;
    protected JPasswordField m_passwordTxt;
    protected JTextArea m_commentsTxt;
    protected JLabel m_status;

    public static final String PHONE_PATTERN = "(###) ###-####";

    public TextDemo() {
        super("Text Components Demo");
        Font monospaced = new Font("Monospaced", Font.PLAIN, 12);
        JPanel pp = new JPanel(new BorderLayout());

        JPanel p = new JPanel(new DialogLayout2());
        p.setBorder(new EmptyBorder(10, 10, 10, 10));
        p.add(new JLabel("First name:"));
        m_firstTxt = new JFormattedTextField(
            new NameFormat());
        m_firstTxt.setInputVerifier(new TextVerifier(
            "First name cannot be empty"));
        m_firstTxt.setColumns(12);
        p.add(m_firstTxt);

        p.add(new JLabel("Last name:"));
        m_lastTxt = new JFormattedTextField(
            new NameFormat());
        m_lastTxt.setColumns(12);
        p.add(m_lastTxt);

        p.add(new JLabel("Phone number:"));
        MaskFormatter formatter = null;
        try {
            formatter = new MaskFormatter(PHONE_PATTERN);
        }
        catch (ParseException pex) {
```

1

First and last name input fields are now formatted text fields with `NameFormat` instances regulating formatting

2

Formatted text fields using a `MaskFormatter` for phone number input

```

    pex.printStackTrace();
}
m_phoneTxt = new JFormattedTextField(formatter);
m_phoneTxt.setColumns(12);
m_phoneTxt.setInputVerifier(new FTFVerifier(
    "Phone format is "+PHONE_PATTERN));
p.add(m_phoneTxt);

p.add(new JLabel("Fax number:"));
m_faxTxt = new JFormattedTextField(
    new Phoneformat());
m_faxTxt.setColumns(12);
m_faxTxt.setInputVerifier(newFTFVerifier(
    "Fax format is "+PHONE_PATTERN));
p.add(m_faxTxt);

p.add(new JLabel("Login password:"));
m_passwordTxt = new JPasswordField(20)
m_passwordTxt.setFont(monospaced);
m_passwordTxt.setInputVerifier(new TextVerifier(
    "Login password cannot be empty"));
p.add(m_passwordTxt);

p.setBorder(new CompoundBorder(
    new TitledBorder(new EtchedBorder(), "Personal Data"),
    new EmptyBorder(1, 5, 3, 5)
));
pp.add(p, BorderLayout.NORTH);

m_commentsTxt = new JTextArea("", 4, 30);
m_commentsTxt.setFont(monospaced);
m_commentsTxt.setLineWrap(true);
m_commentsTxt.setWrapStyleWord(true);
p = new JPanel(new BorderLayout());
p.add(new JScrollPane(m_commentsTxt));
p.setBorder(new CompoundBorder(
    new TitledBorder(new EtchedBorder(), "Comments"),
    new EmptyBorder(3, 5, 3, 5)
));
pp.add(p, BorderLayout.CENTER);

m_status = new JLabel("Input data");
m_status.setBorder(new CompoundBorder(
    new EmptyBorder(2, 2, 2, 2),
    new SoftBevelBorder(SoftBevelBorder.LOWERED)));
pp.add(m_status, BorderLayout.SOUTH);
Dimension d = m_status.getPreferredSize();
m_status.setPreferredSize(new Dimension(150, d.height));

pp.setBorder(new EmptyBorder(5, 5, 5,5))
getContentPane().add(pp);
pack();
}

public static void main(String[] args) {

```

2 Formatted text fields using a **MaskFormatter** for phone number input

3 Formatted text fields using a **PhoneFormat** instance for fax number input

4 Custom **InputVerifier** added to the password field to enforce nonempty password

5 Label used as a status bar

```

JFrame frame = new TextDemo();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true)
}

/**
 *Format to capitalize all words
 */
class NameFormat extends Format {
    public StringBuffer format(Object obj, StringBuffer toAppendTo,
        FieldPosition fieldPosition) {
        fieldPosition.setBeginIndex(toAppendTo.length());
        String str = obj.toString();
        char prevCh = ' ';
        for (int k=0; k<str.length(); k++) {
            char nextCh = str.charAt(k);
            if (Character.isLetter(nextCh) && prevCh == ' ')
                nextCh = Character.toUpperCase(nextCh);
            toAppendTo.append(nextCh);
            prevCh = nextCh;
        }
        fieldPosition.setEndIndex(toAppendTo.length());
        return toAppendTo;
    }
}

```

6 Custom Format to capitalize each word separated by a space

```

/**
 *Format phone numbers
 */
class PhoneFormat extends Format {
    public StringBuffer format(Object obj, StringBuffer toAppendTo,
        FieldPosition fieldPosition) {
        fieldPosition.setBeginIndex(toAppendTo.length());

        // Get digits of the number
        String str = obj.toString();
        StringBuffer number = new StringBuffer();
        for (int k=0; k<str.length(); k++) {
            char nextCh = str.charAt(k);
            if (Character.isDigit(nextCh)) {
                number.append(nextCh);
            } else if (Character.isLetter(nextCh)) {
                nextCh = Character.toUpperCase(nextCh);
                switch (nextCh) {
                    case 'A':
                    case 'B':
                    case 'C':
                        number.append('2');
                        break;
                    case 'D':
                    case 'E':
                    case 'F':
                        number.append('3');
                        break;
                }
            }
        }
    }
}

```

7 Custom Format for phone numbers allowing extension and converting letter characters to their digit equivalents

```

        case 'G':
        case 'H':
        case 'I':
            number.append('4');
            break;
        case 'J':
        case 'K':
        case 'L':
            number.append('5');
            break;
        case 'M':
        case 'N':
        case 'O':
            number.append('6');
            break;
        case 'P':
        case 'Q':
        case 'R':
        case 'S':
            number.append('7');
            break;
        case 'T':
        case 'U':
        case 'V':
            number.append('8');
            break;
        case 'W':
        case 'X':
        case 'Y':
        case 'Z':
            number.append('9');
            break;
    }
}

// Format digits according to the pattern
int index = 0
for (int k=0; k<PHONE_PATTERN.length(); k++) {
    char ch = PHONE_PATTERN.charAt(k);
    if (ch == '#') {
        if (index >=number.length())
            break;
        toAppendTo.append(number.charAt(index++));
    }
    else
        toAppendTo.append(ch);
}

fieldPosition.setEndIndex(toAppendTo.length());
return toAppend(ch);
}

```

```

public Object parseObject(String text, ParsePosition pos) {
    pos.setIndex(pos.getIndex()+text.length());
    return text;
}
}

```

```

/**
 * Verify input to JTextField
 */
class TextVerifier extends InputVerifier {
    private String m_errMsg;

    public TextVerifier(String errMsg) {
        m_errMsg = errMsg;
    }

    public boolean verify(JComponent input) {
        m_status.setText("");
        if (!input instanceof JTextField)
            return true;
        JTextField txt = (JTextField)input;
        String str = txt.getText();
        if (str.length() == 0) {
            m_status.setText(m_errMsg);
            return false;
        }
        return true;
    }
}

```

8

Input Verifier to enforce nonempty text fields

```

/**
 * Verify input to JFormattedTextField
 */
class FTFVerifier extends InputVerifier {
    private String m_errMsg;

    public FTFVerifier(String errMsg) {
        m_errMsg = errMsg;
    }

    public boolean verify(JComponent input) {
        m_status.setText("");
        if (!input instanceof JFormattedTextField)
            return true;
        JFormattedTextField ftf = (JFormattedTextField)input;
        JFormattedTextField.AbstractFormatter formatter =
            ftf.getFormatter();
        if (formatter == null)
            return true;
        try {
            formatter.toStringValue(ftf.getText());
            return true;
        }
        catch (ParseException pe) {
            m_status.setText(m_errmsg);
        }
    }
}

```

9

Input Verifier to enforce validation against current formatter

```

        return false;
    }
}
}
}

```

11.5.2 Understanding the code

Class TextDemo

This example extends the `TextDemo` example from section 11.3. The following changes have been made:

- ❶ • `m_firstTxt` and `m_lastTxt` are now `JFormattedTextFields` with an instance of our custom `NameFormat` class attached as the `Format`. Also, `m_firstTxt` receives an instance of our `TextVerifier` as an `InputVerifier`.
- ❷ • `JFormattedTextField m_phoneTxt` has been added for phone number input. This component's `Format` is an instance of `MaskFormatter` with phone number mask `PHONE_PATTERN`. Also, `m_phoneTxt` receives an instance of our custom `FTFVerifier` as an `InputVerifier`.
- ❸ • `JFormattedTextField m_faxTxt` has been added to allow input of a fax number. Unlike `m_phoneTxt`, this component's `Format` is an instance of our custom `PhoneFormat` class.
- ❹ • `JPasswordField m_passwordTxt` receives an instance of `TextVerifier` as an `InputVerifier`.
- ❺ • `JLabel m_status` has been added to the bottom of the frame to display input errors in formatted fields.

❻ *Class NameFormat*

The purpose of this custom `Format` is to capitalize all words in an input string. The `format()` method splits the input string into space-separated words and replaces the first letter of each word by its capitalized equivalent one. Note how the `FieldPosition` parameter is used.

❼ *Class PhoneFormat*

This custom `Format` presents an alternative to using `MaskFormatter` to format phone numbers. The advantages `PhoneFormat` provides are:

- Does not always display empty mask: "() - " in our case.
- Allows input of various lengths to allow for telephone extensions, for instance. (This can be viewed as either an advantage or disadvantage, depending on your situation.)
- Replaces letter characters in a phone number with the corresponding digits (anyone who deals with 1-800-numbers will appreciate this).

❽ *Class TextVerifier*

This class extends `InputVerifier` to verify that the input in a `JTextField` is not empty. If it is empty, this verifier does not allow focus to leave the `JTextField` and displays an error message (provided in the constructor) in the status bar.

9 *Class FTFVerifier*

This class extends `InputVerifier` to verify that input in a `JFormattedTextField` can be formatted by its associated formatter. If a formatting error occurs, this verifier does not allow the focus to leave the `JFormattedTextField` and displays an error message (provided in the constructor) in the status bar.

BUG ALERT! From another application such as a text editor, try copying the string “1234567890” into the clipboard (a 10-digit string). Then, position the text cursor in the phone number field as far left as it will go and paste into the field. You will see “(123) 456-789”. The last digit is left off, even though you can type it in manually. The behavior of this has something to do with the number of “filler” characters in the mask, but we did not dig deep enough to figure out the exact relationship. Thanks to David Karr for pointing this out.

11.6 *FORMATTED SPINNER EXAMPLE*

This example demonstrates how to apply formatting to a `JSpinner` component (a new component added to Java 1.4, covered in chapter 10). `JSpinner` does not extend `JTextComponent`. However, some of its editors (see section 10.6) contain a `JFormattedTextField` component within, allowing us to assign a `Format` instance to them to manage spinner input and display.

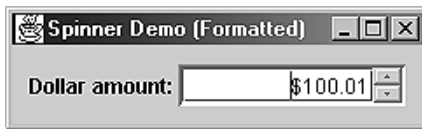


Figure 11.10
Formatted `JSpinner` example

Example 11.6

FormattedSpinnerDemo.java

```
see \Chapter11\6
import java.awt.*;
import java.text.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.text.*;

class FormattedSpinnerDemo extends JFrame {

    public FormattedSpinnerDemo() {
        super("Spinner Demo (Formatted)");

        JPanel p = new JPanel();
        p.setLayout(new BorderLayout(p, BorderLayout.X_AXIS))
        p.setBorder(new EmptyBorder(10, 10, 10, 10));
        p.add(new JLabel("Dollar amount: "));
```

```

SpinnerModel model = new SpinnerNumberModel(
    new Double(100.01),
    new Double(0),
    null,
    new Double(20)
);
JSpinner spn = new JSpinner(model);
JFormattedTextField ftf = ((JSpinner.DefaultEditor)spn.
    getEditor()).getTextField();
ftf.setColumns(10);

NumberFormatter nf = new NumberFormatter(
    NumberFormat.getCurrencyInstance(Locale.US));
DefaultFormatterFactory dff = new DefaultFormatterFactory();
dff.setDefaultFormatter(nf);
dff.setDisplayFormatter(nf);
dff.setEditFormatter(nf);
ftf.setFormatterFactory(dff);

p.add(spn);

getContentPane().add(p, BorderLayout.NORTH);
pack();
}

public static void main( String args[] ) {
    FormattedSpinnerDemo mainFrame = new FormattedSpinnerDemo();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.setVisible(true);
}
}

```

1 Obtain a reference to JSpinner's formatted text field

11.6.1 Understanding the code

Class FormattedSpinnerDemo

This class extends `JFrame` to implement the frame container for this example. A `JSpinner` is created with a `SpinnerNumberModel`. Therefore this spinner will use a `JSpinner.NumberEditor` as its editor. We know from section 10.6 that this editor contains a `JFormattedTextField` component. In order to access this `JFormattedTextField` instance, we obtain the editor with `JSpinner`'s `getEditor()` method, and then call `getTextField()`, which gives us a reference to the `JFormattedTextField`.

It turns out there is no simple method to assign a `Format` instance to the existing `JFormattedTextField` component within a `JSpinner`'s editor. We have to create a `DefaultFormatterFactory` instance, set our `NumberFormatter` as the default, display, and edit formatters, and then call the `JFormattedTextField`'s `setFormatterFactory()` method.

11.6.2 Running the code

Figure 11.10 shows our example in action. By accessing `JSpinner`'s `JFormattedTextField` and assigning it a new `Format`, we are able to create a spinner for selection/input of a U.S. dollar amount.

11.7 UNDO/REDO

Undo/redo options are commonplace in applications such as paint programs and word processors, and they have been used extensively throughout the writing of this book. It is interesting that this functionality is provided as part of the Swing library, as it is completely Swing independent. In this section we will briefly introduce the `javax.swing.undo` constituents and, in the process of doing so, we will present an example showing how undo/redo functionality can be integrated into any type of application. The text components come with built-in undo/redo functionality, and we will also discuss how to take advantage of this.

11.7.1 The UndoableEdit interface

abstract interface javax.swing.undo.UndoableEdit

This interface acts as a template definition for anything that can be undone/redone. Implementations should normally be very lightweight, as undo/redo operations commonly occur quickly in succession.

`UndoableEdits` are designed to have three states: undoable, redoable, and dead. When an `UndoableEdit` is in the undoable state, calling `undo()` will perform an undo operation. Similarly, when an `UndoableEdit` is in the redoable state, calling `redo()` will perform a redo operation. The `canUndo()` and `canRedo()` methods provide ways to see whether an `UndoableEdit` is in the undoable or redoable state. We can use the `die()` method to explicitly send an `UndoableEdit` to the dead state. In the dead state, an `UndoableEdit` cannot be undone or redone, and any attempt to do so will generate an exception.

`UndoableEdits` maintain three `String` properties, which are normally used as menu item text: `presentationName`, `undoPresentationName`, and `redoPresentationName`. The `addEdit()` and `replaceEdit()` methods are meant to be used to merge two edits and replace an edit, respectively. `UndoableEdit` also defines the concept of significant and insignificant edits. An insignificant edit is one that `UndoManager` (see section 11.7.6) ignores when an undo/redo request is made. `CompoundEdit` (see section 11.7.3), however, will pay attention to both significant and insignificant edits. The `significant` property of an `UndoableEdit` can be queried with `isSignificant()`.

11.7.2 AbstractUndoableEdit

class javax.swing.undo.AbstractUndoableEdit

`AbstractUndoableEdit` implements `UndoableEdit` and defines two boolean properties that represent the three `UndoableEdit` states. The `alive` property is `true` when an edit is not dead. The `done` property is `true` when an undo can be performed, and `false` when a redo can be performed.

The default behavior provided by this class is good enough for most subclasses. All `AbstractUndoableEdits` are *significant*, and the `undoPresentationName` and `redoPresentationName` properties are formed by simply appending “Undo” and “Redo” to `presentationName`.

The following example demonstrates a basic square painting program with undo/redo functionality. This application simply draws a square outline wherever a mouse press occurs. A `Vector` of `Points` is maintained which represents the upper left-hand corner of each square

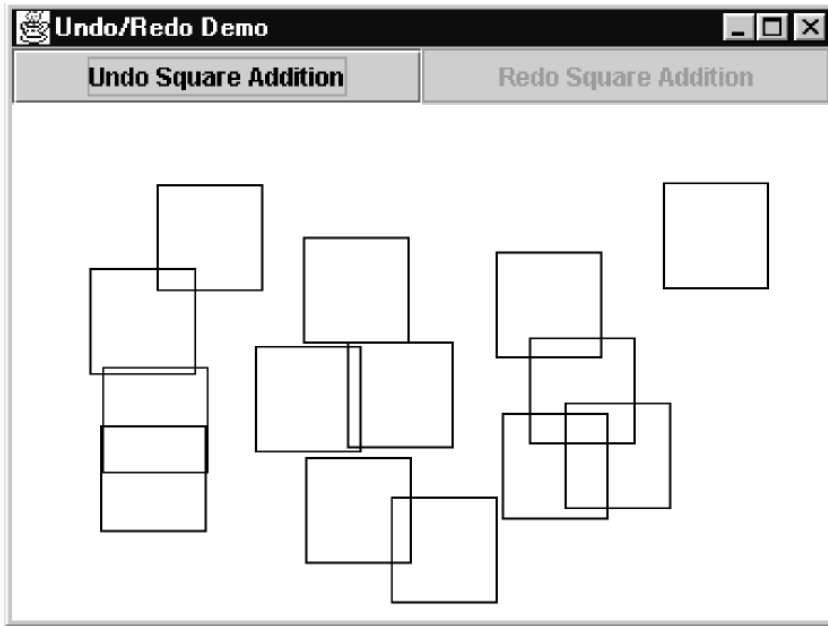


Figure 11.11 A square painting application with one level of undo/redo

that is drawn on the canvas. We create an `AbstractUndoableEdit` subclass to maintain a reference to a `Point`, with `undo()` and `redo()` methods that remove and add that `Point` from the `Vector`. Figure 11.11 illustrates the output of example 11.7.

Example 11.7

UndoRedoPaintApp.java

see \Chapter11\7

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

public class UndoRedoPaintApp extends JFrame
{
    protected Vector m_points = new Vector();
    protected PaintCanvas m_canvas = new PaintCanvas(m_points);
    protected UndoablePaintSquare m_edit;
    protected JButton m_undoButton = new JButton("Undo");
    protected JButton m_redoButton = new JButton("Redo");

    public UndoRedoPaintApp() {
        super("Undo/Redo Demo");
    }
}
```

```

m_undoButton.setEnabled(false);
m_redoButton.setEnabled(false);

JPanel buttonPanel = new JPanel(new GridLayout());
buttonPanel.add(m_undoButton);
buttonPanel.add(m_redoButton);

getContentPane().add(buttonPanel, BorderLayout.NORTH);
getContentPane().add(m_canvas, BorderLayout.CENTER);

m_canvas.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        Point point = new Point(e.getX(), e.getY());
        m_points.addElement(point);
        m_edit = new UndoablePaintSquare(point, m_points);
        m_undoButton.setText(m_edit.getUndoPresentationName());
        m_redoButton.setText(m_edit.getRedoPresentationName());
        m_undoButton.setEnabled(m_edit.canUndo());
        m_redoButton.setEnabled(m_edit.canRedo());
        m_canvas.repaint();
    }
});

m_undoButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try { m_edit.undo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
        m_undoButton.setEnabled(m_edit.canUndo());
        m_redoButton.setEnabled(m_edit.canRedo());
    }
});

m_redoButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try { m_edit.redo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
        m_undoButton.setEnabled(m_edit.canUndo());
        m_redoButton.setEnabled(m_edit.canRedo());
    }
});

setSize(400,300);
setVisible(true);
}

public static void main(String argv[]) {
    new UndoRedoPaintApp();
}

}

class PaintCanvas extends JPanel
{
    Vector m_points;
    protected int width = 50;

```

```

protected int height = 50;

public PaintCanvas(Vector vect) {
    super();
    m_points = vect;
    setOpaque(true);
    setBackground(Color.white);
}

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    Enumeration enum = m_points.elements();
    while(enum.hasMoreElements()) {
        Point point = (Point) enum.nextElement();
        g.drawRect(point.x, point.y, width, height);
    }
}

class UndoablePaintSquare extends AbstractUndoableEdit
{
    protected Vector m_points;
    protected Point m_point;

    public UndoablePaintSquare(Point point, Vector vect) {
        m_points = vect;
        m_point = point;
    }

    public String getPresentationName() {
        return "Square Addition";
    }

    public void undo() {
        super.undo();
        m_points.remove(m_point);
    }

    public void redo() {
        super.redo();
        m_points.add(m_point);
    }
}

```

One thing to note about example 11.7 is that it is extremely limited. Because we are not maintaining an ordered collection of `UndoableEdits`, we can only perform one undo/redo. `CompoundEdit` and `UndoManager` directly address this limitation.

11.7.3 CompoundEdit

class javax.swing.undo.CompoundEdit

This class extends `AbstractUndoableEdit` to support an ordered collection of `UndoableEdits`, which are maintained as a protected `Vector` called `edits`. `UndoableEdits` can be

added to this vector with `addEdit()`, but they cannot so easily be removed (for this, a subclass would be necessary).

Even though `CompoundEdit` is more powerful than `AbstractUndoableEdit`, it is far from the ideal solution. Edits cannot be undone until all edits have been added. Once all `UndoableEdits` are added, we are expected to call `end()`, at which point `CompoundEdit` will no longer accept any additional edits. Once `end()` is called, a call to `undo()` will undo all edits, whether they are *significant* or not. A `redo()` will then redo them all, and we can continue to cycle back and forth like this as long as the `CompoundEdit` itself remains alive. For this reason, `CompoundEdit` is useful for a predefined or intentionally limited set of states.

`CompoundEdit` introduces an additional state property called `inProgress`, which is true if `end()` has not been called. We can retrieve the value of `inProgress` with `isInProgress()`. The `significant` property, inherited from `UndoableEdit`, will be true if one or more of the contained `UndoableEdits` is significant, and it will be false otherwise.

11.7.4 UndoableEditEvent

class javax.swing.event.UndoableEditEvent

This event encapsulates a source `Object` and an `UndoableEdit`, and it is meant to be passed to implementations of the `UndoableEditListener` interface.

11.7.5 The UndoableEditListener interface

class javax.swing.event.UndoableEditListener

This listener is intended for use by any class wishing to listen for operations that can be undone/redone. When such an operation occurs, an `UndoableEditEvent` can be sent to an `UndoableEditListener` for processing. `UndoManager` implements this interface so we can simply add it to any class that defines undoable/redone operations. It is important to emphasize that `UndoableEditEvents` are not fired when an undo or redo actually occurs, but when an operation occurs which has an `UndoableEdit` associated with it. This interface declares one method, `undoableEditHappened()`, which accepts an `UndoableEditEvent`. We are generally responsible for passing `UndoableEditEvents` to this method. Example 11.8 in the next section demonstrates this.

11.7.6 UndoManager

class javax.swing.undo.UndoManager

`UndoManager` extends `CompoundEdit` and relieves us of the limitation where undos and redos cannot be performed until `edit()` is called. It also relieves us of the limitation where all edits are undone or redone at once. Another major difference from `CompoundEdit` is that `UndoManager` simply skips over all insignificant edits when `undo()` or `redo()` is called, effectively not paying them any attention. Interestingly, `UndoManager` allows us to add edits while `inProgress` is true, but if `end()` is ever called, `UndoManager` immediately starts acting like a `CompoundEdit`.

`UndoManager` introduces a new state called `undoOrRedo` which, when true, signifies that calling `undo()` or `redo()` is valid. This property can only be true if there is more than

one edit stored, and only if there is at least one edit in the undoable state and one in the redoable state. The value of this property can be retrieved with `canUndoOrRedo()`, and the `getUndoOrRedoPresentationName()` method will return an appropriate name for use in a menu item or elsewhere.

We can retrieve the next significant `UndoableEdit` that is scheduled to be undone or redone with `editToBeUndone()` or `editToBeRedone()`. We can kill all stored edits with `discardAllEdits()`. The `redoTo()` and `undoTo()` methods can be used to programmatically invoke `undo()` or `redo()` on all edits from the current edit to the edit that is provided as parameter.

We can set the maximum number of edits that can be stored with `setLimit()`. The value of the `limit` property (100 by default) can be retrieved with `getLimit()`, and if it is set to a value smaller than the current number of edits, the edits will be reduced using the protected `trimForLimit()` method. Based on the index of the current edit within the `edits` vector, this method will attempt to remove the most balanced number of edits, in undoable and redoable states, as it can in order to achieve the given limit. The further away an edit is (based on its vector index in the `edits` vector), the more of a candidate it is for removal when a trim occurs, as edits are taken from the extreme ends of the `edits` vector.

It is very important to note that when an edit is added to the `edits` vector, all edits in the redoable state (those appearing after the index of the current edit) do not simply get moved up one index. Rather, they are removed. So, for example, suppose in a word processor application you enter some text, change the style of ten different regions of that text, and then undo the five most recent style additions. Then a new style change is made. The first five style changes that were made remain in the undoable state, and the new edit is added, also in the undoable state. However, the five style changes that were undone (moved to the redoable state) are now completely lost.

NOTE All public `UndoManager` methods are synchronized to enable thread safety, and to make `UndoManager` a good candidate for use as a central undo/redo manager for any number of functionalities.

Example 11.8 shows how we can modify our `UndoRedoPaintApp` example to allow multiple undos and redos using an `UndoManager`. Because `UndoManager` implements `UndoableEditListener`, we should normally add `UndoableEditEvents` to it using the `undoableEditHappened()` method rather than `addEdit()`—`undoableEditHappened()` calls `addEdit()` for us, and at the same time allows us to keep track of the source of the operation. This enables `UndoManager` to act as a central location for all undo/redo edits in an application.

Example 11.8

UndoRedoPaintApp.java

```
see \Chapter11\8
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.undo.*;
```

```

import javax.swing.event.*;

public class UndoRedoPaintApp extends JFrame
{
    protected Vector m_points = new Vector();
    protected PaintCanvas m_canvas = new PaintCanvas(m_points);
    protected UndoManager m_undoManager = new UndoManager();
    protected JButton m_undoButton = new JButton("Undo");
    protected JButton m_redoButton = new JButton("Redo");

    public UndoRedoPaintApp() {
        super("Undo/Redo Demo");

        m_undoButton.setEnabled(false);
        m_redoButton.setEnabled(false);

        JPanel buttonPanel = new JPanel(new GridLayout());
        buttonPanel.add(m_undoButton);
        buttonPanel.add(m_redoButton);

        getContentPane().add(buttonPanel, BorderLayout.NORTH);
        getContentPane().add(m_canvas, BorderLayout.CENTER);

        m_canvas.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                Point point = new Point(e.getX(), e.getY());
                m_points.addElement(point);

                m_undoManager.undoableEditHappened(new UndoableEditEvent(m_canvas,
                    new UndoablePaintSquare(point, m_points));

                m_undoButton.setText(m_undoManager.getUndoPresentationName());
                m_redoButton.setText(m_undoManager.getRedoPresentationName());
                m_undoButton.setEnabled(m_undoManager.canUndo());
                m_redoButton.setEnabled(m_undoManager.canRedo());
                m_canvas.repaint();
            }
        });

        m_undoButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { m_undoManager.undo(); }
                catch (CannotRedoException cre) { cre.printStackTrace(); }
                m_canvas.repaint();
                m_undoButton.setEnabled(m_undoManager.canUndo());
                m_redoButton.setEnabled(m_undoManager.canRedo());
            }
        });

        m_redoButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { m_undoManager.redo(); }
                catch (CannotRedoException cre) { cre.printStackTrace(); }
                m_canvas.repaint();
                m_undoButton.setEnabled(m_undoManager.canUndo());
                m_redoButton.setEnabled(m_undoManager.canRedo());
            }
        });
    }
}

```

```

    });

    setSize(400,300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
}

public static void main(String argv[]) {
    new UndoRedoPaintApp();
}
}

// Classes PaintCanvas and UndoablePaintSquare are unchanged
// from example 11.7

```

Run this example and notice that we can have up to 100 squares in the undoable or redoable state at any given time. Also notice that when several squares are in the redoable state, adding a new square will eliminate them, and the redo button will become disabled, indicating that no redos can be performed.

11.7.7 The StateEditable interface

abstract interface javax.swing.undo.StateEditable

The `StateEditable` interface is intended to be used by objects that wish to maintain specific *before* (pre) and *after* (post) states. This provides an alternative to managing undos and redos in `UndoableEdits`. Once a before and after state is defined, we can use a `StateEdit` object to switch between the two states. Two methods must be implemented by `StateEditable` implementations. `storeState()` is to be used by an object to store its state as a set of key/value pairs in a given `Hashtable`. Normally this entails storing the name of an object and a copy of that object (unless a primitive is stored). `restoreState()` is to be used by an object to restore its state according to the key/value pairs stored in a given `Hashtable`.

11.7.8 StateEdit

class javax.swing.undo.StateEdit

`StateEdit` extends `AbstractUndoableEdit`, and it is meant to store the before and after `Hashtables` of a `StateEditable` instance. When a `StateEdit` is instantiated, it is passed a `StateEditable` object, and a protected `Hashtable` called `preState` is passed to that `StateEditable`'s `storeState()` method. Similarly, when `end()` is called on a `StateEdit`, a protected `Hashtable` called `postState` is passed to the corresponding `StateEditable`'s `storeState()` method. After `end()` is called, undos and redos toggle the state of the `StateEditable` between `postState` and `preState` by passing the appropriate `Hashtable` to that `StateEditable`'s `restoreState()` method.

11.7.9 UndoableEditSupport

class javax.swing.undo.UndoableEditSupport

This convenience class is used for managing UndoableEditListeners. We can add and remove an UndoableEditListener with `addUndoableEditListener()` and `removeUndoableEditListener()`. UndoableEditSupport maintains an `updateLevel` property which specifies how many times the `beginUpdate()` method has been called. As long as this value is above 0, UndoableEdits added with the `postEdit()` method will be stored in a temporary CompoundEdit object without being fired. The `endEdit()` method decrements the `updateLevel` property. When `updateLevel` is 0, any calls to `postEdit()` will fire the edit that is passed in, or the CompoundEdit that has been accumulating edits up to that point.

WARNING The `endUpdate()` and `beginUpdate()` methods may call `undoableEditHappened()` in each UndoableEditListener, possibly resulting in deadlock if these methods are actually invoked from one of the listeners themselves.

11.7.10 CannotUndoException

class javax.swing.undo.CannotUndoException

This exception is thrown when `undo()` is invoked on an UndoableEdit that cannot be undone.

11.7.11 CannotRedoException

class javax.swing.undo.CannotRedoException

This exception is thrown when `redo()` is invoked on an UndoableEdit that cannot be redone.

11.7.12 Using built-in text component undo/redo functionality

All default text component Document models fire UndoableEdits. For PlainDocuments, this involves keeping track of text insertions and removals, as well as any structural changes. For StyledDocuments, however, this involves keeping track of a much larger group of changes. Fortunately this work has been built into these document models for us. The following example, 11.9, shows how easy it is to add undo/redo support to text components. Figure 11.9 illustrates the output.

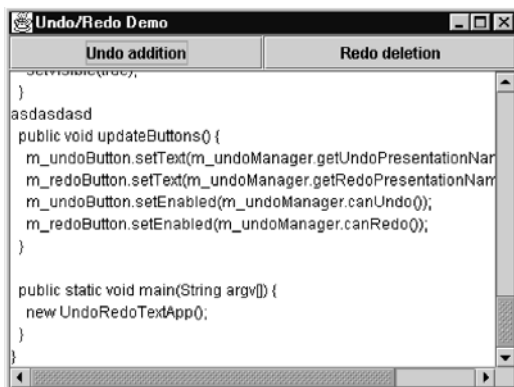


Figure 11.12
Undo/redo functionality
added to a JTextArea

Example 11.9

UndoRedoTextApp.java

see \Chapter11\9

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

public class UndoRedoTextApp extends JFrame
{
    protected JTextArea m_editor = new JTextArea();
    protected UndoManager m_undoManager = new UndoManager();
    protected JButton m_undoButton = new JButton("Undo");
    protected JButton m_redoButton = new JButton("Redo");

    public UndoRedoTextApp() {
        super("Undo/Redo Demo");

        m_undoButton.setEnabled(false);
        m_redoButton.setEnabled(false);

        JPanel buttonPanel = new JPanel(new GridLayout());
        buttonPanel.add(m_undoButton);
        buttonPanel.add(m_redoButton);

        JScrollPane scroller = new JScrollPane(m_editor);

        getContentPane().add(buttonPanel, BorderLayout.NORTH);
        getContentPane().add(scroller, BorderLayout.CENTER);

        m_editor.getDocument().addUndoableEditListener(
            new UndoableEditListener() {
                public void undoableEditHappened(UndoableEditEvent e) {
                    m_undoManager.addEdit(e.getEdit());
                    updateButtons();
                }
            });

        m_undoButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { m_undoManager.undo(); }
                catch (CannotRedoException cre) { cre.printStackTrace(); }
                updateButtons();
            }
        });

        m_redoButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try { m_undoManager.redo(); }
                catch (CannotRedoException cre) { cre.printStackTrace(); }
                updateButtons();
            }
        });
    }
}
```

```
    });  
  
    setSize(400,300);  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    setVisible(true);  
}  
  
public void updateButtons() {  
    m_undoButton.setText(m_undoManager.getUndoPresentationName());  
    m_redoButton.setText(m_undoManager.getRedoPresentationName());  
    m_undoButton.setEnabled(m_undoManager.canUndo());  
    m_redoButton.setEnabled(m_undoManager.canRedo());  
}  
  
public static void main(String argv[]) {  
    new UndoRedoTextApp();  
}  
}
```