CHAPTER 20

Constructing an HTML Editor Application

20.1 HTML editor, part I: introducing HTML 635
20.2 HTML editor, part II: managing fonts 642
20.3 HTML editor, part III: document properties 650
20.4 HTML editor, part IV: working with HTML styles and tables 667
20.5 HTML editor, part V: clipboard and undo/redo 677
20.6 HTML editor, part VI: advanced font management 682
20.7 HTML editor, part VII: find and replace 695
20.8 HTML editor, part IX: spell checker (using JDBC and SQL) 708

This chapter is devoted to the construction of a fully functional HTML editor application. The examples in this chapter demonstrate practical applications of many of the topics covered in chapter 19. The main focus throughout is working with styled text documents, and the techniques discussed here can be applied to almost any styled text editor.

NOTE Chapter 20 in the first edition of this book was titled "Constructing a word processor" and was devoted to the construction of a fully functional RTF word processor application. This chapter and examples from the first edition remain freely available at www.manning.com/sbe.

20.1 HTML EDITOR, PART I: INTRODUCING HTML

In this section we introduce an example demonstrating the use of `JTextPane` and `HTMLEditorKit` to display and edit HTML documents. The following features are included:

- Creating a new HTML document
- Opening an existing HTML document
- Saving changes
- Saving the document under a new name/location
- Prompting the user to save changes before loading a new document or exiting the application

This example serves as the foundation for our HTML editor application that will be expanded upon throughout this chapter.

NOTE The Swing HTML package supports HTML 3.2 but not 4.0. Support for 4.0 has been deferred to a future release of J2SE.

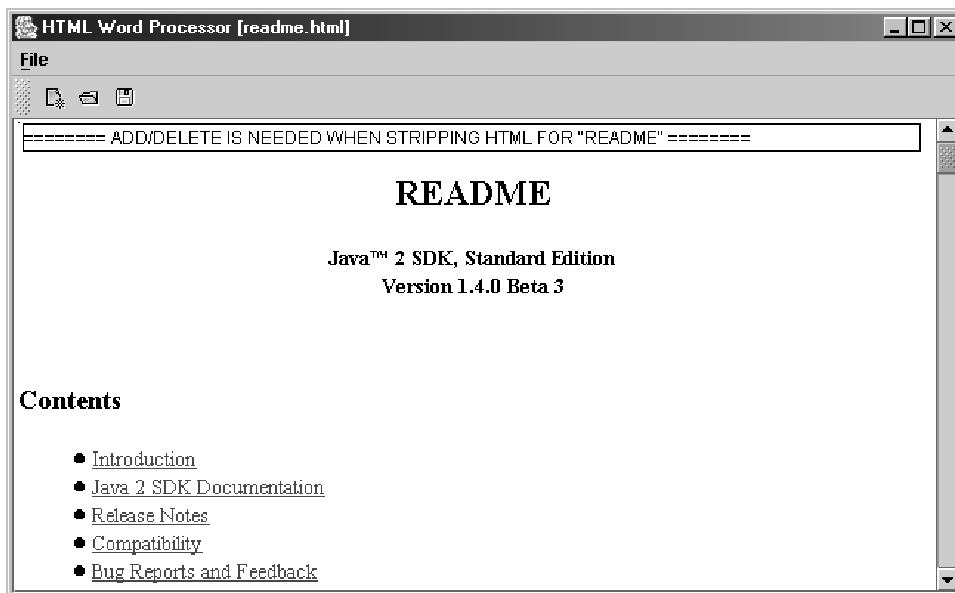

Figure 20.1 **`JTextPane` displaying an HTML document**

Example 20.1

HtmlProcessor.java

see \Chapter20\1

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

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

public class HtmlProcessor extends JFrame {

   public static final String APP_NAME = "HTML Word Processor";

   protected JTextPane m_editor;
   protected StyleSheet m_context;
   protected HTMLDocument m_doc;
   protected HTMLEditorKit m_kit;
   protected SimpleFilter m_htmlFilter;
   protected JToolBar m_toolBar;

   protected JFileChooser m_chooser;
   protected File m_currentFile;

   protected boolean m_textChanged = false;

   public HtmlProcessor() {
      super(APP_NAME);
      setSize(650, 400);

      m_editor = new JTextPane();
      m_kit = new HTMLEditorKit();
      m_editor.setEditorKit(m_kit);

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

      JMenuBar menuBar = createMenuBar();
      setJMenuBar(menuBar);

      m_chooser = new JFileChooser();
      m_htmlFilter = new SimpleFilter("html", "HTML Documents");
      m_chooser.setFileFilter(m_htmlFilter);
      try {
         File dir = (new File(".")).getCanonicalFile();
         m_chooser.setCurrentDirectory(dir);
      } catch (IOException ex) {}

      newDocument();

      WindowListener wndCloser = new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            if (!promptToSave())
               return;
            System.exit(0);
         }
         public void windowActivated(WindowEvent e) {
            m_editor.requestFocus();
         }
      };
      addWindowListener(wndCloser);
   }
```

1 Ensure user has opportunity to save changes before closing

```java
protected JMenuBar createMenuBar() {
  JMenuBar menuBar = new JMenuBar();

  JMenu mFile = new JMenu("File");
  mFile.setMnemonic('f');

  ImageIcon iconNew = new ImageIcon("New16.gif");
  Action actionNew = new AbstractAction("New", iconNew) {
    public void actionPerformed(ActionEvent e) {
      if (!promptToSave())
        return;
      newDocument();
    }
  };
  JMenuItem item = new JMenuItem(actionNew);
  item.setMnemonic('n');
  item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_N, InputEvent.Ctrl_MASK));
  mFile.add(item);

  ImageIcon iconOpen = new ImageIcon("Open16.gif");
  Action actionOpen = new AbstractAction("Open...", iconOpen) {
    public void actionPerformed(ActionEvent e) {
      if (!promptToSave())
        return;
      openDocument();
    }
  };
  item = new JMenuItem(actionOpen);
  item.setMnemonic('o');
  item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_O, InputEvent.Ctrl_MASK));
  mFile.add(item);

  ImageIcon iconSave = new ImageIcon("Save16.gif");
  Action actionSave = new AbstractAction("Save", iconSave) {
    public void actionPerformed(ActionEvent e) {
      saveFile(false);
    }
  };
  item = new JMenuItem(actionSave);
  item.setMnemonic('s');
  item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_S, InputEvent.Ctrl_MASK));
  mFile.add(item);

  ImageIcon iconSaveAs = new ImageIcon("SaveAs16.gif");
  Action actionSaveAs =
  new AbstractAction("Save As...", iconSaveAs) {
    public void actionPerformed(ActionEvent e) {
      saveFile(true);
    }
  };
  item = new JMenuItem(actionSaveAs);
```

2 **Creates a menu bar with New, Open, Save, Save As, and Exit menu items**

```
    item.setMnemonic('a');
    mFile.add(item);

    mFile.addSeparator();

    Action actionExit = new AbstractAction("Exit") {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };

    item =mFile.add(actionExit);
    item.setMnemonic('x');
    menuBar.add(mFile);

    m_toolBar = new JToolBar();
    JButton bNew = new SmallButton(actionNew,
     "New document");
    m_toolBar.add(bNew);

    JButton bOpen = new SmallButton(actionOpen,
     "Open HTML document");
    m_toolBar.add(bOpen);

    JButton bSave = new SmallButton(actionSave,
     "Save HTML document");
    m_toolBar.add(bSave);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  protected String getDocumentName() {
    return m_currentFile==null ? "Untitled" :
      m_currentFile.getName();
  }

  protected void newDocument() {
    m_doc = (HTMLDocument)m_kit.createDefaultDocument();
    m_context = m_doc.getStyleSheet();

    m_editor.setDocument(m_doc);
    m_currentFile = null;
    setTitle(APP_NAME+" ["+getDocumentName()+"]");

    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
        m_doc.addDocumentListener(new UpdateListener());
        m_textChanged = false;
      }
    });
  }

  protected void openDocument() {
    if (m_chooser.showOpenDialog(HtmlProcessor.this) !=
      JFileChooser.APPROVE_OPTION)
```

3 Returns name of current file

4 Creates new HTML document

5 Uses JFileChooser and FileInputStream to read in an HTML file

```java
      return;
    File f = m_chooser.getSelectedFile();
    if (f == null || !f.isFile())
      return;
    m_currentFile = f;
    setTitle(APP_NAME+" ["+getDocumentName()+"]");

    HtmlProcessor.this.setCursor(
      Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

    try {
      InputStream in = new FileInputStream(m_currentFile);
      m_doc = (HTMLDocument)m_kit.createDefaultDocument();
      m_kit.read(in, m_doc, 0);
      m_context = m_doc.getStyleSheet();
      m_editor.setDocument(m_doc);
      in.close();
    }
    catch (Exception ex) {
      showError(ex, "Error reading file "+m_currentFile);
    }
    HtmlProcessor.this.setCursor(Cursor.getPredefinedCursor(
      Cursor.DEFAULT_CURSOR));

    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        m_editor.setCaretPosition(1);
        m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
        m_doc.addDocumentListener(new UpdateListener());
        m_textChanged = false;
      }
    });
  }

  protected boolean saveFile(boolean saveAs) {
    if (!saveAs && !m_textChanged)
      return true;
    if (saveAs || m_currentFile == null) {
      if (m_chooser.showSaveDialog(HtmlProcessor.this) !=
        JFileChooser.APPROVE_OPTION)
        return false;
      File f = m_chooser.getSelectedFile();
      if (f == null)
        return false;
      m_currentFile = f;
      setTitle(APP_NAME+" ["+getDocumentName()+"]");
    }

    HtmlProcessor.this.setCursor(
      Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    try {
      OutputStream out = new FileOutputStream(m_currentFile);
      m_kit.write(out, m_doc, 0, m_doc.getLength());
      out.close();
```

6 **Uses JFileChooser and FileOutputStream to save current document to file**

```
        m_textChanged = false;
      }
      catch (Exception ex) {
        showError(ex, "Error saving file "+m_currentFile);
      }
      HtmlProcessor.this.setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      return true;
    }

    protected boolean promptToSave() {
      if (!m_textChanged)
        return true;
      int result = JOptionPane.showConfirmDialog(this,
        "Save changes to "+getDocumentName()+"?",
        APP_NAME, JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.INFORMATION_MESSAGE);
      switch (result) {
      case JOptionPane.YES_OPTION:
        if (!saveFile(false))
          return false;
        return true;
      case JOptionPane.NO_OPTION:
        return true;
      case JOptionPane.CANCEL_OPTION:
        return false;
      }
      return true;
    }

    public void showError(Exception ex, String message) {
      ex.printStackTrace();
      JOptionPane.showMessageDialog(this,
        message, APP_NAME,
        JOptionPane.WARNING_MESSAGE);
    }

    public static void main(String argv[]) {
      JFrame.setDefaultLookAndFeelDecorated(true);
      JDialog.setDefaultLookAndFeelDecorated(true);

      HtmlProcessor frame = new HtmlProcessor();
      frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
      frame.setVisible(true);
    }

    class UpdateListener implements DocumentListener {

      public void insertUpdate(DocumentEvent e) {
        m_textChanged = true;
      }

      public void removeUpdate(DocumentEvent e) {
        m_textChanged = true;
      }
```

7 **Prompts user to save**

8 **Displays error messages in dialogs**

9 **Used to change the flag which indicates whether a document has been modified**

```
      public void changedUpdate(DocumentEvent e) {
        m_textChanged = true;
      }
    }
}

// Class SmallButton unchanged from section 12.4

// Class SimpleFilter unchanged from section 14.1.9
```

20.1.1 Understanding the code

Class HtmlProcessor

This class extends `JFrame` to provide the supporting frame for this example. Several instance variables are declared:

- `JTextPane m_editor`: main text component.
- `StyleContext m_context`: a group of styles and their associated resources for the documents in this example.
- `HTMLDocument m_doc`: current document model.
- `HTMLEditorKit m_kit`: editor kit that knows how to read/write HTML documents.
- `SimpleFilter m_HTMLFilter`: file filter for ".HTML" files.
- `JToolBar m_toolBar`: toolbar containing New, Open, Save buttons.
- `JFileChooser m_chooser`: file chooser used to load and save HTML files.
- `File m_currentFile`: currently opened HTML file (if any).
- `boolean m_textChanged`: keeps track of whether any changes have been made since the document was last saved.

The `HtmlProcessor` constructor first instantiates our `JTextPane` and `HTMLEditorKit`, and assigns the editor kit to the text pane (it is important that this is done before any documents are created). The editor component is then placed in a `JScrollPane` which is placed in the center of the frame. The `JFileChooser` component is created and an instance of our `Simple-Filter` class (developed in chapter 14) is used as a filter to only allow the choice of HTML documents. A `WindowListener` is added to call our custom `promptToSave()` method to ensure that the user has the opportunity to save any changes before closing the application. This `WindowListener` also ensures that our editor component automatically receives the focus when this application regains the focus.

❶ (see above)

❷ The `createMenuBar()` method creates a menu bar with a single menu titled "File" and a toolbar with three buttons. Actions for New, Open, Save, Save As, and Exit are created and added to the File menu. The New, Open, and Save actions are also added to the toolbar. This code is very similar to the code used in the examples of chapter 12. The important difference is that we use `InputStreams` and `OutputStreams` rather than `Readers` and `Writers`. The reason for this is that HTML uses 1-byte encoding which is incompatible with the 2-byte encoding used by readers and writers.

❸ The `getDocumentName()` method simply returns the name of the file corresponding to the current document, or untitled if it hasn't been saved to disk.

❹ The `newDocument()` method is responsible for creating a new `HTMLDocument` instance using `HTMLEditorKit`'s `createDefaultDocument()` method. Once created our `StyleContext`

variable, `m_context`, is assiged to this new document's stylesheet with `HTMLDocument`'s `getStyleSheet()` method. The title of the frame is then updated and a `Runnable` instance is created and sent to the `SwingUtilities.invokeLater()` method to scroll the document to the beginning when it is finished loading. Finally, an instance of our custom `UpdateListener` class is added as a `DocumentListener`, and the `m_textChanged` variable is set to `false` to indicate that no changes to the document have been made yet.

5 The `openDocument()` is similar to the `newDocument()` method but uses the `JFileChooser` to allow selection of an existing HTML file to load, and uses an `InputStream` object to read the contents of that file.

6 The `saveFile()` method takes a `boolean` parameter specifying whether the method should act as a Save As process or just a regular Save. If `true`, indicating a Save As process, the `JFileChooser` is displayed to allow the user to specify the file and location to save the document to. An `OutputStream` is used to write the contents of the document to the destination `File`.

7 The `promptToSave()` method checks the `m_textChanged` flag and, if `true`, displays a `JOptionPane`asking whether or not the current document should be saved. This method is called before a new document is created, a document is opened, or the application is closed to ensure that the user has a chance to save any changes to the current document before losing them.

8 The `showError()` method is used to display error messages in a `JOptionPane`. It is often useful to display exceptions to users so that they know an error happened and so that they may eventually report errors back to you if they are in fact bugs.

Class UpdateListener

9 This `DocumentListener` subclass is used to modify the state of our `m_textChanged` variable. Whenever an insertion, removal, or document change is made this variable is set to `true`. This allows `HtmlProcessor`'s `promptToSave()` method to ensure the user has the option of saving any changes before loading a new document or exiting the application.

20.1.2 Running the code

Figure 20.1 shows our HTML editor in action. Use menu or toolbar buttons to open an HTML file. Save the HTML file and open it in another HTML-aware application (such as Netscape) to verify compatibility. Try modifying a document and exiting the application before saving it. Note the dialog that is displayed asking whether or not you'd like to save the changes you've made before exiting.

20.2 HTML EDITOR, PART II: MANAGING FONTS

The following example adds the ability to:

- Select any font available on the system
- Change font size
- Select bold and italic characteristics

This functionality is similar to the font functionality used in the examples of chapter 12. The important difference here is that the selected font applies not to the whole text component (the only possible thing with plain text documents), but to the selected region of our HTML–styled document text.

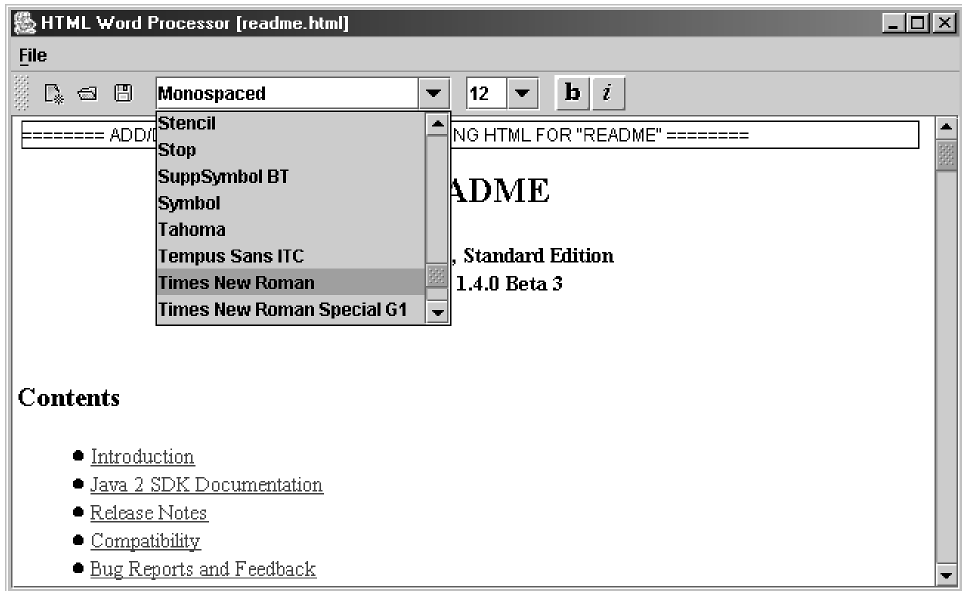

Figure 20.2 JTextPane word processor allowing font attribute assignments to selected text

Example 20.2

HtmlProcessor.java

see \Chapter20\2

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

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

public class HtmlProcessor extends JFrame {

// Unchanged code from example 20.1

  protected JComboBox m_cbFonts;
  protected JComboBox m_cbSizes;
  protected SmallToggleButton m_bBold;
  protected SmallToggleButton m_bItalic;

  protected String m_fontName = "";
  protected int m_fontSize = 0;
```

```
  protected boolean m_skipUpdate;

  protected int m_xStart = -1;
  protected int m_xFinish = -1;

  public HtmlProcessor() {
    super(APP_NAME);
    setSize(650, 400);

// Unchanged code from example 20.1

    CaretListener lst = new CaretListener() {
      public void caretUpdate(CaretEvent e) {
        showAttributes(e.getDot());
      }
    };
    m_editor.addCaretListener(lst);

    FocusListener flst = new FocusListener() {
      public void focusGained(FocusEvent e) {
        int len = m_editor.getDocument().getLength();
        if (m_xStart>=0 && m_xFinish>=0 &&
          m_xStart<len && m_xFinish<len)
          if (m_editor.getCaretPosition()==m_xStart) {
            m_editor.setCaretPosition(m_xFinish);
            m_editor.moveCaretPosition(m_xStart);
          }
          else
            m_editor.select(m_xStart, m_xFinish);
      }

      public void focusLost(FocusEvent e) {
        m_xStart = m_editor.getSelectionStart();
        m_xFinish = m_editor.getSelectionEnd();
      }
    };
    m_editor.addFocusListener(flst);

    newDocument();

    // Unchanged code from example 20.1
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    // Unchanged code from example 20.1

    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    String[] fontNames = ge.getAvailableFontFamilyNames();

    m_toolBar.addSeparator();
    m_cbFonts = new JComboBox(fontNames);
    m_cbFonts.setMaximumSize(new Dimension(200, 23));
    m_cbFonts.setEditable(true);
```

1 Caret listener used to update toolbar state when caret moves

2 Focus listener to save and restore the caret position when selection occurs in another text component

3 Get complete list of available font names

4 New font choice combo box

```
ActionListener lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_fontName = m_cbFonts.getSelectedItem().toString();
    MutableAttributeSet attr = new SimpleAttributeSet();
    StyleConstants.setFontFamily(attr, m_fontName);
    setAttributeSet(attr);
    m_editor.grabFocus();
  }
};
m_cbFonts.addActionListener(lst);
m_toolBar.add(m_cbFonts);

m_toolBar.addSeparator();
m_cbSizes = new JComboBox(new String[] {"8", "9", "10",
  "11", "12", "14", "16", "18", "20", "22", "24", "26",
  "28", "36", "48", "72"});
m_cbSizes.setMaximumSize(new Dimension(50, 23));
m_cbSizes.setEditable(true);

lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    int fontSize = 0;
    try {
      fontSize = Integer.parseInt(m_cbSizes.
        getSelectedItem().toString());
    }
    catch (NumberFormatException ex) { return; }

    m_fontSize = fontSize;
    MutableAttributeSet attr = new SimpleAttributeSet();
    StyleConstants.setFontSize(attr, fontSize);
    setAttributeSet(attr);
    m_editor.grabFocus();
  }
};
m_cbSizes.addActionListener(lst);
m_toolBar.add(m_cbSizes);

m_toolBar.addSeparator();
ImageIcon img1 = new ImageIcon("Bold16.gif");
m_bBold = new SmallToggleButton(false, img1, img1,
  "Bold font");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    MutableAttributeSet attr = new SimpleAttributeSet();
    StyleConstants.setBold(attr, m_bBold.isSelected());
    setAttributeSet(attr);
    m_editor.grabFocus();
  }
};
m_bBold.addActionListener(lst);
m_toolBar.add(m_bBold);
```

5 Applies new font to selected text

6 New font sizes combo box

6 Applies new font size to selected text

7 Toggle button to manage bold property

```
    img1 = new ImageIcon("Italic16.gif");
    m_bItalic = new SmallToggleButton(false, img1, img1,
      "Italic font");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setItalic(attr, m_bItalic.isSelected());
        setAttributeSet(attr);
        m_editor.grabFocus();
      }
    };
    m_bItalic.addActionListener(lst);
    m_toolBar.add(m_bItalic);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  // Unchanged code from example 20.1

  protected void newDocument() {
   // Unchanged code from example 20.1

    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        showAttributes(0);
        m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
        m_doc.addDocumentListener(new UpdateListener());
        m_textChanged = false;
      }
    });
  }

  protected void openDocument() {
   // Unchanged code from example 20.1

    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        m_editor.setCaretPosition(1);
        showAttributes(1);
        m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
        m_doc.addDocumentListener(new UpdateListener());
        m_textChanged = false;
      }
    });
  }

  // Unchanged code from example 20.1

  protected void showAttributes(int p) {
    m_skipUpdate = true;
    AttributeSet attr = m_doc.getCharacterElement(p).
      getAttributes();
    String name = StyleConstants.getFontFamily(attr);
    if (!m_fontName.equals(name)) {
```

5 Toggle button to manage italic property

8 Sets state of toolbar buttons based on position of caret

```
        m_fontName = name;
        m_cbFonts.setSelectedItem(name);
      }
      int size = StyleConstants.getFontSize(attr);
      if (m_fontSize != size) {
        m_fontSize = size;
        m_cbSizes.setSelectedItem(Integer.toString(m_fontSize));
      }
      boolean bold = StyleConstants.isBold(attr);
      if (bold != m_bBold.isSelected())
        m_bBold.setSelected(bold);
      boolean italic = StyleConstants.isItalic(attr);
      if (italic != m_bItalic.isSelected())
        m_bItalic.setSelected(italic);
      m_skipUpdate = false;
    }

    protected void setAttributeSet(AttributeSet attr) {
      if (m_skipUpdate)
        return;
      int xStart = m_editor.getSelectionStart();
      int xFinish = m_editor.getSelectionEnd();
      if (!m_editor.hasFocus()) {
        xStart = m_xStart;
        xFinish = m_xFinish;
      }
      if (xStart != xFinish) {
        m_doc.setCharacterAttributes(xStart, xFinish - xStart,
          attr, false);
      }
      else {
        MutableAttributeSet inputAttributes =
          m_kit.getInputAttributes();
        inputAttributes.addAttributes(attr);
      }
    }

    public static void main(String argv[]) {
      HtmlProcessor frame = new HtmlProcessor();
      frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
      frame.setVisible(true);
    }

    // Unchanged code from example 20.1
  }

  // Unchanged code from example 20.1

  // Class SmallToggleButton unchanged from section 12.4
```

9 Used to assign a given set of attributes to currently selected text

20.2.1 Understanding the code

Class HtmlProcessor

Several new instance variables have been added:

- `JComboBox m_cbFonts`: toolbar component to select the font name.
- `JComboBox m_cbSizes`: toolbar component to select the font size.
- `SmallToggleButton m_bBold`: toolbar component to select the bold font style.
- `SmallToggleButton m_bItalic`: toolbar component to select the italic font style.
- `String m_fontName`: current font name.
- `int m_fontSize`: current font size.
- `boolean m_skipUpdate`: flag used to skip word processor update (see below).
- `int m_xStart`: used to store the selection start position.
- `int m_xFinish`: used to store the selection end position.

❶ The `HtmlProcessor` constructor adds a `CaretListener` to our `m_editor` text pane. The `caretUpdate()` method of this listener is invoked whenever the caret position is changed. `caretUpdate()` calls our `showAttributes()` method to update the toolbar component's states to display the currently selected font attributes.

❷ A `FocusListener` is also added to our `m_editor` component. The two methods of this listener, `focusGained()` and `focusLost()`, will be invoked when the editor gains and loses the focus respectively. The purpose of this implementation is to save and restore the starting and end positions of the text selection. The reason we do this is because Swing supports only one text selection at any given time. This means that if the user selects some text in the editor component to modify its attributes, and then goes off and makes a text selection in some other component, the original text selection will disappear. This can potentially be very annoying to the user. To fix this problem we save the selection before the editor component loses the focus. When the focus is gained we restore the previously saved selection. We distinguish between two possible situations: when the caret is located at the beginning of the selection and when it is located at the end of the selection. In the first case we position the caret at the end of the stored interval with the `setCaretPosition()` method, and then move the caret backward to the beginning of the stored interval with the `moveCaretPosition()` method. The second situation is easily handled using the `select()` method.

❹ The `createMenuBar()` method creates new components to manage font properties for the selected text interval. First, the `m_cbFonts` combo box is used to select the font family name. Unlike the example in chapter 12, which used several predefined font names, this example uses all fonts available to the user's system. A complete list of the available font names can be ❸ obtained through the `getAvailableFontFamilyNames()` method of `GraphicsEnvironment` (see section 2.8). Also note that the editable property of this combo box component is set to `true`, so the font name can be both selected from the drop-down list and entered in by hand.

❺ Once a new font name is selected, it is applied to the selected text through the use of an attached `ActionListener`. The selected font family name is assigned to a `SimpleAttributeSet` instance with the `StyleConstants.setFontFamily()` method. Then our custom `setAttributeSet()` method is called to modify the attributes of the selected text according to this `SimpleAttributeSet`.

6. The m_cbSizes combo box is used to select the font size. It is initiated with a set of predefined sizes. The editable property is set to true so the font size can be both selected from the drop-down list and entered by hand. Once a new font size is selected, it is applied to the selected text through the use of an attached ActionListener. The setup is similar to that used for the m_cbFonts component. The StyleConstants.setFontSize() method is used to set the font size. Our custom setAttributeSet() method is then used to apply this attribute set to the selected text.

7. The bold and italic properties are managed by two SmallToggleButtons (a custom button class we developed in chapter 12): m_bBold and m_bItalic respectively. These buttons receive ActionListeners which create a SimpleAttributeSet instance with the bold or italic property with StyleConstants.setBold() or StyleConstants.setItalic(). Then our custom setAttributeSet() method is called to apply this attribute set.

8. The showAttributes() method is called to set the state of the toolbar components described earlier according to the font properties of the text at the given caret position. This method sets the m_skipUpdate flag to true at the beginning and false at the end of its execution (the purpose of this will be explained soon). Then an AttributeSet instance corresponding to the character element at the current caret position in the editor's document is retrieved with the getAttributes() method. The StyleConstants.getFontFamily() method is used to retrieve the current font name from this attribute set. If it is not equal to the previously selected font name (stored in the m_fontName instance variable) it is selected in the m_cbFonts combo box. The other toolbar controls are handled in a similar way.

9. The setAttributeSet() method is used to assign a given set of attributes to the currently selected text. Note that this method does nothing (simply returns) if the m_skipUpdate flag is set to true. This is done to prevent the backward link with the showAttributes() method. As soon as we assign some value to a combo box in the showAttributes() method (e.g., font size) this internally triggers a call to the setAttributeSet() method (because Action-Listeners attached to combo boxes are invoked even when selection changes occur programmatically). The purpose of showAttributes() is to simply make sure that the attributes corresponding to the character element at the current text position are accurately reflected in the toolbar components. To prevent the combo box ActionListeners from invoking unnecessary operations we prohibit any text property updates from occuring in setAttributeSet() while the showAttributes() method is being executed (this is the whole purpose of the m_skipUpdate flag).

setAttributeSet() first determines the start and end positions of the selected text. If m_editor currently does not have the focus, the stored bounds, m_xStart and m_xFinish, are used instead. If the selection is not empty (xStart != xFinish), the setCharacter-Attributes() method is called to assign the given set of attributes to the selection. Note that this new attribute set does not have to contain a complete set of attributes. It simply replaces only the existing attributes for which it has new values, leaving the remainder unchanged. If the selection is empty, the new attributes are added to the input attributes of the editor kit (recall that StyledEditorKit's input attributes are those attributes that will be applied to newly inserted text–HTMLEditorKit extends StyledEditorKit).

20.2.2 Running the code

Figure 20.2 shows our editor with a font combo box selection in process. Open an existing HTML file and move the cursor to various positions in the text. Note that the text attributes displayed in the toolbar components are updated correctly. Select a portion of text and use the toolbar components to modify the selection's font attributes. Type a new font name and font size in the editable combo box and press Enter. This has the same effect as selecting a choice from the drop-down list. Save the HTML file and open it in another HTML-aware application to verify that your changes were saved correctly.

20.3 HTML EDITOR, PART III: DOCUMENT PROPERTIES

In this example we add the following features to our HTML editor application:

- Image insertion
- Hyperlink insertion
- Foreground color selection
- An HTML page properties dialog to assign text color, link colors, background color and title
- An HTML source dialog that allows editing

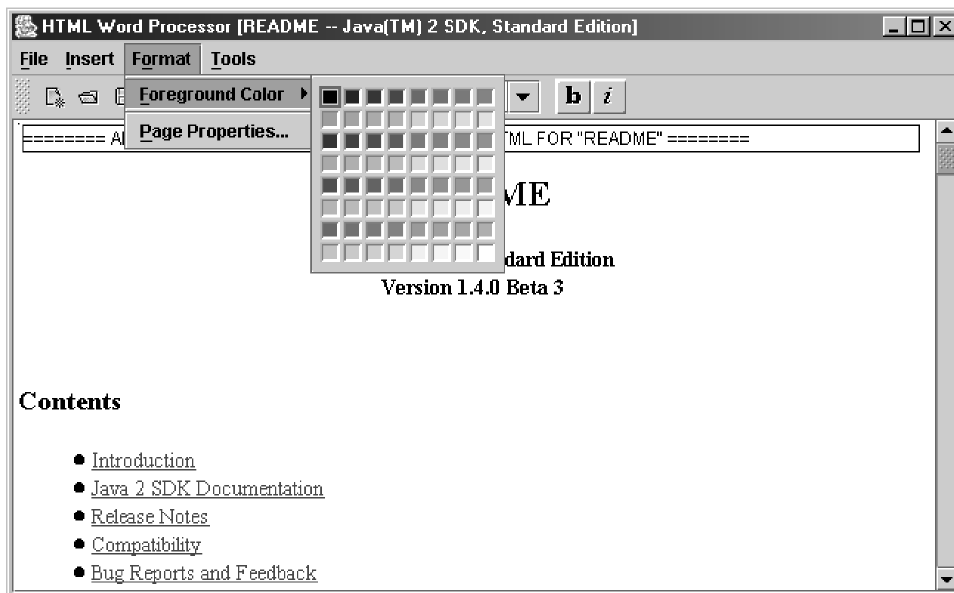

Figure 20.3 `HtmlProcessor` **showing foreground color selection component**

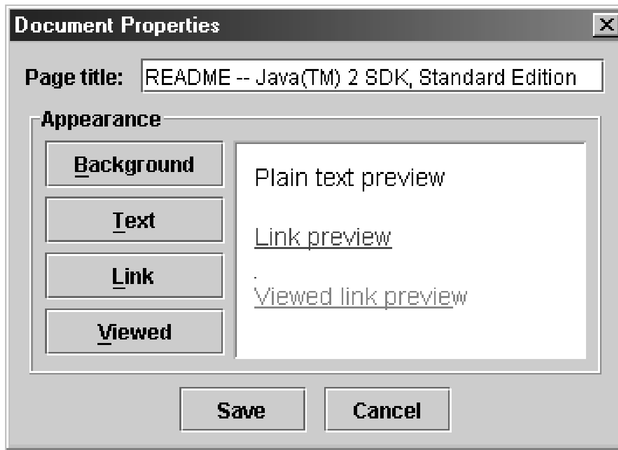

Figure 20.4 **`HtmlProcessor`'s document properties dialog**

Example 20.3

HtmlProcessor.java

see \Chapter20\3

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

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

import dl.*;

public class HtmlProcessor extends JFrame {

  public static final String APP_NAME = "HTML HTML Editor";

  protected JTextPane m_editor;
  protected StyleSheet m_context;
  protected MutableHTMLDocument m_doc;
  protected CustomHTMLEditorKit m_kit;
  protected SimpleFilter m_htmlFilter;
  protected JToolBar m_toolBar;

// Unchanged code from example 20.2

  protected int m_xStart = -1;
  protected int m_xFinish = -1;

  protected ColorMenu m_foreground;

// Unchanged code from example 20.2
```

❶ **Custom document and editor kit classes are now used**

❷ **Menu component used to select foreground color**

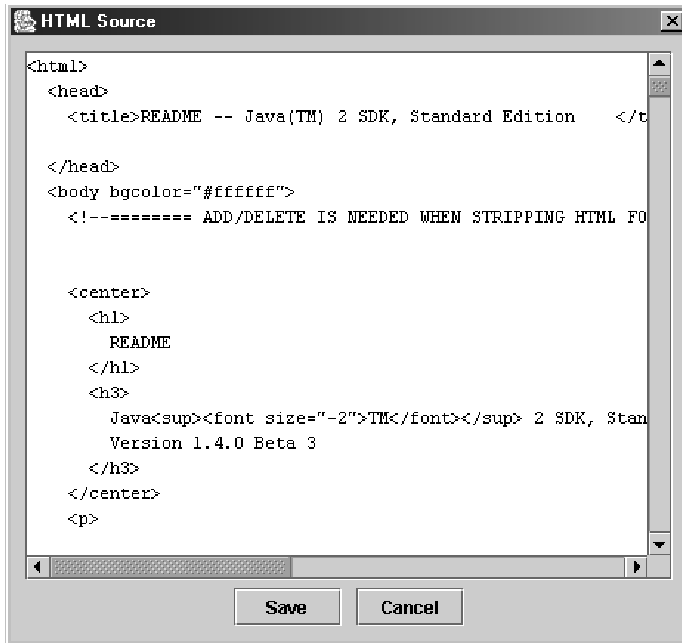

Figure 20.5
`HtmlProcessor`'s
HTML source dialog

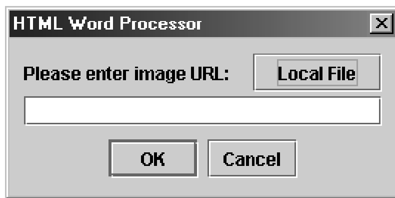

Figure 20.6
`HtmlProcessor`'s
insert image dialog

```
protected JMenuBar createMenuBar() {
   JMenuBar menuBar = new JMenuBar();

   JMenu mFile = new JMenu("File");
   mFile.setMnemonic('f');
  // Unchanged code from example 20.2

   JMenu mInsert = new JMenu("Insert");
   mInsert.setMnemonic('i');

   item = new JMenuItem("Image...");
   item.setMnemonic('i');
   lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         String url = inputURL("Please enter image URL:", null);}
         if (url == null)
            return;
         try {
            ImageIcon icon = new ImageIcon(new URL(url));
```

3 **New menu item allowing insertion of an image**

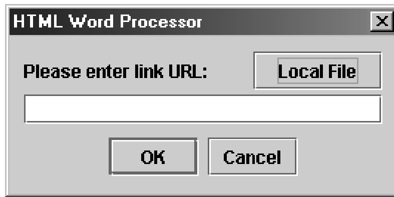

Figure 20.7
`HtmlProcessor`**'s insert link dialog**

```
            int w = icon.getIconWidth();
            int h = icon.getIconHeight();
            if (w<=0 || h<=0) {
              JOptionPane.showMessageDialog(HtmlProcessor.this,
                "Error reading image URL\n"+
                url, APP_NAME,
                JOptionPane.WARNING_MESSAGE);
                return;

            MutableAttributeSet attr = new SimpleAttributeSet();
            attr.addAttribute(StyleConstants.NameAttribute,
           HTML.Tag.IMG);
            attr.addAttribute(HTML.Attribute.SRC, url);
            attr.addAttribute(HTML.Attribute.HEIGHT,
           Integer.toString(h));
            attr.addAttribute(HTML.Attribute.WIDTH,
           Integer.toString(w));
            int p = m_editor.getCaretPosition();
            m_doc.insertString(p, " ", attr);
          }
          catch (Exception ex) {
            showError(ex, "Error: "+ex);
          }
        }
      };
      item.addActionListener(lst);
      mInsert.add(item);

      item = new JMenuItem("Hyperlink...");
      item.setMnemonic('h');
      lst = new ActionListener(){
        public void actionPerformed(ActionEvent e) {
          String oldHref = null;

          int p = m_editor.getCaretPosition();
          AttributeSet attr = m_doc.getCharacterElement(p).
            getAttributes();
          AttributeSet anchor =
            (AttributeSet)attr.getAttribute(HTML.Tag.A);
          if (anchor != null)
            oldHref = (String)anchor.getAttribute(HTML.Attribute.HREF);

          String newHref = inputURL("Please enter link URL:", oldHref);
          if (newHref == null)
```

3 New menu item allowing insertion of a hyperlink

```
            return;
         SimpleAttributeSet attr2 = new SimpleAttributeSet();
         attr2.addAttribute(StyleConstants.NameAttribute, HTML.Tag.A);
         attr2.addAttribute(HTML.Attribute.HREF, newHref);
         setAttributeSet(attr2, true);
         m_editor.grabFocus();
      }
   };
   item.addActionListener(lst);
   mInsert.add(item);

   menuBar.add(mInsert);

   JMenu mFormat = new JMenu("Format");
   mFormat.setMnemonic('o');

   m_foreground = new ColorMenu("Foreground Color");
   m_foreground.setColor(m_editor.getForeground());
   m_foreground.setMnemonic('f');
   lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         MutableAttributeSet attr = new SimpleAttributeSet();
         StyleConstants.setForeground(attr,
         m_foreground.getColor());
         setAttributeSet(attr);
      }
   };
   m_foreground.addActionListener(lst);
   mFormat.add(m_foreground);

   MenuListener ml = new MenuListener() {
      public void menuSelected(MenuEvent e) {
         int p = m_editor.getCaretPosition();
         AttributeSet attr = m_doc.getCharacterElement(p).
            getAttributes();
         Color c = StyleConstants.getForeground(attr);
         m_foreground.setColor(c);
      }
      public void menuDeselected(MenuEvent e) {}
      public void menuCanceled(MenuEvent e) {}
   };
   m_foreground.addMenuListener(ml);
   mFormat.addSeparator();

   item = new JMenuItem("Page Properties...");
   item.setMnemonic('p');
   lst = new ActionListener(){
      public void actionPerformed(ActionEvent e) {
         DocumentPropsDlg dlg = new
         DocumentPropsDlg(HtmlProcessor.this, m_doc);
         dlg.show();
         if (dlg.succeeded())
            documentChanged();
```

4 Menu item allowing selection of foreground color

4 Menu item to display page properties dialog

```java
      }
    };
    item.addActionListener(lst);
    mFormat.add(item);

    menuBar.add(mFormat);

    JMenu mTools = new JMenu("Tools");
    mTools.setMnemonic('t');

    item = new JMenuItem("HTML Source...");
    item.setMnemonic('s');
    lst = new ActionListener(){
      public void actionPerformed(ActionEvent e) {
        try {
          StringWriter sw = new StringWriter();
          m_kit.write(sw, m_doc, 0, m_doc.getLength());
          sw.close();

          HtmlSourceDlg dlg = new HtmlSourceDlg(
            HtmlProcessor.this, sw.toString());
          dlg.show();
          if (!dlg.succeeded())
            return;

          StringReader sr = new StringReader(dlg.getSource());
          m_doc = (MutableHTMLDocument)m_kit.createDocument();
          m_context = m_doc.getStyleSheet();
          m_kit.read(sr, m_doc, 0);
          sr.close();
          m_editor.setDocument(m_doc);
          documentChanged();
        }
        catch (Exception ex) {
          showError(ex, "Error: "+ex);
        }
      }
    };
    item.addActionListener(lst);
    mTools.add(item);

    menuBar.add(mTools);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  protected String getDocumentName() {
    String title = m_doc.getTitle();
    if (title != null && title.length() > 0)
      return title;
    return m_currentFile==null ? "Untitled" :
      m_currentFile.getName();
  }
```

5 Menu item to display HTML source dialog

```
protected void newDocument() {
  m_doc = (MutableHTMLDocument)m_kit.createDocument();
  m_context = m_doc.getStyleSheet();

  m_editor.setDocument(m_doc);
  m_currentFile = null;
  setTitle(APP_NAME+" ["+getDocumentName()+"]");

  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      showAttributes(0);
      m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
      m_doc.addDocumentListener(new UpdateListener());
      m_textChanged = false;
    }
  });
}

protected void openDocument() {
 // Unchanged code from example 20.2

  HtmlProcessor.this.setCursor(
    Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  try {
    InputStream in = new FileInputStream(m_currentFile);
    m_doc = (MutableHTMLDocument)m_kit.createDocument();
    m_kit.read(in, m_doc, 0);
    m_context = m_doc.getStyleSheet();
    m_editor.setDocument(m_doc);
    in.close();
  }
  catch (Exception ex) {
    showError(ex, "Error reading file "+m_currentFile);
  }
  HtmlProcessor.this.setCursor(Cursor.getPredefinedCursor(
    Cursor.DEFAULT_CURSOR));

  // Unchanged code from example 20.2
}

// Unchanged code from example 20.2

protected void setAttributeSet(AttributeSet attr) {
  setAttributeSet(attr, false);
}

protected void setAttributeSet(AttributeSet attr,
  boolean setParagraphAttributes) {
  if (m_skipUpdate)
    return;
  int xStart = m_editor.getSelectionStart();
  int xFinish = m_editor.getSelectionEnd();
  if (!m_editor.hasFocus()) {
    xStart = m_xStart;
    xFinish = m_xFinish;
  }
```

6 **Updated to allow specification paragraph or character attributes**

```java
      if (setParagraphAttributes)
        m_doc.setParagraphAttributes(xStart,
          xFinish - xStart, attr, false);
      else if (xStart != xFinish)
        m_doc.setCharacterAttributes(xStart,
          xFinish - xStart, attr, false);
      else {
        MutableAttributeSet inputAttributes =
          m_kit.getInputAttributes();
        inputAttributes.addAttributes(attr);
      }
    }

    protected String inputURL(String prompt, String initialValue) {
      JPanel p = new JPanel();
      p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
      p.add(new JLabel(prompt));
      p.add(Box.createHorizontalGlue());
      JButton bt = new JButton("Local File");
      bt.setRequestFocusEnabled(false);
      p.add(bt);

      final JOptionPane op = new JOptionPane(p,
        JOptionPane.PLAIN_MESSAGE, JOptionPane.OK_CANCEL_OPTION);
      op.setWantsInput(true);
      if (initialValue != null)
        op.setInitialSelectionValue(initialValue);

      ActionListener lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          JFileChooser chooser = new JFileChooser();
          if (chooser.showOpenDialog(HtmlProcessor.this) !=
            JFileChooser.APPROVE_OPTION)
            return;
          File f = chooser.getSelectedFile();
          try {
            String str = f.toURL().toString();
            op.setInitialSelectionValue(str);
          }
          catch (Exception ex) {
            ex.printStackTrace();
          }
        }
      };
      bt.addActionListener(lst);

      JDialog dlg = op.createDialog(this, APP_NAME);
      dlg.show();
      dlg.dispose();

  Object value = op.getInputValue();
      if (value == JOptionPane.UNINITIALIZED_VALUE)
        return null;
      String str = (String)value;
```

7 Method to insert a URL (either an image or hyperlink in this example)

```
    if (str != null && str.length() == 0)
      str = null;
    return str;
  }

  public void documentChanged() {
    m_editor.setDocument(new HTMLDocument());
    m_editor.setDocument(m_doc);
    m_editor.revalidate();
    m_editor.repaint();
    setTitle(APP_NAME+" ["+getDocumentName()+"]");
    m_textChanged = true;
  }

  // Unchanged code from example 20.2
}

// Unchanged code from example 20.2

// Class ColorMenu unchanged from chapter 12

class Utils
{
  // Copied from javax.swing.text.html.CSS class
  // because it is not publicly accessible there.
  public static String colorToHex(Color color) {
    String colorstr = new String("#");

    // Red
    String str = Integer.toHexString(color.getRed());
    if (str.length() > 2)
      str = str.substring(0, 2);
    else if (str.length() < 2)
      colorstr += "0" + str;
    else
      colorstr += str;

    // Green
    str = Integer.toHexString(color.getGreen());
    if (str.length() > 2)
      str = str.substring(0, 2);
    else if (str.length() < 2)
      colorstr += "0" + str;
    else
      colorstr += str;

    // Blue
    str = Integer.toHexString(color.getBlue());
    if (str.length() > 2)
      str = str.substring(0, 2);
    else if (str.length() < 2)
      colorstr += "0" + str;
    else
      colorstr += str;
    return colorstr;
  }
```

8 Brute force method at updating document display

9 Returns the hot value for a given color

```
}

class CustomHTMLEditorKit extends HTMLEditorKit {
  public Document createDocument() {
    StyleSheet styles = getStyleSheet();
    StyleSheet ss = new StyleSheet();

    ss.addStyleSheet(styles);

    MutableHTMLDocument doc = new MutableHTMLDocument(ss);
    doc.setParser(getParser());
    doc.setAsynchronousLoadPriority(4);
    doc.setTokenThreshold(100);
    return doc;
  }
}

class MutableHTMLDocument extends HTMLDocument {
  public MutableHTMLDocument(StyleSheet styles) {
    super(styles);
  }

  public Element getElementByTag(HTML.Tag tag) {
    Element root = getDefaultRootElement();
    return getElementByTag(root, tag);
  }

  public Element getElementByTag(Element parent, HTML.Tag tag) {
    if (parent == null || tag == null)
      return null;
    for (int k=0; k<parent.getElementCount(); k++) {
      Element child = parent.getElement(k);
      if (child.getAttributes().getAttribute(
          StyleConstants.NameAttribute).equals(tag))
        return child;
      Element e = getElementByTag(child, tag);
      if (e != null)
        return e;
    }
    return null;
  }

  public String getTitle() {
    return (String)getProperty(Document.TitleProperty);
  }

  // This will work only if the <title> element was
  // previously created. Looks like a bug in the HTML package.
  public void setTitle(String title) {
    Dictionary di = getDocumentProperties();
    di.put(Document.TitleProperty, title);
    setDocumentProperties(di);
  }

  public void addAttributes(Element e, AttributeSet attributes) {
    if (e == null || attributes == null)
```

10 Custom editor kit to return MutableHTMLDocuments

11 Custom HTMLDocument with enhancement to locate text elements corresponding to a given HTML tag

```
      return;
    try {
      writeLock();
      MutableAttributeSet mattr =
        (MutableAttributeSet)e.getAttributes();
      mattr.addAttributes(attributes);
      fireChangedUpdate(new DefaultDocumentEvent(0, getLength(),
        DocumentEvent.EventType.CHANGE));
    }
    finally {
      writeUnlock();
    }
  }
}

class DocumentPropsDlg extends JDialog {
  protected boolean m_succeeded = false;
  protected MutableHTMLDocument m_doc;

  protected Color m_backgroundColor;
  protected Color m_textColor;
  protected Color m_linkColor;
  protected Color m_viewedColor;

  protected JTextField m_titleTxt;
  protected JTextPane m_previewPane;

  public DocumentPropsDlg(JFrame parent, MutableHTMLDocument doc) {
    super(parent, "Page Properties", true);
    m_doc = doc;

    Element body = m_doc.getElementByTag(HTML.Tag.BODY);
    if (body != null) {
      AttributeSet attr = body.getAttributes();
      StyleSheet syleSheet = m_doc.getStyleSheet();
      Object obj = attr.getAttribute(HTML.Attribute.BGCOLOR);
      if (obj != null)
        m_backgroundColor = syleSheet.stringToColor((String)obj);
      obj = attr.getAttribute(HTML.Attribute.TEXT);
      if (obj != null)
        m_textColor = syleSheet.stringToColor((String)obj);
      obj = attr.getAttribute(HTML.Attribute.LINK);
      if (obj != null)
        m_linkColor = syleSheet.stringToColor((String)obj);
      obj = attr.getAttribute(HTML.Attribute.VLINK);
      if (obj != null)
        m_viewedColor = syleSheet.stringToColor((String)obj);
    }

    ActionListener lst;
    JButton bt;

    JPanel pp = new JPanel(new DialogLayout2());
    pp.setBorder(new EmptyBorder(10, 10, 5, 10));

    pp.add(new JLabel("Page title:"));
```

12 **Custom dialog class to modify HTML document properties such as title, background color, text color, hyperlink color, and viewed hyperlink color**

```
m_titleTxt = new JTextField(m_doc.getTitle(), 24);
pp.add(m_titleTxt);

JPanel pa = new JPanel(new BorderLayout(5, 5));
Border ba = new TitledBorder(new EtchedBorder(
  EtchedBorder.RAISED), "Appearance");
pa.setBorder(new CompoundBorder(ba, new EmptyBorder(0, 5, 5, 5)));

JPanel pb = new JPanel(new GridLayout(4, 1, 5, 5));
bt = new JButton("Background");
bt.setMnemonic('b');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_backgroundColor =
    JColorChooser.showDialog(DocumentPropsDlg.this,
      "Document Background", m_backgroundColor);
    showColors();
  }
};
bt.addActionListener(lst);
pb.add(bt);

bt = new JButton("Text");
bt.setMnemonic('t');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_textColor = JColorChooser.showDialog(DocumentPropsDlg.this,
      "Text Color", m_textColor);
    showColors();
  }
};
bt.addActionListener(lst);
pb.add(bt);

bt = new JButton("Link");
bt.setMnemonic('l');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_linkColor = JColorChooser.showDialog(DocumentPropsDlg.this,
      "Links Color", m_linkColor);
    showColors();
  }
};
bt.addActionListener(lst);
pb.add(bt);

bt = new JButton("Viewed");
bt.setMnemonic('v');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_viewedColor = JColorChooser.showDialog(DocumentPropsDlg.this,
      "Viewed Links Color", m_viewedColor);
    showColors();
  }
```

```
    };
    bt.addActionListener(lst);
    pb.add(bt);
    pa.add(pb, BorderLayout.WEST);

    m_previewPane = new JTextPane();
    m_previewPane.setBackground(Color.white);
    m_previewPane.setEditable(false);
    m_previewPane.setBorder(new CompoundBorder(
      new BevelBorder(BevelBorder.LOWERED),
      new EmptyBorder(10, 10, 10, 10)));
    showColors();
    pa.add(m_previewPane, BorderLayout.CENTER);

    pp.add(pa);

    bt = new JButton("Save");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        saveData();
        dispose();
      }
    };
    bt.addActionListener(lst);
    pp.add(bt);

    bt = new JButton("Cancel");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dispose();
      }
    };
    bt.addActionListener(lst);
    pp.add(bt);

    getContentPane().add(pp, BorderLayout.CENTER);
    pack();
    setResizable(false);
    setLocationRelativeTo(parent);
  }

  public boolean succeeded() {
    return m_succeeded;
  }

  protected void saveData() {
    m_doc.setTitle(m_titleTxt.getText());

    Element body = m_doc.getElementByTag(HTML.Tag.BODY);
    MutableAttributeSet attr = new SimpleAttributeSet();
    if (m_backgroundColor != null)
      attr.addAttribute(HTML.Attribute.BGCOLOR,
        Utils.colorToHex(m_backgroundColor));
    if (m_textColor != null)
      attr.addAttribute(HTML.Attribute.TEXT,
        Utils.colorToHex(m_textColor));
```

```
      if (m_linkColor != null)
        attr.addAttribute(HTML.Attribute.LINK,
          Utils.colorToHex(m_linkColor));
      if (m_viewedColor != null)
        attr.addAttribute(HTML.Attribute.VLINK,
          Utils.colorToHex(m_viewedColor));
      m_doc.addAttributes(body, attr);

      m_succeeded = true;
    }

    protected void showColors() {
      DefaultStyledDocument doc = new DefaultStyledDocument();

      SimpleAttributeSet attr = new SimpleAttributeSet();
      StyleConstants.setFontFamily(attr, "Arial");
      StyleConstants.setFontSize(attr, 14);
      if (m_backgroundColor != null) {
        StyleConstants.setBackground(attr, m_backgroundColor);
        m_previewPane.setBackground(m_backgroundColor);
      }

      try {
        StyleConstants.setForeground(attr, m_textColor!=null ?
          m_textColor : Color.black);
        doc.insertString(doc.getLength(),
         "Plain text preview\n\n", attr);

        StyleConstants.setForeground(attr, m_linkColor!=null ?
          m_linkColor : Color.blue);
        StyleConstants.setUnderline(attr, true);
        doc.insertString(doc.getLength(), "Link preview\n\n", attr);

        StyleConstants.setForeground(attr, m_viewedColor!=null ?
          m_viewedColor : Color.magenta);
        StyleConstants.setUnderline(attr, true);
        doc.insertString(doc.getLength(), "Viewed link preview\n", attr);
      }
      catch (BadLocationException be) {
        be.printStackTrace();
      }
      m_previewPane.setDocument(doc);
    }
  }

  class HtmlSourceDlg extends JDialog {
    protected boolean m_succeeded = false;

    protected JTextArea m_sourceTxt;

    public HtmlSourceDlg(JFrame parent, String source) {
      super(parent, "HTML Source", true);

      JPanel pp = new JPanel(new BorderLayout());
      pp.setBorder(new EmptyBorder(10, 10, 5, 10));

      m_sourceTxt = new JTextArea(source, 20, 60);
```

13 Custom dialog to allow viewing and editing of HTML source

```
        m_sourceTxt.setFont(new Font("Courier", Font.PLAIN, 12));
        JScrollPane sp = new JScrollPane(m_sourceTxt);
        pp.add(sp, BorderLayout.CENTER);

        JPanel p = new JPanel(new FlowLayout());
        JPanel p1 = new JPanel(new GridLayout(1, 2, 10, 0));
        JButton bt = new JButton("Save");
        ActionListener lst = new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            m_succeeded = true;
            dispose();
          }
        };
        bt.addActionListener(lst);
        p1.add(bt);

        bt = new JButton("Cancel");
        lst = new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            dispose();
          }
        };
        bt.addActionListener(lst);
        p1.add(bt);
        p.add(p1);
        pp.add(p, BorderLayout.SOUTH);

        getContentPane().add(pp, BorderLayout.CENTER);
        pack();
        setResizable(true);
        setLocationRelativeTo(parent);
      }

      public boolean succeeded() {
        return m_succeeded;
      }

      public String getSource() {
        return m_sourceTxt.getText();
      }
    }
```

20.3.1 Understanding the code

Class HtmlProcessor

❶ Two instance variables have changed class type:

- `MutableHTMLDocument m_doc`: the main text component's `Document` is now an instance of our custom `HTMLDocument` subclass.
- `CustomHTMLEditorKit m_kit`: the editor kit is now an instance of our custom `HTML-EditorKit` subclass.

❷ One new instance variable has been added to this example:

- `ColorMenu m_foreground`: used to choose the selected text foreground color.

3 The `createMenuBar()` method adds three new menus. An Insert menu is added with menu items Image and Hyperlink. These menu items are responsible for inserting images and hyperlinks respectively. Their `ActionListeners` both use our custom `inputURL()` method to display a dialog that allows specification of the path to the image or hyperlink. Given a URL to an image or hyperlink, note how we use the `HTML.Attribute` and `HTML.Tag` classes to insert tags and attributes respectively.

Also note that before displaying the input dialog for inserting a hyperlink, our code checks whether there is already a hyperlink at the current caret position. If there is, this hyperlink is displayed in the insert hyperlink dialog so that it may be modified (a more professional implementation might allow us to right-click on the hyperlink to display a popup menu in which one of the options would be to edit it).

4 A Format menu is added with menu items Foreground Color and Page Properties. The Foreground Color item is an instance of our custom `ColorMenu` class developed in chapter 12 as a simple color selection menu item (see section 12.5). This item sets the foreground color attribute of the selected text to the color chosen in the `ColorMenu` component. A `MenuListener` is added to the `ColorMenu` component to set its selected color to the foreground color of the text at the current carret location when this menu is displayed. The Page Properties menu item creates an instance of our custom `DocumentPropsDlg` dialog which is used to change the text and background color page properties, as well as the document title. If changes are made with the `DocumentPropsDlg` we call our `documentChanged()` method to update our editor properly with the modified document.

5 A Tools menu is added with item HTML Source. The HTML Source item displays the HTML source code behind the current document in an instance of our custom `HtmlSourceDlg` dialog. A `StringWriter` is used to convert the current document to a `String` which is passed to the `HtmlSourceDlg` constructor. If changes have been made to the HTML by `HtmlSourceDlg` we use a `StringReader` to bundle the new source from `HtmlSourceDlg`'s `getSource()` method, create a new document using our custom editor kit, read the contents of the `StringReader` into the document, and set this new document as our editor's current `Document` instance. Then we call our `documentChanged()` method to update our editor properly with the new document.

The `getDocumentName()` method is modified to return the contents of the title tag, or if undefined, the current file name.

The `newDocument()` method is modified to create a new instance of our custom `Mutable-HTMLDocument` class using our custom editor kit implementation. The `openDocument()` method is modified similarly.

6 There are now two `setAttributeSet()` methods. The main `setAttributeSet()` method takes an `AttributeSet` parameter and a `boolean` parameter. The `boolean` parameter specifies whether or not the given `AttributeSet` should be applied as paragraph or character attributes. If `true` the attributes will be applied to the currently selected paragraph(s).

7 The `inputURL()` method takes two `String` parameters representing a message to display and an initial value. This method creates a `JOptionPane` with a custom panel and given initial value. The panel consists of a label containing the given message string and a button called Local File. This button is used to navigate the local computer for a file and, if selected, the

path to the file will appear in the JOptionPane's input field. Once the JOptionPane is dismissed the inputURL() method returns the chosen string URL.

8 The documentChanged() method updates the text pane with the current Document instance after it has been changed by an outside source.

BUG ALERT! Unfortunately we were forced to resort to rather barbaric techniques in order to properly handle Document updates that occur outside of the editor. Because simply calling JTextPane's setDocument() method does not properly update the editor, we assign it a completely new Document, then reassign it our modified document and then revalidate and repaint the editor. We encourage you to search for a better solution to this problem if one exists, and we hope that this problem is addressed in a future version of JTextPane.

Class Utils

9 This class consists of the static colorToHex() method which was copied from the javax.swing.text.html.CSS class. This method takes a Color parameter and returns a String representing a hex value used in HTML documents to specify a color.

NOTE We copied this method directly from Swing source code and placed it in this separate class because, unfortunately, the method is private in the CSS class. We are unsure why it is private and hope to see this changed in a future release.

10 *Class CustomHTMLEditorKit*

This class extends HTMLEditorKit and overrides its createDocument() method to return an instance of our custom MutableHTMLDocument class.

11 *Class MutableHTMLDocument*

This class extends HTMLDocument to add functionality for locating Elements corresponding to a specific HTML tag, retrieving and setting the document <title> tag value, and adding attributes to an Element corresponding to a specific HTML tag.

12 *Class DocumentPropsDlg*

This class extends JDialog and has eight instance variables:

- boolean m_succeeded: flag set to true if new attributes are successfully added.
- MutableHTMLDocument m_doc: reference to the HTML document passed into the constructor.
- Color m_backgroundColor: used to store the HTML document's background color.
- Color m_textColor: used to store the HTML document's text foreground color.
- Color m_linkColor: used to store the HTML document's hyperlink color.
- Color m_viewedColor: used to store the HTML document's visited hyperlink color.
- JTextField m_titleTxt: input field used to change the HTML document's title value.
- JTextPane m_previewPane: text pane to preview HTML document color settings.

The DocumentPropsDlg constructor takes JFrame and MutableHTMLDocument parameters. The document passed in represents the HTML document whose properties are to be modified by this dialog. The <body> element is located and the color variables are initialized based on this element's attributes. A panel is created to contain the input field, preview pane,

and a series of buttons used for displaying a JColorChooser to change each color variable and update the preview panel. Whenever a color change is made the showColors() method is called to update the preview panel. A Save button and a Cancel button are also added to this panel to save changes or abort them respectively. This panel is added to the content pane and the dialog is centered with respect to the JFrame parent.

The succeeded() method simply returns m_succeeded which indicates whether or not attempts to change the document's attributes were successful.

The saveData() method is called when the Save button is pressed. This method updates the <body> element's attributes in the HTML document and sets the m_succeeded variable to true if it succeeds.

The showColors() method updates the preview text pane with a new Default-Styled-Document. Text is added to this document with attributes corresponding to the text and hyperlink colors to demonstrate the current selections. The text pane's background is also set to the currently selected background color.

13 *Class HtmlSourceDlg*

This class extends JDialog and has two instance variables:

- boolean m_succeeded: flag set to true if the Save button is pressed.
- JTextArea m_sourceTxt: text area used for displaying and editing HTML source code.

The HtmlSourceDlg constructor takes JFrame and String parameters. The String represents the HTML source code and is placed in the text area. The text area is placed in a JScrollPane and added to a panel. A Save button and a Cancel button are also added to this panel to save changes or abort them respectively. This panel is added to the content pane and the dialog is centered with respect to the JFrame parent.

The succeeded() method simply returns m_succeeded which indicates whether or not attempts to change the document's attributes were successful.

The getSource() method returns the current contents of the JTextArea representing the HTML source code.

20.3.2 Running the code

Figure 20.3 shows our editor with the color menu open. Figures 20.4 through 20.7 show our page properties, HTML source, image, and URL insertion dialogs. Open an existing HTML file, select a portion of text, and use the custom color menu component to modify its foreground. From the Document menu select the page properties dialog and the HTML source dialog to modify internal aspects of the document. Verify that these dialogs work as expected. Try inserting hyperlinks and images. Save the HTML file and open it in another HTML-aware application to verify that your changes have been saved correctly.

20.4 HTML editor, part IV: working with HTML styles and tables

Using Styles to manage a set of attributes as a single named entity can greatly simplify editing. The user only has to apply a known style to a paragraph of text rather than selecting all appropriate text attributes from the provided toolbar components. By adding a combo box allowing the choice of styles, we can not only save the user time and effort, but we can also

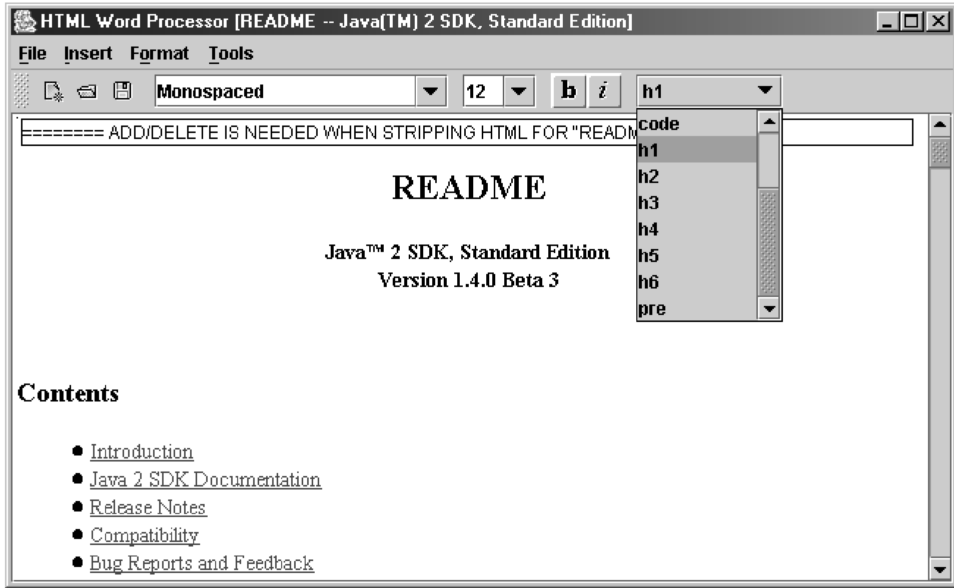

Figure 20.8 HtmlProcessor with style management

provide more uniform text formatting throughout the resulting document. In this section we'll add the following features:

- Ability to apply HTML styles to paragraphs of text.
- A dialog to create HTML tables.

Example 20.4

HtmlProcessor.java

see \Chapter20\4

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

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

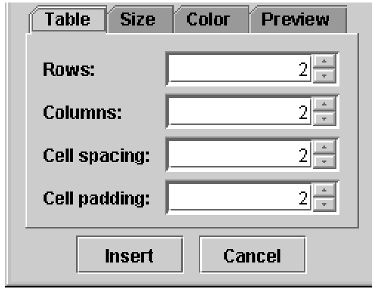

Figure 20.9
`HtmlProcessor`'s table creation dialog–Table pane

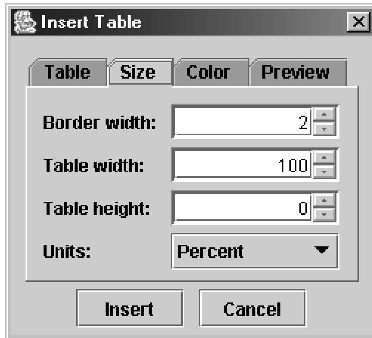

Figure 20.10
`HtmlProcessor`'s table creation dialog–Size pane

```
import dl.*;

public class HtmlProcessor extends JFrame {

  public static final String APP_NAME = "HTML HTML Editor";

  // Unchanged code from example 20.3

  protected JComboBox m_cbStyles;
  public static HTML.Tag[] STYLES = {
    HTML.Tag.P, HTML.Tag.BLOCKQUOTE, HTML.Tag.CENTER,
   HTML.Tag.CITE, HTML.Tag.CODE, HTML.Tag.H1, HTML.Tag.H2,
   HTML.Tag.H3, HTML.Tag.H4, HTML.Tag.H5, HTML.Tag.H6,
   HTML.Tag.PRE };

  public HtmlProcessor() {
   // Unchanged code from example 20.3
  }

  protected JMenuBar createMenuBar() {

// Unchanged code from example 20.3

    item = new JMenuItem("Table...");
    item.setMnemonic('t');
    lst = new ActionListener(){
```

1 New combo box containing HTML styles

2 Menu item to invoke table dialog for creating an HTML table

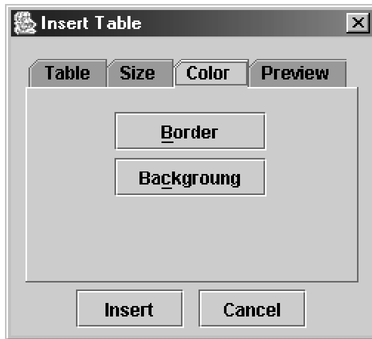

Figure 20.11
`HtmlProcessor`'s table creation dialog–Color pane

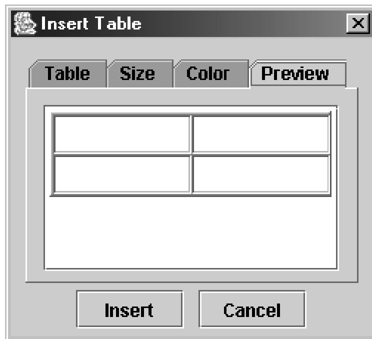

Figure 20.12
`HtmlProcessor`'s table creation dialog–Preview pane

```
      public void actionPerformed(ActionEvent e) {
        TableDlg dlg = new TableDlg(HtmlProcessor.this, m_doc);
        dlg.show();
        if (dlg.succeeded()) {
          String tableHtml = dlg.generateHTML();
          Element ep = m_doc.getParagraphElement(
            m_editor.getSelectionStart());
          try {
            m_doc.insertAfterEnd(ep, tableHtml);
          }
          catch (Exception ex) {
            ex.printStackTrace();
          }
          documentChanged();
        }
      }
    };
    item.addActionListener(lst);
    mInsert.add(item);

    menuBar.add(mInsert);

    JMenu mFormat = new JMenu("Format");
    mFormat.setMnemonic('o');

  // Unchanged code from example 20.3
```

```
      m_toolBar.addSeparator();
      m_cbStyles = new JComboBox(STYLES);
      m_cbStyles.setMaximumSize(new Dimension(100, 23));
      m_cbStyles.setRequestFocusEnabled(false);
      m_toolBar.add(m_cbStyles);

      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          HTML.Tag style = (HTML.Tag)m_cbStyles.getSelectedItem();
          if (style == null)
            return;
          MutableAttributeSet attr = new SimpleAttributeSet();
          attr.addAttribute(StyleConstants.NameAttribute, style);
          setAttributeSet(attr, true);
          m_editor.grabFocus();
        }
      };
      m_cbStyles.addActionListener(lst);

      getContentPane().add(m_toolBar, BorderLayout.NORTH);

      return menuBar;
    }

// Unchanged code from example 20.3

    protected void showAttributes(int p) {

    // Unchanged code from example 20.3

      Element ep = m_doc.getParagraphElement(p);
      HTML.Tag attrName = (HTML.Tag)ep.getAttributes().
        getAttribute(StyleConstants.NameAttribute);

      int index = -1;
      if (attrName != null) {
        for (int k=0; k<STYLES.length; k++) {
          if (STYLES[k].equals(attrName)) {
            index = k;
            break;
          }
        }
      }
      m_cbStyles.setSelectedIndex(index);

      m_skipUpdate = false;
    }

// Unchanged code from example 20.3

}

// Unchanged code from example 20.3

class TableDlg extends JDialog {
  protected boolean m_succeeded = false;
  protected MutableHTMLDocument m_doc;
```

3 Styles combo box applies selected HTML.Tag to current paragraph

4 Custom dialog to create an HTML table allowing specification of # of rows, # of columns, cell spacing, cell padding, border width, table width, table height, table units, border color, and background color; also includes a preview of proposed table

```
protected JSpinner m_rowsSpn;
protected JSpinner m_colsSpn;
protected JSpinner m_spacingSpn;
protected JSpinner m_paddingSpn;

protected JSpinner m_borderWidthSpn;
protected JSpinner m_tableWidthSpn;
protected JSpinner m_tableHeightSpn;
protected JComboBox m_tableUnitsCb;

protected JTextPane m_previewPane;

protected Color m_borderColor;
protected Color m_backgroundColor;

protected HTMLEditorKit m_kit = new HTMLEditorKit();

public TableDlg(JFrame parent, MutableHTMLDocument doc) {
  super(parent, "Insert Table", true);
  m_doc = doc;

  ActionListener lst;
  JButton bt;

  JPanel pp = new JPanel(new DialogLayout2());
  pp.setBorder(new EmptyBorder(10, 10, 5, 10));

  JPanel p1 = new JPanel(new DialogLayout2());
  p1.setBorder(new EmptyBorder(10, 10, 5, 10));

  p1.add(new JLabel("Rows:"));
  m_rowsSpn = new JSpinner(new SpinnerNumberModel(
    new Integer(2), new Integer(0), null, new Integer(1)));
  p1.add(m_rowsSpn);

  p1.add(new JLabel("Columns:"));
  m_colsSpn = new JSpinner(new SpinnerNumberModel(
    new Integer(2), new Integer(0), null, new Integer(1)));
  p1.add(m_colsSpn);

  p1.add(new JLabel("Cell spacing:"));
  m_spacingSpn = new JSpinner(new SpinnerNumberModel(
    new Integer(2), new Integer(0), null, new Integer(1)));
  p1.add(m_spacingSpn);

  p1.add(new JLabel("Cell padding:"));
  m_paddingSpn = new JSpinner(new SpinnerNumberModel(
    new Integer(2), new Integer(0), null, new Integer(1)));
  p1.add(m_paddingSpn);

  JPanel p2 = new JPanel(new DialogLayout2());
  p2.setBorder(new EmptyBorder(10, 10, 5, 10));

  p2.add(new JLabel("Border width:"));
  m_borderWidthSpn = new JSpinner(new SpinnerNumberModel(
    new Integer(2), new Integer(0), null, new Integer(1)));
  p2.add(m_borderWidthSpn);

  p2.add(new JLabel("Table width:"));
```

```
m_tableWidthSpn = new JSpinner(new SpinnerNumberModel(
  new Integer(100), new Integer(0), null, new Integer(1)));
p2.add(m_tableWidthSpn);

p2.add(new JLabel("Table height:"));
m_tableHeightSpn = new JSpinner(new SpinnerNumberModel(
  new Integer(0), new Integer(0), null, new Integer(1)));
p2.add(m_tableHeightSpn);

p2.add(new JLabel("Units:"));
m_tableUnitsCb = new JComboBox(new String[]
 {"Percent", "Pixels" });
p2.add(m_tableUnitsCb);

JPanel p3 = new JPanel(new FlowLayout());
p3.setBorder(new EmptyBorder(10, 10, 5, 10));
JPanel pb = new JPanel(new GridLayout(2, 1, 5, 5));
p3.add(pb);

bt = new JButton("Border");
bt.setMnemonic('b');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_borderColor = JColorChooser.showDialog(
    TableDlg.this, "Border Color", m_borderColor);
  }
};
bt.addActionListener(lst);
pb.add(bt);

bt = new JButton("Background");
bt.setMnemonic('c');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_backgroundColor = JColorChooser.showDialog(
    TableDlg.this, "Background Color", m_backgroundColor);
  }
};
bt.addActionListener(lst);
pb.add(bt);

JPanel p4 = new JPanel(new BorderLayout());
p4.setBorder(new EmptyBorder(10, 10, 5, 10));

m_previewPane = new JTextPane();
m_previewPane.setEditorKit(m_kit);
m_previewPane.setBackground(Color.white);
m_previewPane.setEditable(false);
JScrollPane sp = new JScrollPane(m_previewPane);
sp.setPreferredSize(new Dimension(200, 100));
p4.add(sp, BorderLayout.CENTER);

final JTabbedPane tb = new JTabbedPane();
tb.addTab("Table", p1);
tb.addTab("Size", p2);
tb.addTab("Color", p3);
```

```
tb.addTab("Preview", p4);
pp.add(tb);

ChangeListener chl = new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
    if (tb.getSelectedIndex() != 3)
      return;
    setCursor(Cursor.getPredefinedCursor(
      Cursor.WAIT_CURSOR));
    try {
      HTMLDocument doc =
      (HTMLDocument)m_kit.createDefaultDocument();
      doc.setAsynchronousLoadPriority(0);
      StringReader sr = new StringReader(generateHTML());
      m_kit.read(sr, doc, 0);
      sr.close();

      m_previewPane.setDocument(doc);
      validate();
      repaint();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
    finally {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
    }
  }
};
tb.addChangeListener(chl);

bt = new JButton("Insert");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_succeeded = true;
    dispose();
  }
};
bt.addActionListener(lst);
pp.add(bt);

bt = new JButton("Cancel");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    dispose();
  }
};
bt.addActionListener(lst);
pp.add(bt);

getContentPane().add(pp, BorderLayout.CENTER);
pack();
setResizable(true);
```

```
    setLocationRelativeTo(parent);
  }

  public boolean succeeded() {
    return m_succeeded;
  }

  protected String generateHTML() {
    StringBuffer buff = new StringBuffer();
    buff.append("<table");

    int tableWidth =
     ((Integer) m_tableWidthSpn.getValue()).intValue();
    int tableHeight =
     ((Integer)m_tableHeightSpn.getValue()).intValue();
    String unit = "";
    if (m_tableUnitsCb.getSelectedIndex()==0)
      unit = "%";
    if (tableWidth > 0)
      buff.append(" width=\"").append(
     tableWidth).append(unit).append("\"");
    if (tableHeight > 0)
      buff.append(" height=\"").append(
     tableHeight).append(unit).append("\"");

    buff.append(" cellspacing=\"").append(
    m_spacingSpn.getValue()).append("\"");
    buff.append(" cellpadding=\"").append(
    m_paddingSpn.getValue()).append("\"");
    buff.append(" border=\"").append(
    m_borderWidthSpn.getValue()).append("\"");
    if (m_borderColor != null)
      buff.append(" bordercolor=\"").append(
     Utils.colorToHex(m_borderColor)).append("\"");
    if (m_backgroundColor != null)
      buff.append(" bgcolor=\"").append(
     Utils.colorToHex(m_backgroundColor)).append("\"");
    buff.append(">\n");

    int nRows = ((Integer)m_rowsSpn.getValue()).intValue();
    int nCols = ((Integer)m_colsSpn.getValue()).intValue();
    for (int k=0; k<nRows; k++) {
      buff.append("<tr>\n");
      for (int s=0; s<nCols; s++)
        buff.append("<td> </td>\n");
      buff.append("</tr>\n");
    }

    buff.append("</table>\n");
    return buff.toString();
  }
}
```

20.4.1 Understanding the code

Class HtmlProcessor

❶ One new instance variable has been added:

- `JComboBox m_cbStyles`: toolbar component to apply HTML styles.

A new static array of `HTML.Tags` is also added:

- `HTML.Tag[] STYLES`: used to hold tags for all HTML styles.

❷ The `createMenuBar()` method creates a new Table menu item added to the Insert menu, and a new combo box for HTML style selection is added to the toolbar. The Table menu item displays an instance of our custom `TableDlg` class for inserting a new HTML table. This item's `ActionListener` is responsible for creating the `TableDlg` instance and inserting the resulting HTML code, retrieved using `TableDlg`'s `generateHTML()` method, after the paragraph the cursor is currently in.

❸ The editable styles combo box, `m_cbStyles`, holds the STYLES list of HTML styles. It receives an `ActionListener` which applies the selected `HTML.Tag` to the paragraph the cursor currently resides in.

The `showAttributes()` method receives additional code to manage the new style's combo box when the caret moves through the document. It retrieves the style corresponding to the paragraph based on caret position and selects the appropriate entry in the combo box.

❹ *Class TableDlg*

This class extends `JDialog` and has several instance variables:

- `boolean m_succeeded`: flag set to `true` if the Save button is pressed.
- `MutableHTMLDocument m_doc`: reference to the HTML document passed into the constructor.
- `JSpinner m_rowsSpn`: used to select number of table rows.
- `JSpinner m_colsSpn`: used to select number of table columns.
- `JSpinner m_spacingSpn`: used to select table cell spacing size.
- `JSpinner m_paddingSpn`: used to select table cell padding size.
- `JSpinner m_borderWidthSpn`: used to select table border width.
- `JSpinner m_tableWidthSpn`: used to select table width.
- `JSpinner m_tableHeightSpn`: used to select table height.
- `JComboBox m_tableUnitsCb`: used to choose units with which to measure HTML table dimensions (either percentage of available space or pixels).
- `JTextPane m_previewPane`: text component to display a preview of the HTML table.
- `Color m_borderColor`: used to maintain the HTML table's border color.
- `Color m_backgroundColor`: used to maintain the HTML table's background color.
- `HTMLEditorKit m_kit`: editor kit used to create new preview pane document.

The `TableDlg` constructor takes `JFrame` and `MutableHTMLDocument` as parameters. The document represents the HTML document to which a table will be added. A `JTabbedPane` is created with four tabs: Table, Size, Color, and Preview. Each of these tabs receives its own panel of components.

The Table tab consists of four `JSpinners` used to select number of table rows and columns, and values for table cell spacing and padding.

The Size tab consists of three `JSpinners` used to select table border width, table width, and table height. It also contains a `JComboBox` used to select whether the spinner values for table width and height in this tab are using Percent or Pixels as units. Percent refers to percentage of available space the table should occupy, whereas Pixels refers to actual pixel values.

The Color tab contains buttons called Border, and Background which are responsible for setting the table border color and table background colors respectively through use of a `JColorChooser`.

The Preview tab consists of a text pane to show a preview of the proposed HTML table. A `ChangeListener` is added to the tabbed pane to detect when the Preview tab is selected and update the text pane in response using our custom `generateHTML()` method.

An Insert button and a Cancel button are also added to this dialog. The Insert button sets the `m_succeeded` flag to `true` before the dialog is disposed; the Cancel button simply diposes of the dialog.

The `getSucceeded()` method returns the `m_succeeded` flag.

The `generateHTML()` method returns a `String` representing the HTML code for a table based on the current values of the input components in the Table, Size, and Color tabs.

20.4.2 Running the code

Figure 20.8 shows our editor with the styles combo box open. Figures 20.9 through 20.12 show each tab of our HTML table dialog. Open an existing HTML file and verify that the selected style is automatically updated while the caret moves through the document. Place the caret in a different paragraph and select a different style from the styles combo box. Note how all text properties are updated according to the new style. Use the Insert menu and select Table to insert an HTML table.

20.5 HTML EDITOR, PART V: CLIPBOARD AND UNDO/REDO

Clipboard and undo/redo operations have become common and necessary components of all modern text editing environments. We have discussed these features in chapters 11 and 19, and in this section we show how to integrate them into our HTML editor application.

Example 20.5

HtmlProcessor.java

see \Chapter20\5

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

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

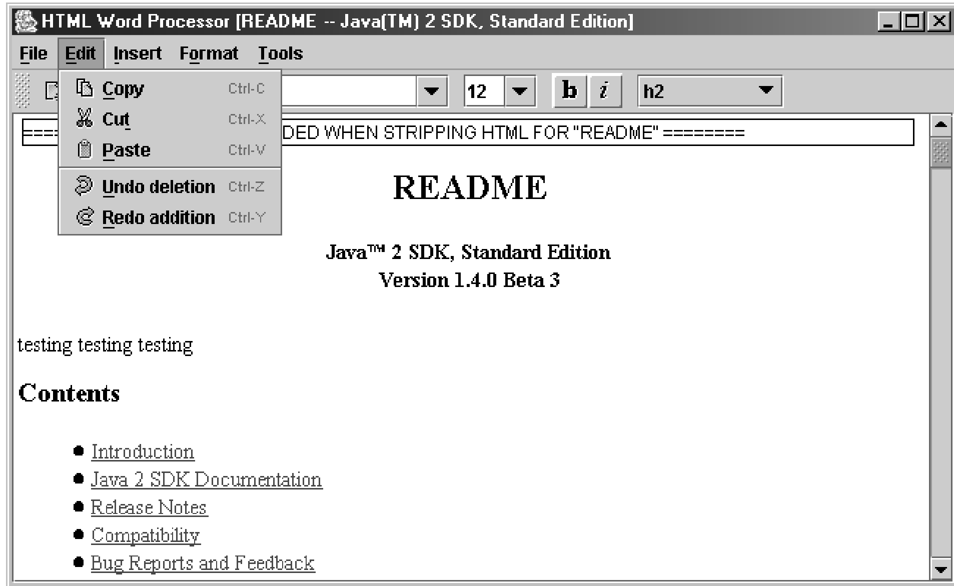

Figure 20.13 `HtmlProcessor` **with undo/redo and clipboard functionality**

```
import javax.swing.undo.*;

import dl.*;

public class HtmlProcessor extends JFrame {

   // Unchanged code from example 20.4

   protected UndoManager m_undo = new UndoManager();
   protected Action m_undoAction;
   protected Action m_redoAction;

   // Unchanged code from example 20.4

   protected JMenuBar createMenuBar() {
      JMenuBar menuBar = new JMenuBar();

      // Unchanged code from example 20.4


      JMenu mEdit = new JMenu("Edit");
      mEdit.setMnemonic('e');

         Action action = new AbstractAction("Copy",
       new ImageIcon("Copy16.gif"))
      {
         public void actionPerformed(ActionEvent e) {
            m_editor.copy();
         }
      };
      item = mEdit.add(action);
```

1 Action to invoke a copy of currently selected text

```
item.setMnemonic('c');
item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
  KeyEvent.Ctrl_MASK));

  action = new AbstractAction("Cut",
 new ImageIcon("Cut16.gif"))
{
  public void actionPerformed(ActionEvent e) {
    m_editor.cut();
  }
};
item = mEdit.add(action);
item.setMnemonic('t');
item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_X,
  KeyEvent.Ctrl_MASK));

  action = new AbstractAction("Paste",
 new ImageIcon("Paste16.gif"))
{
  public void actionPerformed(ActionEvent e) {
    m_editor.paste();}
};
item = mEdit.add(action);
item.setMnemonic('p');
item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V,
  KeyEvent.Ctrl_MASK));

  mEdit.addSeparator();

  m_undoAction = new AbstractAction("Undo",
 new ImageIcon("Undo16.gif"))
{
  public void actionPerformed(ActionEvent e) {
    try {
      m_undo.undo();
    }
    catch (CannotUndoException ex) {
      System.err.println("Unable to undo: " + ex);
    }
    updateUndo();
  }
};
item = mEdit.add(m_undoAction);
item.setMnemonic('u');
item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Z,
  KeyEvent.Ctrl_MASK));

  m_redoAction = new AbstractAction("Redo",
 new ImageIcon("Redo16.gif"))
{
  public void actionPerformed(ActionEvent e) {
    try {
      m_undo.redo();
    }
```

1 Action to invoke a copy of currently selected text

1 Action to invoke a cut of currently selected text

1 Action to invoke a paste of current clipboard text

2 Action to invoke an Undo

2 Action to invoke a Redo

```
          catch (CannotRedoException ex) {
            System.err.println("Unable to redo: " + ex);
          }
          updateUndo();
        }
      };
      item =mEdit.add(m_redoAction);
      item.setMnemonic('r');
      item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Y,
        KeyEvent.Ctrl_MASK));

        menuBar.add(mEdit);

      // Unchanged code from example 20.4

      return menuBar;
    }
    // Unchanged code from example 20.4

    protected void newDocument() {
      // Unchanged code from example 20.4

      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          showAttributes(0);
          m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
          m_doc.addDocumentListener(new UpdateListener());
          m_doc.addUndoableEditListener(new Undoer());
          m_textChanged = false;
        }
      });
    }

    protected void openDocument() {
     // Unchanged code from example 20.4

      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          m_editor.setCaretPosition(1);
          showAttributes(1);
          m_editor.scrollRectToVisible(new Rectangle(0,0,1,1));
          m_doc.addDocumentListener(new UpdateListener());
          m_doc.addUndoableEditListener(new Undoer());
          m_textChanged = false;
        }
      });
    }

  // Unchanged code from example 20.4

    protected void updateUndo() {
      if(m_undo.canUndo()) {
        m_undoAction.setEnabled(true);
        m_undoAction.putValue(Action.NAME,
        m_undo.getUndoPresentationName());
      }
```

2 Action to invoke a Redo

3 Updates undo and redo Actions based on undo stack

```
        else {
          m_undoAction.setEnabled(false);
          m_undoAction.putValue(Action.NAME, "Undo");
        }
        if(m_undo.canRedo()) {
          m_redoAction.setEnabled(true);
          m_redoAction.putValue(Action.NAME,
          m_undo.getRedoPresentationName());
        }
        else {
          m_redoAction.setEnabled(false);
          m_redoAction.putValue(Action.NAME, "Redo");
        }
      }

      public static void main(String argv[]) {
        HtmlProcessor frame = new HtmlProcessor();
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        frame.setVisible(true);
      }

    // Unchanged code from example 20.4

      class Undoer implements UndoableEditListener {
        public Undoer() {
          m_undo.die();
          updateUndo();
        }

        public void undoableEditHappened(UndoableEditEvent e) {
          UndoableEdit edit = e.getEdit();
          m_undo.addEdit(e.getEdit());
          updateUndo();
        }
      }
    }

    // Unchanged code from example 20.4
```

4 Adds undoable edit events to UndoManager and updates state of undo/redo components

20.5.1 Understanding the code

Class HtmlProcessor

We now import the `javax.swing.undo` package and add three new instance variables:

- `UndoManager m_undo`: used to manage undo/redo operations.
- `Action m_undoAction`: used for a menu item/action to perform undo operations.
- `Action m_redoAction`: used for a menu item/action to perform redo operations.

1 The `createMenuBar()` method now creates a menu titled Edit (which traditionally follows the File menu) containing menu items titled Copy, Cut, Paste, Undo, and Redo. The first three items merely trigger calls to the `copy()`, `cut()`, and `paste()` methods of our `m_editor` text pane. These methods perform basic clipboard operations. They are available when the editor has the current focus and the appropriate keyboard accelerator is pressed.

The Undo menu item is created from an `AbstractAction` whose `actionPerformed()` method first invokes `undo()` on the `UndoManager`, and then invokes our custom `updateUndo()` method to update our Undo/Redo menu items appropriately. Similarly, the Redo menu item is created from an `AbstractAction` which invokes `redo()` on the `UndoManager`, and then calls our `updateUndo()` method.

The `newDocument()` and `openDocument()` methods now add an instance of our custom `Undoer` class as an `UndoableEditListener` to all newly created or loaded documents.

The `updateUndo()` method enables or disables the Undo and Redo menu items, and updates their names according to the operation which can be undone/redone (if any). If the `Undo-Manager`'s `canUndo()` method returns `true`, the `m_undoAction` is enabled and its name is set to the string returned by `getUndoPresentationName()`. Otherwise it is disabled and its name is set to Undo. The Redo menu item is handled similarly.

Class HtmlProcessor.Undoer

This inner class implements the `UndoableEditListener` interface to receive notifications about undoable operations. The `undoableEditHappened()` method receives `UndoableEditEvents`, retrieves their encapsulated `UndoableEdit` instances, and passes them to the `UndoManager`. The `updateUndo()` method is also invoked to update the undo/redo menu items appropriately.

20.5.2 Running the code

Figure 20.13 shows our editor with the Edit menu open. Open an existing HTML file and verify that copy, cut, and paste clipboard operations transfer text successfully. Make some changes to the textual content or styles and note that the title of the Undo menu item is updated. Select this menu item, or press its keyboard accelerator (Ctrl-Z) to undo a series of changes. This will enable the Redo menu item. Use this menu item or press its keyboard accelerator (Ctrl-Y) to redo a series of changes.

20.6 HTML EDITOR, PART VI: ADVANCED FONT MANAGEMENT

In section 20.2 we used toolbar components to change font properties. This is useful for making a quick modification without leaving the main application frame, and is typical for word processor applications. However, all serious editor applications also provide a dialog for the editing of all available font properties from one location. In this section's example we'll show how to create such a dialog, which includes components to select various font properties and preview the result.

Example 20.6

HtmlProcessor.java

see \Chapter20\6

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

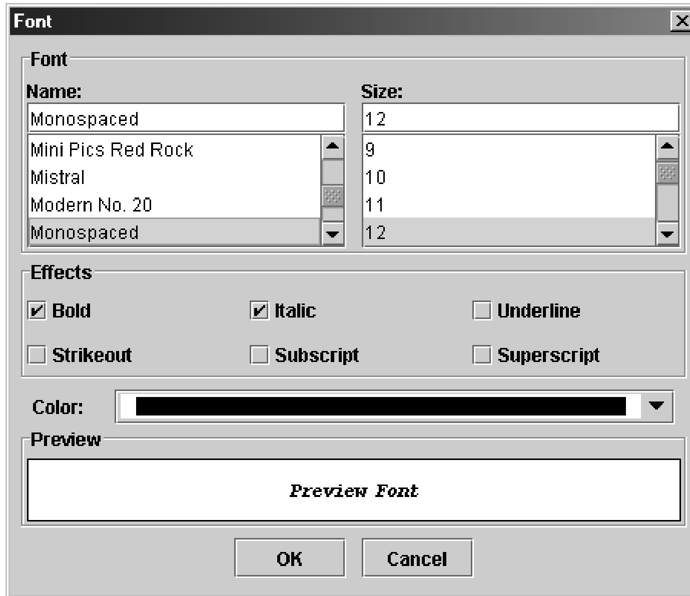

Figure 20.14 Font properties and preview dialog

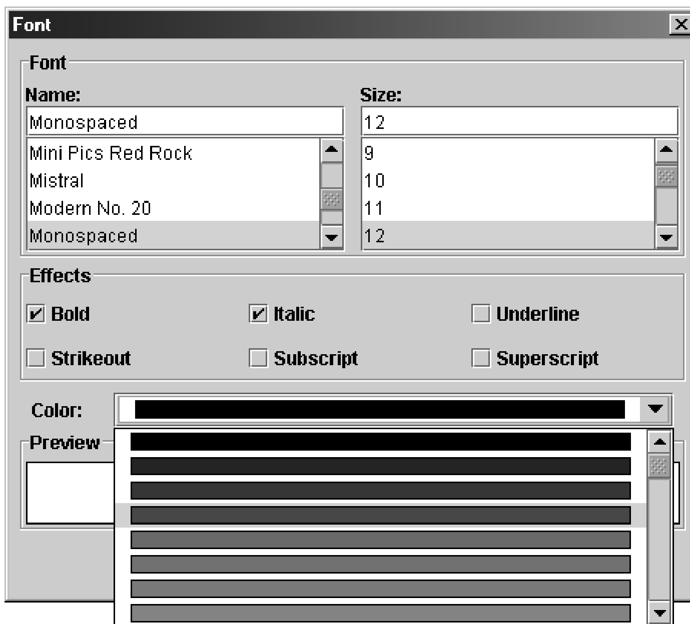

Figure 20.15 Font dialog displaying custom list and list cell renderer for foreground color selection

```
import java.util.*;

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

import dl.*;

public class HtmlProcessor extends JFrame {

  public static final String APP_NAME = "HTML HTML Editor";

// Unchanged code from example 20.5

  protected String[] m_fontNames;
  protected String[] m_fontSizes;

  // Unchanged code from example 20.5

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

// Unchanged code from example 20.5

    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    m_fontNames = ge.getAvailableFontFamilyNames();

    m_toolBar.addSeparator();
    m_cbFonts = new JComboBox(m_fontNames);
    m_cbFonts.setMaximumSize(new Dimension(200, 23));
    m_cbFonts.setEditable(true);

    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_fontName = m_cbFonts.getSelectedItem().toString();
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setFontFamily(attr, m_fontName);
        setAttributeSet(attr);
        m_editor.grabFocus();
      }
    };
    m_cbFonts.addActionListener(lst);

    m_toolBar.add(m_cbFonts);

    m_toolBar.addSeparator();
    m_fontSizes = new String[] {"8", "9", "10",
      "11", "12", "14", "16", "18", "20", "22", "24", "26",
      "28", "36", "48", "72"};
    m_cbSizes = new JComboBox(m_fontSizes);
    m_cbSizes.setMaximumSize(new Dimension(50, 23));
    m_cbSizes.setEditable(true);
```

```
    // Unchanged code from example 20.5

    item = new JMenuItem("Font...");
    item.setMnemonic('o');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        FontDialog dlg = new FontDialog(HtmlProcessor.this,
          m_fontNames, m_fontSizes);
        AttributeSet a = m_doc.getCharacterElement(
          m_editor.getCaretPosition()).getAttributes();
        dlg.setAttributes(a);
        dlg.show();
        if (dlg.succeeded()) {
          setAttributeSet(dlg.getAttributes());
          showAttributes(m_editor.getCaretPosition());
        }
      }
    };
    item.addActionListener(lst);
    mFormat.add(item);
    mFormat.addSeparator();

    return menuBar;
  }

// Unchanged code from example 20.5

}

// Unchanged code from example 20.5

class FontDialog extends JDialog {
  protected boolean m_succeeded = false;
  protected OpenList m_lstFontName;
  protected OpenList m_lstFontSize;
  protected MutableAttributeSet m_attributes;
  protected JCheckBox m_chkBold;
  protected JCheckBox m_chkItalic;
  protected JCheckBox m_chkUnderline;

  protected JCheckBox m_chkStrikethrough;
  protected JCheckBox m_chkSubscript;
  protected JCheckBox m_chkSuperscript;

  protected JComboBox m_cbColor;
  protected JLabel m_preview;

  public FontDialog(JFrame parent,
    String[] names, String[] sizes)
  {
    super(parent, "Font", true);
    JPanel pp = new JPanel();
    pp.setBorder(new EmptyBorder(5,5,5,5));
    pp.setLayout(new BoxLayout(pp, BoxLayout.Y_AXIS));

    JPanel p = new JPanel(new GridLayout(1, 2, 10, 2));
    p.setBorder(new TitledBorder(new EtchedBorder(), "Font"));
```

1 New menu item to invoke custom FontDialog for font management

2 Custom font dialog allows specification of font properties such as size, name, bold, italic, underline, strikethrough, subscript, superscript, color and also includes a preview illustrating selections

```
m_lstFontName = new OpenList(names, "Name:");
p.add(m_lstFontName);

m_lstFontSize = new OpenList(sizes, "Size:");
p.add(m_lstFontSize);
pp.add(p);

p = new JPanel(new GridLayout(2, 3, 10, 5));
p.setBorder(new TitledBorder(new EtchedBorder(), "Effects"));
m_chkBold = new JCheckBox("Bold");
p.add(m_chkBold);
m_chkItalic = new JCheckBox("Italic");
p.add(m_chkItalic);
m_chkUnderline = new JCheckBox("Underline");
p.add(m_chkUnderline);
m_chkStrikethrough = new JCheckBox("Strikeout");
p.add(m_chkStrikethrough);
m_chkSubscript = new JCheckBox("Subscript");
p.add(m_chkSubscript);
m_chkSuperscript = new JCheckBox("Superscript");
p.add(m_chkSuperscript);
pp.add(p);
pp.add(Box.createVerticalStrut(5));

p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
p.add(Box.createHorizontalStrut(10));
p.add(new JLabel("Color:"));
p.add(Box.createHorizontalStrut(20));
m_cbColor = new JComboBox();

int[] values = new int[] { 0, 128, 192, 255 };
for (int r=0; r<values.length; r++) {
  for (int g=0; g<values.length; g++) {
    for (int b=0; b<values.length; b++) {
      Color c = new Color(values[r], values[g], values[b]);

m_cbColor.addItem(c);
    }
  }
}

m_cbColor.setRenderer(new ColorComboRenderer());
p.add(m_cbColor);
p.add(Box.createHorizontalStrut(10));
pp.add(p);

ListSelectionListener lsel = new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    updatePreview();
  }
};
m_lstFontName.addListSelectionListener(lsel);
m_lstFontSize.addListSelectionListener(lsel);
```

3 Custom OpenList components used to select font name and size

3 Check boxes for various font properties

3 Create combo box used to select font color using custom ColorComboRenderer

5 Updates preview component wherever font name or size changes

```
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        updatePreview();
      }
    };
    m_chkBold.addActionListener(lst);
    m_chkItalic.addActionListener(lst);
    m_cbColor.addActionListener(lst);

    p = new JPanel(new BorderLayout());
    p.setBorder(new TitledBorder(new EtchedBorder(), "Preview"));
    m_preview = new JLabel("Preview Font", JLabel.CENTER);
    m_preview.setBackground(Color.white);
    m_preview.setForeground(Color.black);
    m_preview.setOpaque(true);
    m_preview.setBorder(new LineBorder(Color.black));
    m_preview.setPreferredSize(new Dimension(120, 40));
    p.add(m_preview, BorderLayout.CENTER);
    pp.add(p);

    p = new JPanel(new FlowLayout());
    JPanel p1 = new JPanel(new GridLayout(1, 2, 10, 0));
    JButton btOK = new JButton("OK");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_succeeded = true;
        dispose();
      }
    };
    btOK.addActionListener(lst);
    p1.add(btOK);

    JButton btCancel = new JButton("Cancel");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dispose();
      }
    };
    btCancel.addActionListener(lst);
    p1.add(btCancel);
    p.add(p1);
    pp.add(p);

    getContentPane().add(pp, BorderLayout.CENTER);
    pack();
    setResizable(false);
    setLocationRelativeTo(parent);
  }

  public void setAttributes(AttributeSet a) {
    m_attributes = new SimpleAttributeSet(a);
    String name = StyleConstants.getFontFamily(a);
    m_lstFontName.setSelected(name);
    int size = StyleConstants.getFontSize(a);
```

4 **Preview panel showing sample of selected font attributes changes**

6 **Placing buttons inside GridLayout inside FlowLayout ensures the buttons are equally sized and centered**

7 **Used to assign initial attribute to selection components when font dialog is invoked**

```
    m_lstFontSize.setSelectedInt(size);
    m_chkBold.setSelected(StyleConstants.isBold(a));
    m_chkItalic.setSelected(StyleConstants.isItalic(a));
    m_chkUnderline.setSelected(StyleConstants.isUnderline(a));
    m_chkStrikethrough.setSelected(
      StyleConstants.isStrikeThrough(a));
    m_chkSubscript.setSelected(StyleConstants.isSubscript(a));
    m_chkSuperscript.setSelected(StyleConstants.isSuperscript(a));
    m_cbColor.setSelectedItem(StyleConstants.getForeground(a));
    updatePreview();
  }

  public AttributeSet getAttributes() {
    if (m_attributes == null)
      return null;
    StyleConstants.setFontFamily(m_attributes,
      m_lstFontName.getSelected());
    StyleConstants.setFontSize(m_attributes,
      m_lstFontSize.getSelectedInt());
    StyleConstants.setBold(m_attributes,
      m_chkBold.isSelected());
    StyleConstants.setItalic(m_attributes,
      m_chkItalic.isSelected());
    StyleConstants.setUnderline(m_attributes,
      m_chkUnderline.isSelected());
    StyleConstants.setStrikeThrough(m_attributes,
      m_chkStrikethrough.isSelected());
    StyleConstants.setSubscript(m_attributes,
      m_chkSubscript.isSelected());
    StyleConstants.setSuperscript(m_attributes,
      m_chkSuperscript.isSelected());
    StyleConstants.setForeground(m_attributes,
      (Color)m_cbColor.getSelectedItem());
    return m_attributes;
  }

  public boolean succeeded() {
    return m_succeeded;
  }

  protected void updatePreview() {
    String name = m_lstFontName.getSelected();
    int size = m_lstFontSize.getSelectedInt();
    if (size <= 0)
      return;
    int style = Font.PLAIN;
    if (m_chkBold.isSelected())
      style |= Font.BOLD;
    if (m_chkItalic.isSelected())
      style |= Font.ITALIC;

    // Bug Alert! This doesn't work if only style is changed.
    Font fn = new Font(name, style, size);
    m_preview.setFont(fn);
```

8 **Used to retrieve current attributes selected in the font dialog**

9 **Updates preview panel with current font dialog selections**

```java
      Color c = (Color)m_cbColor.getSelectedItem();
      m_preview.setForeground(c);
      m_preview.repaint();
    }
  }

  class OpenList extends JPanel
    implements ListSelectionListener, ActionListener
  {
    protected JLabel m_title;
    protected JTextField m_text;
    protected JList m_list;
    protected JScrollPane m_scroll;

    public OpenList(String[] data, String title) {
      setLayout(null);
      m_title = new JLabel(title, JLabel.LEFT);
      add(m_title);
      m_text = new JTextField();
      m_text.addActionListener(this);
      add(m_text);
      m_list = new JList(data);
      m_list.setVisibleRowCount(4);
      m_list.addListSelectionListener(this);
      m_list.setFont(m_text.getFont());
      m_scroll = new JScrollPane(m_list);
      add(m_scroll);
    }

    public void setSelected(String sel) {
      m_list.setSelectedValue(sel, true);
      m_text.setText(sel);
    }

    public String getSelected() { return m_text.getText(); }

    public void setSelectedInt(int value) {
      setSelected(Integer.toString(value));
    }

    public int getSelectedInt() {
      try {
        return Integer.parseInt(getSelected());
      }
      catch (NumberFormatException ex) { return -1; }
    }

    public void valueChanged(ListSelectionEvent e) {
      Object obj = m_list.getSelectedValue();
      if (obj != null)
        m_text.setText(obj.toString());
    }

    public void actionPerformed(ActionEvent e) {
      ListModel model = m_list.getModel();
      String key = m_text.getText().toLowerCase();
```

10 **Custom component resembling a permanently open combo box**

```
    for (int k=0; k<model.getSize(); k++) {
      String data = (String)model.getElementAt(k);
      if (data.toLowerCase().startsWith(key)) {
        m_list.setSelectedValue(data, true);
        break;
      }
    }
  }

  public void addListSelectionListener(ListSelectionListener lst) {
    m_list.addListSelectionListener(lst);
  }

  public Dimension getPreferredSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getPreferredSize();
    Dimension d2 = m_text.getPreferredSize();
    Dimension d3 = m_scroll.getPreferredSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public Dimension getMaximumSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getMaximumSize();
    Dimension d2 = m_text.getMaximumSize();
    Dimension d3 = m_scroll.getMaximumSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public Dimension getMinimumSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getMinimumSize();
    Dimension d2 = m_text.getMinimumSize();
    Dimension d3 = m_scroll.getMinimumSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public void doLayout() {
    Insets ins = getInsets();
    Dimension d = getSize();
    int x = ins.left;
    int y = ins.top;
    int w = d.width-ins.left-ins.right;
    int h = d.height-ins.top-ins.bottom;

    Dimension d1 = m_title.getPreferredSize();
```

11 **All layout at this component is handled here**

```
      m_title.setBounds(x, y, w, d1.height);
      y += d1.height;
      Dimension d2 = m_text.getPreferredSize();
      m_text.setBounds(x, y, w, d2.height);
      y += d2.height;
      m_scroll.setBounds(x, y, w, h-y);
    }
}

class ColorComboRenderer extends JPanel implements ListCellRenderer
{
   protected Color m_color = Color.black;
   protected Color m_focusColor =
      (Color) UIManager.get("List.selectionBackground");
   protected Color m_nonFocusColor = Color.white;

   public Component getListCellRendererComponent(JList list,
    Object obj, int row, boolean sel, boolean hasFocus)
   {
      if (hasFocus || sel)
         setBorder(new CompoundBorder(
            new MatteBorder(2, 10, 2, 10, m_focusColor),
            new LineBorder(Color.black)));
      else
         setBorder(new CompoundBorder(
            new MatteBorder(2, 10, 2, 10, m_nonFocusColor),
            new LineBorder(Color.black)));

      if (obj instanceof Color)
         m_color = (Color) obj;
      return this;
   }

   public void paintComponent(Graphics g) {
      setBackground(m_color);
      super.paintComponent(g);
   }
}
```

Custom list cell renderer used to display Colors in a presentable fashion (12)

20.6.1 Understanding the code

Class HtmlProcessor

Two new instance variables are added:

- `String[] m_fontNames`: array of available font family names.
- `String[] m_fontSizes`: array of font sizes.

These arrays were used earlier as local variables to create the toolbar combo box components. Since we need to use them in our font dialog as well, we decided to make them instance variables (this requires minimal changes to the `createMenuBar()` method).

> **NOTE** Reading the list of available fonts takes a significant amount of time. For performance reasons it is best to do this only once in an application's lifetime.

1. A new menu item titled Font… is now added to the Format menu. When the corresponding `ActionListener` is invoked an instance of our custom `FontDialog` is created and the attributes of the character element corresponding to the current caret position are retrieved as an `AttributeSet` instance and passed to the dialog for selection (using its `setAttributes()` method), and the dialog is centered relative to the parent frame and displayed. If the dialog is closed with the OK button (determined by checking whether its `succeeded()` method returns `true`), we retrieve the new font attributes with `FontDialog`'s `getAttributes()` method, and assign these attributes to the selected text with our `setAttributeSet()` method. Finally, our toolbar components are updated with our `showAttributes()` method.

2. *Class FontDialog*

This class extends `JDialog` and acts as a font properties editor and previewer for our HTML editor application. Several instance variables are declared:

- `boolean m_succeeded`: a flag that will be set to `true` if font changes are accepted.
- `OpenList m_lstFontName`: custom `JList` subclass for selecting the font family name.
- `OpenList m_lstFontSize`: custom `JList` subclass for selecting the font size.
- `MutableAttributeSet m_attributes`: a collection of font attributes used to preserve the user's selection.
- `JCheckBox m_chkBold`: check box to select the bold attribute.
- `JCheckBox m_chkItalic`: check box to select the italic attribute.
- `JCheckBox m_chkUnderline`: check box to select the font underline attribute.
- `JCheckBox m_chkStrikethrough`: check box to select the font strikethrough attribute.
- `JCheckBox m_chkSubscript`: check box to select the font subscript attribute.
- `JCheckBox m_chkSuperscript`: check box to select the font superscript attribute.
- `JComboBox m_cbColor`: combo box to select the font foreground color.
- `JLabel m_preview`: label to preview the selections.

The `FontDialog` constructor first creates a superclass modal dialog titled Font. The constructor creates and initializes all GUI components used in this dialog. A *y*-oriented `BoxLayout` is used to place component groups from top to bottom.

3. Two `OpenList` components are placed at the top to select an available font family name and font size. These components encapsulate a label, text box, and list components which work together. They are similar to editable combo boxes that always keep their drop-down list open. Below the `OpenLists`, a group of six check boxes are placed for selecting bold, italic, underline, strikethrough, subscript, and superscript font attributes. `JComboBox m_cbColor` is placed below this group, and is used to select the font foreground color. Sixty-four `Colors` are added, and an instance of our custom `ColorComboRenderer` class is used as its list cell renderer. (4) `JLabel m_preview` is used to preview the selected font before applying it to the editing text, and is placed below the foreground color combo box.

5. The `m_lstFontName` and `m_lstFontSize` `OpenList` components each receive the same `ListSelectionListener` instance which calls our custom `updatePreview()` method whenever the list selection is changed. Similarly, the check boxes and the foreground color combo box receive an `ActionListener` which does the same thing. This provides dynamic preview of the selected font attributes as soon as any is changed.

BUG ALERT! Underline, strikethrough, subscript, and superscript font properties are not supported by the AWT Font class, so they cannot be shown in the `JLabel` component. This is why the corresponding check box components do not receive an `ActionListener`. As we will see, the strikethrough, subscript, and superscript properties also do not work properly in HTML documents. They are included in this dialog for completeness, in the hopes that they will work properly in a future Swing release.

❶ Two buttons labeled OK and Cancel are placed at the bottom of the dialog. They are placed in a panel managed by a 1x2 `GridLayout`, which is in turn placed in a panel managed by a `FlowLayout`. This is to ensure the equal sizing and central placement of the buttons. Both receive `ActionListeners` which dispose of the dialog. The OK button also sets the `m_succeeded` flag to `true` to indicate that the changes made in the font dialog should be applied.

The dialog window is packed to give it a natural size, and is then centered with respect to the parent frame.

❼ The `setAttributes()` method takes an `AttributeSet` instance as a parameter. It copies this attribute set into a `SimpleAttributeSet` stored as our `m_attributes` instance variable. Appropriate font attributes are extracted using `StyleConstants` methods, and used to assign values to the dialog's controls. Finally the preview label is updated according to these new settings by calling our `updatePreview()` method. Note that the `setAttributes()` method is public and is used for data exchange between this dialog and its owner (in our case `HtmlProcessor`).

❽ The `getAttributes()` method plays an opposite role with respect to `setAttributes()`. It retrieves data from the dialog's controls, packs them into an `AttributeSet` instance using `StyleConstants` methods, and returns this set to the caller.

The `succeeded()` method simply returns the `m_succeeded` flag.

❾ The `updatePreview()` method is called to update the font preview label when a font attribute is changed. It retrieves the selected font attributes (family name, size, bold, and italic properties) and creates a new Font instance to render the label. The selected color is retrieved from the `m_cbColor` combo box and set as the label's foreground.

❿ *Class OpenList*

This component consists of a title label, a text field, and a list in a scroll pane. The user can either select a value from the list, or enter it in the text box manually. `OpenList` extends `JPanel` and maintains the following four instance variables:

- `JLabel m_title`: title label used to identify the purpose of this component.
- `JTextField m_text`: editable text field.
- `JList m_list`: list component.
- `JScrollPane m_scroll`: scroll pane containing the list component.

The `OpenList` constructor assigns a null layout manager because this container manages its child components on its own. The four components are instantiated and simply added to this container.

The `setSelected()` method sets the text field text to that of the given `String`, and selects the corresponding item in the list (which is scrolled to display the newly selected value). The `getSelected()` method retrieves and returns the selected item as a `String`.

Methods `setSelectedInt()/getSelectedInt()` do the same but with `int` values. These methods are implemented to simplify working with a list of `ints`.

The `valueChanged()` and `actionPerformed()` methods provide coordination between the list component and the text field. The `valueChanged()` method is called whenever the list selection changes, and will assign the result of a `toString()` call on the selected item as the text field's text. The `actionPerformed()` method will be called when the user presses Enter while the text field has the current focus. This implementation performs a case-insensitive search through the list items in an effort to find an item which begins with the entered text. If such an item is found, it is selected.

The public `addListSelectionListener()` method adds a `ListSelectionListener` to our list component (which is protected). In this way, external objects can dynamically receive notifications about changes in that list's selection.

⓫ The `getPreferredSize()`, `getMaximumSize()`, and `getMinimumSize()` methods calculate and return a preferred, maximum, and minimum dimension of this container respectively. They assume that the three child components (label, text field, and scroll pane containing the list) will be laid out one under another from top to bottom, receiving an equal width and their preferable heights. The `doLayout()` method actually lays out the components according to this scheme. Note that the insets (resulting from an assigned border, for instance) must always be taken into account (see chapter 4 for more about custom layout management).

⓬ *Class ColorComboRenderer*

This class implements the `ListCellRenderer` interface (discussed in chapters 9 and 10) and is used to represent various `Colors`. Three instance variables are defined:

- `Color m_color`: used for the main background color to represent a `Color`.
- `Color m_focusColor`: used for the thick border color of a selected item.
- `Color m_nonFocusColor`: used for the thick border color of an unselected item.

The `getListCellRendererComponent()` method is called prior to the rendering of each list item (in our `HtmlProcessor` example this list is contained within our foreground colors combo box). The `Color` instance is retrieved and stored in the `m_color` instance variable. This color is used as the renderer's background, while a white matte border is used to surround unselected cells, and a light blue matte border is used to surround a selected cell. The `paintComponent()` method simply sets the background to `m_color` and calls the superclass `paintComponent()` method.

20.6.2 Running the code

Figure 20.14 shows our custom `FontDialog` in action and figure 20.15 shows the foreground color combo box open, displaying our custom list cell renderer. Open an existing HTML file, select a portion of text, and bring up the font dialog. Verify that the initial values correspond to the font attributes of the paragraph of text at the current caret position. Try selecting different font attributes and note that the preview component is updated dynamically. Press the OK

button to apply the selected attributes to the current paragraph. Also verify that clicking Cancel does not apply any changes.

20.7 HTML EDITOR, PART VII: FIND AND REPLACE

Along with font and paragraph dialogs, find and replace functionality has also become a fairly common tool in GUI-based text editing environments. It is safe to assume that most users would be sadly disappointed if this functionality was not included in a new word processor application. In this section we will show how to add this functionality. Traditionally such tools are represented in an Edit menu and can be activated by keyboard accelerators. We will use a dialog containing a single tabbed pane with tabs for finding and replacing a specific region of text. We will also provide several options for searching: match case, search whole words only, and search up or down.

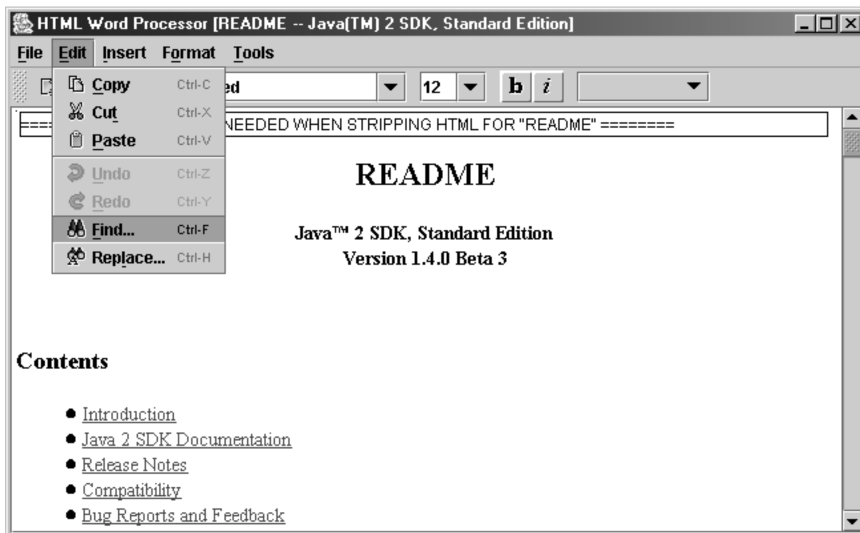

Figure 20.16 `HtmlProcessor` **with complete find and replace functionality; Find... and Replace... menu items shown here**

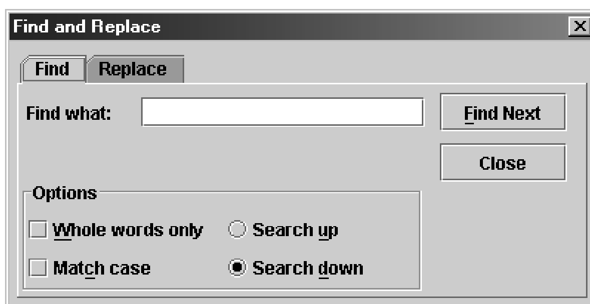

Figure 20.17 Find tab of our custom find and replace dialog

Example 20.7

HtmlProcessor.java

see \Chapter20\7

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

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

import dl.*;

public class HtmlProcessor extends JFrame {

  public static final String APP_NAME = "HTML HTML Editor";

// Unchanged code from example 20.6

  protected FindDialog m_findDialog;

// Unchanged code from example 20.6

  protected JMenuBar createMenuBar() {

    JMenuBar menuBar = new JMenuBar();

    // Unchanged code from example 20.6

    Action findAction = new AbstractAction("Find...",
     new ImageIcon("Find16.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        if (m_findDialog==null)
          m_findDialog = new FindDialog(HtmlProcessor.this, 0);
        else
          m_findDialog.setSelectedIndex(0);
        m_findDialog.show();
      }
    };
    item = mEdit.add(findAction);
    item.setMnemonic('f');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F,
      KeyEvent.Ctrl_MASK));
```

Action to invoke find dialog (1)

```
      Action replaceAction = new AbstractAction("Replace...",
       new ImageIcon("Replace16.gif")) {
        public void actionPerformed(ActionEvent e) {
          if (m_findDialog==null)
            m_findDialog = new FindDialog(HtmlProcessor.this, 1);
          else
            m_findDialog.setSelectedIndex(1);
          m_findDialog.show();
        }
      };
      item = mEdit.add(replaceAction);
      item.setMnemonic('l');
      item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_H,
        KeyEvent.Ctrl_MASK));

      menuBar.add(mEdit);

// Unchanged code from example 20.6

      return menuBar;
    }

    public Document getDocument() {
      return m_doc;
    }

    public JTextPane getTextPane() {
      return m_editor;
    }

    public void setSelection(int xStart, int xFinish, boolean moveUp) {
      if (moveUp) {
        m_editor.setCaretPosition(xFinish);
        m_editor.moveCaretPosition(xStart);
      }
      else
        m_editor.select(xStart, xFinish);
      m_xStart = m_editor.getSelectionStart();
      m_xFinish = m_editor.getSelectionEnd();
    }

    // Unchanged code from example 20.6
}

// Unchanged code from example 20.6

class Utils
{
    // Unchanged code from example 20.6

    public static final char[] WORD_SEPARATORS = {' ', '\t', '\n',
      '\r', '\f', '.', ',', ':', '-', '(', ')', '[', ']', '{',
      '}', '<', '>', '/', '|', '\\', '\'', '\"'};
```

1 Action to invoke find dialog with Replace tab selected

2 Methods to allow easier access by external services

3 Word separator characters

```
  public static boolean isSeparator(char ch) {
    for (int k=0; k<WORD_SEPARATORS.length; k++)
      if (ch == WORD_SEPARATORS[k])
        return true;
    return false;
  }
}
```

3 **Method returns true if a given character is a separator character**

```
// Unchanged code from example 20.6

class FindDialog extends JDialog {
  protected HtmlProcessor m_owner;
  protected JTabbedPane m_tb;
  protected JTextField m_txtFind1;
  protected JTextField m_txtFind2;
  protected Document m_docFind;
  protected Document m_docReplace;
  protected ButtonModel m_modelWord;
  protected ButtonModel m_modelCase;
  protected ButtonModel m_modelUp;
  protected ButtonModel m_modelDown;

  protected int m_searchIndex = -1;
  protected boolean m_searchUp = false;
  protected Stringm_searchData;
```

4 **Dialog with tabbed pane for performing "Find" and "Replace" functionality**

```
  public FindDialog(HtmlProcessor owner, int index) {
    super(owner, "Find and Replace", false);
    m_owner = owner;

    m_tb = new JTabbedPane();

    // "Find" panel
    JPanel p1 = new JPanel(new BorderLayout());

    JPanel pc1 = new JPanel(new BorderLayout());

    JPanel pf = new JPanel();
    pf.setLayout(new DialogLayout2(20, 5));
    pf.setBorder(new EmptyBorder(8, 5, 8, 0));
    pf.add(new JLabel("Find what:"));

    m_txtFind1 = new JTextField();
    m_docFind = m_txtFind1.getDocument();
    pf.add(m_txtFind1);
    pc1.add(pf, BorderLayout.CENTER);

    JPanel po = new JPanel(new GridLayout(2, 2, 8, 2));
    po.setBorder(new TitledBorder(new EtchedBorder(),
      "Options"));

    JCheckBox chkWord = new JCheckBox("Whole words only");
    chkWord.setMnemonic('w');
    m_modelWord = chkWord.getModel();
    po.add(chkWord);

    ButtonGroup bg = new ButtonGroup();
```

5 **Dialog is not modal**

```
JRadioButton rdUp = new JRadioButton("Search up");
rdUp.setMnemonic('u');
m_modelUp = rdUp.getModel();
bg.add(rdUp);
po.add(rdUp);

JCheckBox chkCase = new JCheckBox("Match case");
chkCase.setMnemonic('c');
m_modelCase = chkCase.getModel();
po.add(chkCase);

JRadioButton rdDown = new JRadioButton("Search down", true);
rdDown.setMnemonic('d');
m_modelDown = rdDown.getModel();
bg.add(rdDown);
po.add(rdDown);
pc1.add(po, BorderLayout.SOUTH);

p1.add(pc1, BorderLayout.CENTER);

JPanel p01 = new JPanel(new FlowLayout());
JPanel p = new JPanel(new GridLayout(2, 1, 2, 8));

ActionListener findAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    findNext(false, true);
  }
};
JButton btFind = new JButton("Find Next");
btFind.addActionListener(findAction);
btFind.setMnemonic('f');
p.add(btFind);

ActionListener closeAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    setVisible(false);
  }
};
JButton btClose = new JButton("Close");
btClose.addActionListener(closeAction);
btClose.setDefaultCapable(true);
p.add(btClose);
p01.add(p);
p1.add(p01, BorderLayout.EAST);

m_tb.addTab("Find", p1);

// "Replace" panel
JPanel p2 = new JPanel(new BorderLayout());

JPanel pc2 = new JPanel(new BorderLayout());

JPanel pc = new JPanel();
pc.setLayout(new DialogLayout2(20, 5));
pc.setBorder(new EmptyBorder(8, 5, 8, 0));

pc.add(new JLabel("Find what:"));
```

```
m_txtFind2 = new JTextField();
m_txtFind2.setDocument(m_docFind);
pc.add(m_txtFind2);

pc.add(new JLabel("Replace:"));
JTextField txtReplace = new JTextField();
m_docReplace = txtReplace.getDocument();
pc.add(txtReplace);
pc2.add(pc, BorderLayout.CENTER);

po = new JPanel(new GridLayout(2, 2, 8, 2));
po.setBorder(new TitledBorder(new EtchedBorder(),
   "Options"));

chkWord = new JCheckBox("Whole words only");
chkWord.setMnemonic('w');
chkWord.setModel(m_modelWord);
po.add(chkWord);

bg = new ButtonGroup();
rdUp = new JRadioButton("Search up");
rdUp.setMnemonic('u');
rdUp.setModel(m_modelUp);
bg.add(rdUp);
po.add(rdUp);

chkCase = new JCheckBox("Match case");
chkCase.setMnemonic('c');
chkCase.setModel(m_modelCase);
po.add(chkCase);

rdDown = new JRadioButton("Search down", true);
rdDown.setMnemonic('d');
rdDown.setModel(m_modelDown);
bg.add(rdDown);
po.add(rdDown);
pc2.add(po, BorderLayout.SOUTH);

p2.add(pc2, BorderLayout.CENTER);

JPanel p02 = new JPanel(new FlowLayout());
p = new JPanel(new GridLayout(3, 1, 2, 8));

ActionListener replaceAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    findNext(true, true);
  }
};
JButton btReplace = new JButton("Replace");
btReplace.addActionListener(replaceAction);
btReplace.setMnemonic('r');
p.add(btReplace);

ActionListener replaceAllAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    int counter = 0;
    while (true) {
```

```
            int result = findNext(true, false);
            if (result < 0)
              return;
            else if (result == 0)
           break;
            counter++;
          }
          JOptionPane.showMessageDialog(m_owner,
            counter+" replacement(s) have been done",
            HtmlProcessor.APP_NAME,
            JOptionPane.INFORMATION_MESSAGE);
        }
      };
      JButton btReplaceAll = new JButton("Replace All");
      btReplaceAll.addActionListener(replaceAllAction);
      btReplaceAll.setMnemonic('a');
      p.add(btReplaceAll);

      btClose = new JButton("Close");
      btClose.addActionListener(closeAction);
      btClose.setDefaultCapable(true);
      p.add(btClose);
      p02.add(p);
      p2.add(p02, BorderLayout.EAST);

      // Make button columns the same size
      p01.setPreferredSize(p02.getPreferredSize());

      m_tb.addTab("Replace", p2);

      m_tb.setSelectedIndex(index);

      JPanel pp = new JPanel(new BorderLayout());
      pp.setBorder(new EmptyBorder(5,5,5,5));
      pp.add(m_tb, BorderLayout.CENTER);
      getContentPane().add(pp, BorderLayout.CENTER);

      pack();
      setResizable(false);
      setLocationRelativeTo(owner);

      WindowListener flst = new WindowAdapter() {
        public void windowActivated(WindowEvent e) {
          m_searchIndex = -1;
        }

        public void windowDeactivated(WindowEvent e) {
          m_searchData = null;
        }
      };
      addWindowListener(flst);
    }

    public void setSelectedIndex(int index) {
      m_tb.setSelectedIndex(index);
      setVisible(true);
```

6 **Make tab panels same size so that shared components stay in same position**

```
    m_searchIndex = -1;
  }

  public int findNext(boolean doReplace, boolean showWarnings) {
    JTextPane monitor = m_owner.getTextPane();
    int pos = monitor.getCaretPosition();
    if (m_modelUp.isSelected() != m_searchUp) {
      m_searchUp = m_modelUp.isSelected();
      m_searchIndex = -1;
    }

    if (m_searchIndex == -1) {
      try {
        Document doc = m_owner.getDocument();
        if (m_searchUp)
          m_searchData = doc.getText(0, pos);
        else
          m_searchData = doc.getText(pos, doc.getLength()-pos);
        m_searchIndex = pos;
      }
      catch (BadLocationException ex) {
        warning(ex.toString());
        return -1;
      }
    }

    String key = "";
    try {
      key = m_docFind.getText(0, m_docFind.getLength());
    }
    catch (BadLocationException ex) {}
    if (key.length()==0) {
      warning("Please enter the target to search");
      return -1;
    }
    if (!m_modelCase.isSelected()) {
      m_searchData = m_searchData.toLowerCase();
      key = key.toLowerCase();
    }
    if (m_modelWord.isSelected()) {
      for (int k=0; k<Utils.WORD_SEPARATORS.length; k++) {
        if (key.indexOf(Utils.WORD_SEPARATORS[k]) >= 0) {
          warning("The text target contains an illegal "+
            "character \'"+Utils.WORD_SEPARATORS[k]+"\'");
          return -1;
        }
      }
    }

    String replacement = "";
    if (doReplace) {
      try {
        replacement = m_docReplace.getText(0,
          m_docReplace.getLength());
```

7 Performs actual find/replace operation

8 Get string to search for and optionally convert both search text and target to lowercase

9 Retrieves replacement text

```
    } catch (BadLocationException ex) {}
  }

  int xStart = -1;
  int xFinish = -1;
  while (true)
  {
    if (m_searchUp)
      xStart = m_searchData.lastIndexOf(key, pos-1);
    else
      xStart = m_searchData.indexOf(key, pos-m_searchIndex);
    if (xStart < 0) {
      if (showWarnings)
        warning("Text not found");
      return 0;
    }

    xFinish = xStart+key.length();

    if (m_modelWord.isSelected()) {
      boolean s1 = xStart>0;
      boolean b1 = s1 && !Utils.isSeparator(m_searchData.charAt(
        xStart-1));
      boolean s2 = xFinish<m_searchData.length();
      boolean b2 = s2 && !Utils.isSeparator(m_searchData.charAt(
        xFinish));

      if (b1 || b2)// Not a whole word
      {
        if (m_searchUp && s1)// Can continue up
        {
          pos = xStart;
          continue;
        }
        if (!m_searchUp && s2)// Can continue down
        {
          pos = xFinish+1;
          continue;
        }
        // Found, but not a whole word, and we cannot continue
        if (showWarnings)
          warning("Text not found");
        return 0;
      }
    }
    break;
  }

  if (!m_searchUp) {
    xStart += m_searchIndex;
    xFinish += m_searchIndex;
  }
  if (doReplace) {
    m_owner.setSelection(xStart, xFinish, m_searchUp);
```

9 Retrieves replacement text

10 Searches backward or forward (up or down) for search string

12 Bl and b2 determine whether the found string is in a word boundary

12 Does actual replacement

```
            monitor.replaceSelection(replacement);
            m_owner.setSelection(xStart, xStart+replacement.length(),
              m_searchUp);
            m_searchIndex = -1;                         Does actual replacement 12
          }
          else
            m_owner.setSelection(xStart, xFinish, m_searchUp);
          return 1;
        }

      protected void warning(String message) {
        JOptionPane.showMessageDialog(m_owner,
          message, HtmlProcessor.APP_NAME,
          JOptionPane.INFORMATION_MESSAGE);
      }
    }
```

20.7.1 Understanding the code

Class HtmlProcessor

`HtmlProcessor` declares one new instance variable:

- `FindDialog m_findDialog`: custom dialog for finding and replacing a section of text.

1 Two new menu items titled Find... and Replace..., are added to the Edit menu. These items are activated with keyboard accelerators Ctrl-F and Ctrl-H respectively. When pressed, both items create an instance of `FindDialog` (if `m_findDialog` is `null`) or activate the existing instance, and the dialog is then displayed. The only difference between the two is that the Find... menu item activates the 0-indexed tabbed pane tab, and the Replace... menu item activates the tab at index 1.

2 Three new public methods have been added to this class to make access to our text pane component, and related objects, easier from external sources. The `getDocument()` method retrieves the text pane's current `Document` instance, and the `getTextPane()` method retrieves the text pane itself. The `setSelection()` method selects a portion of text between given start and end positions, and positions the caret at the beginning or end of the selection, depending on the value of the `moveUp` boolean parameter. The coordinates of such a selection are then stored in the `m_xStart` and `m_xFinish` instance variables (recall that these variables always hold the coordinates of the current text selection and are used to restore this selection when our text pane regains the focus).

Class Utils

3 A simple static utility method and an array of chars representing word separator characters is added to this class. The `isSeparator()` method simply checks whether a given character belongs to the static `WORD_SEPARATORS` char array.

4 *Class FindDialog*

This class is a modal `JDialog` subclass encapsulating our find and replace functionality. It contains a tabbed pane with two tabs, Find and Replace. Both tabs contain several common controls that should always be in synch: a check box for whole words only, a check box for

match case, a radio button for search up, a radio button for search down, and a text field for the text we are searching for.

Since the components can exist in only one container, we need to place identical components in each tab. To simplify the task of maintaining consistency in component states, each pair of common components is assigned the same model.

`FindDialog` maintains the following instance variables:

- `HtmlProcessor m_owner`: an explicit reference to our `HtmlProcessor` parent application frame.
- `JTabbedPane m_tb`: the tabbed pane containing the find and replace pages.
- `JTextField m_txtFind1`: used to enter the string to find.
- `JTextField m_txtFind2`: used to enter the string to replace.
- `Document m_docFind`: a shared data model for the Find text fields.
- `Document m_docReplace`: a data model for the Replace text field.
- `ButtonModel m_modelWord`: a shared data model for the Whole words only check boxes.
- `ButtonModel m_modelCase`: a shared data model for the Match case check boxes.
- `ButtonModel m_modelUp`: a shared data model for the Search up radio buttons.
- `ButtonModel m_modelDown`: a shared data model for the Search down radio buttons.
- `int m_searchIndex`: position in the document to start searching from.
- `boolean m_searchUp`: a search direction flag.
- `String m_searchData`: string to search for.

5 The `FindDialog` constructor creates a superclass nonmodal dialog instance titled Find and Replace. The main tabbed pane, `m_tb`, is created, and `JPanel p1` (the main container of the Find tab) receives the `m_txtFind1` text field along with a Find what: label. This text field is used to enter the target string to be searched for. Note that the `Document` instance associated with this textbox is stored in the `m_docFind` instance variable (which will be used to facilitate sharing between another text field).

> **NOTE** In a more sophisticated implementation you might use editable combo boxes with memory in place of text fields, similar to those discussed in the final examples of chapter 9.

Two check boxes titled Whole words only and Match case, and two radio buttons, Search up and Search down, (initially selected) are placed at the bottom of the `p1` panel. These components are surrounded by a titled Options border. Two `JButtons` titled Find Next and Close are placed at the right side of the panel. The first button calls our `findNext()` method when pressed. The second button hides the dialog. Finally the `p1` panel is added to `m_tb` with a tab title of Find.

6 `JPanel p2` (the main container of the Replace tab) receives the `m_txtFind2` text field along with a "Find what:" label. It also receives another pair labeled Replace. An instance of our custom layout manager, `DialogLayout` (discussed in chapter 4), is used to lay out these text fields and corresponding labels without involving any intermediate containers. The same layout is used in the Find panel. We also synchronize the preferred size of the two panels to avoid movement of the synchronized components when a new page is activated.

Note that the m_docFind data object is set as the document for the m_txtFind2 text field. This ensures consistency between the two different Find text fields in the two tabbed panels.

Two check boxes and two radio buttons are placed at the bottom of the panel to control the replacement options. They have identical meaning and representation as the corresponding four controls in the Find panel, and to ensure consistency between them, the data models are shared between each identical component.

Three JButtons titled Replace, Replace All, and Close are placed at the right side of the panel. The Replace button makes a single call to our findNext() method when pressed. The Replace All button is associated with an actionPerformed() method which repeatedly invokes findNext() to perform replacement until it returns -1 to signal an error, or 0 to signal that no more replacements can be made. If an error occurs this method returns, the action-Performed() method simple returns (since an error will be properly reported to the user by the findNext() method). Otherwise the number of replacements made is reported to the user in a JOptionPane message dialog. The Close button hides the dialog. Finally the p2 panel is added to the m_tb tabbed pane with a tab title of Replace.

Since this is a nonmodal dialog, the user can freely switch to the main application frame and return back to the dialog while each remains visible (a typical find-and-replace feature). Once the user leaves the dialog he/she can modify the document's content, or move the caret position. To account for this, we add a WindowListener to the dialog whose windowActivated() method sets m_searchIndex to -1. This way, the next time findNext() is called the search data will be reinitialized, allowing the search to continue as expected, corresponding to the new caret position and document content.

The setSelectedIndex() method activates a page with the given index and makes this dialog visible. This method is intended mostly for use externally by our app when it wants to display this dialog with a specific tab selected.

7 The findNext() method is responsible for performing the actual find and replace operations. It takes two arguments:

- boolean doReplace: if true, find and replace, otherwise just find.
- boolean showWarnings: if true, display a message dialog if target text cannot be found, otherwise do not display a message.

findNext() returns an int result with the following meaning:

- -1: an error has occurred.
- 0: the target text cannot be found.
- 1: a find or find and replace was completed successfully.

The m_searchIndex == -1 condition specifies that the portion of text to be searched through must be refreshed. In this case we store the portion of text from the beginning of the document to the current caret position if we are searching up, or between the current caret position and the end of the document if we are searching down. This text is stored in the m_searchData instance variable. The current caret position is stored in the m_searchIndex variable.

NOTE This solution may not be adequate for large documents. However, a more sophisticated solution would take us too far from the primary goal of this example.

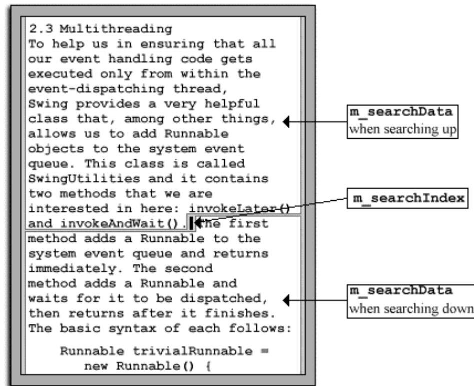

Figure 20.18
Usage of instance variables for searching up and down through document text

8 The text to search for is retrieved from the `m_docFind` shared `Document`. If the case-insensitive option is selected, both the `m_searchData` text and the text to search for are converted into lower case. If the Whole words only option is selected, we check whether the text to search for contains any separator characters defined in our `Util` utilities class.

> **NOTE** If a given `String` is already in all lower or upper case, the `toLowerCase()` (or `toUpperCase()`) method returns the original `String` without creating a new object.

9 After this, if the `doReplace` parameter is `true`, we retrieve the replacement text from our `m_docReplace Document`. At this point we're ready to actually perform a search. We take advantage of existing `String` functionality to accomplish this:

```
if (m_searchUp)
   xStart = m_searchData.lastIndexOf(key, pos-1);
else
   xStart = m_searchData.indexOf(key, pos-m_searchIndex);
```

10 If we are seaching up, we search for the last occurrence of the target string from the current caret position. Otherwise we search for the first occurrence of the target string from the current caret position. If the target string is not found, we cannot continue the search, and a warning is displayed if the `showWarnings` parameter is `true`.

11 This simple scheme is complicated considerably if the Whole words only option is selected. In this case we need to verify whether symbols on the left and on the right of a matching region of text are either word separators defined in our `Utils` class, or the string lies at the end of the data being searched. If these conditions are not satisfied, we attempt to continue searching, unless the end of the search data is reached.

12 In any case, if we locate an acceptable match, we select the located text. If the replace option is selected, we replace this selected region with the specified replacement text and then select the new replacement text. In this latter case we also set `m_searchIndex` to `-1` to force the `m_searchData` variable to be updated. This is necessary for continued searching because the data being searched most likely changes after each replace. The location of the caret also usually changes.

20.7.2 Running the code

Figure 20.16 shows our editor's Find and Replace menu items. Figures 20.17 and 20.18 show our custom `FindDialog`'s Find and Replace tabs respectively. Open an existing HTML file and use the Edit menu, or the appropriate keyboard accelerator, to bring up the Find and Replace dialog with the Find tab selected. Enter some text to search for, select some search options, and press the Find Next button. If your target text is found, the matching region will be highlighted in the base document. Click this button again to find subsequent entries (if any). Verify that the Whole words only and Match case options function as discussed earlier. Change focus to the main application window and modify the document and/or change the caret position. Return to the Find and Replace dialog and note that the search continues as expected.

Select the Replace tab and verify that the state of all search options, including the search target string, are preserved from the Find tab (and vice versa when switching between tabs). Enter a replacement string and verify that the Replace and Replace All buttons work as expected.

20.8 HTML EDITOR, PART IX: SPELL CHECKER (USING JDBC AND SQL)

Most modern word processor applications offer tools and utilities which help the user in finding grammatical and spelling mistakes in a document. In this section we will add spell-checking to our HTML editor application. To do this we will need to perform some of our own multithreading, and use JDBC to connnect to a database containing a dictionary of words. We will use a simple database with one table, Data, which has the following structure:

Name	Type	Description
word	String	A single English word
sound	String	A 4-letter SOUNDEX code

An example of this database, populated with words from several Shakespeare comedies and tragedies, is provided in this example's directory: Shakespeare.mdb. (This database must be a registered database in your database manager prior to using it. This is not a JDBC tutorial, so we'll skip the details.)

NOTE The custom SOUNDEX algorithm used in this example hashes words for efficiency by using a simple model which approximates the sound of the word when spoken. Each word is reduced to a four character string, the first character being an upper case letter and the remaining three being digits. (This algorithm was created and patented by Robert C. Russell in 1918.)

Example 20.8

HtmlProcessor.java

see \Chapter20\8

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.*;

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

import dl.*;

public class HtmlProcessor extends JFrame {

  public static final String APP_NAME = "HTML HTML Editor";

  // Unchanged code from example 20.7

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();
```

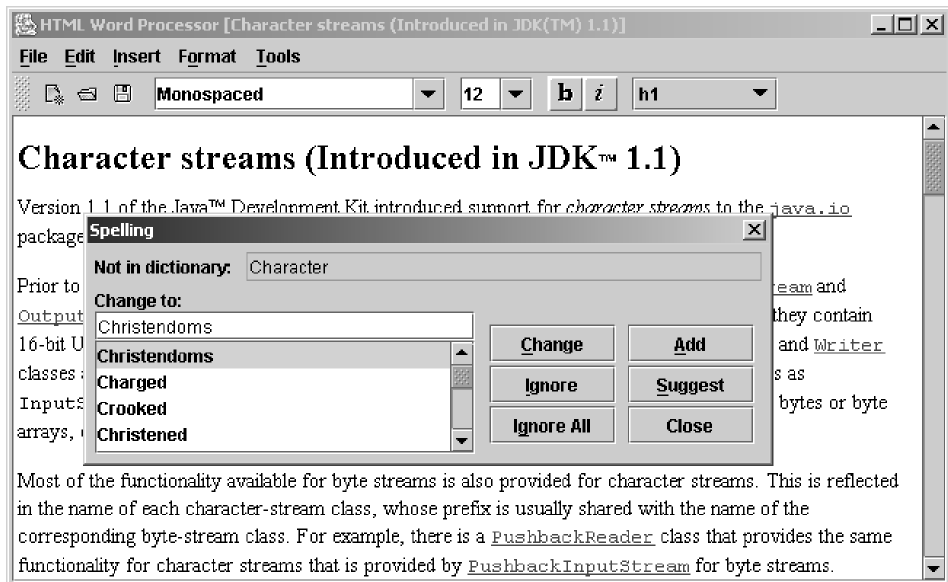

Figure 20.19 HtmlProcessor's `SpellChecker.SpellingDialog` dialog

```
    // Unchanged code from example 20.7

    Action spellAction = new AbstractAction("Spelling...",
     new ImageIcon("SpellCheck16.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        SpellChecker checker = new SpellChecker(HtmlProcessor.this);
        HtmlProcessor.this.setCursor(Cursor.getPredefinedCursor(
          Cursor.WAIT_CURSOR));
        checker.start();
      }
    };
    item =mTools.add(spellAction);
    item.setMnemonic('s');
    item.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_F7, 0));

    menuBar.add(mTools);

    // Unchanged code from example 20.7

    return menuBar;
  }

  // Unchanged code from example 20.7
}

// Unchanged code from example 20.7
class Utils
{
  // Unchanged code from example 20.7

  public static String soundex(String word) {
    char[] result = new char[4];
    result[0] = word.charAt(0);
    result[1] = result[2] = result[3] = '0';
    int index = 1;

    char codeLast = '*';
    for (int k=1; k<word.length(); k++) {
      char ch = word.charAt(k);
      char code = ' ';
      switch (ch) {
        case 'b': case 'f': case 'p': case 'v':
          code = '1';
          break;
        case 'c': case 'g': case 'j': case 'k':
        case 'q': case 's': case 'x': case 'z':
          code = '2';
          break;
        case 'd': case 't':
          code = '3';
          break;
        case 'l':
          code = '4';
          break;
```

1 New menu item initiates spell checking

2 Calculate the SOUNDEX code of a given word

```java
            case 'm': case 'n':
               code = '5';
               break;
            case 'r':
               code = '6';
               break;
            default:
               code = '*';
               break;
         }
         if (code == codeLast)
            code = '*';
         codeLast = code;
         if (code != '*') {
            result[index] = code;
            index++;
            if (index > 3)
               break;
         }
      }
      return new String(result);
   }

   public static boolean hasDigits(String word) {
      for (int k=1; k<word.length(); k++) {
         char ch = word.charAt(k);
         if (Character.isDigit(ch))
            return true;
      }
      return false;
   }

   public static String titleCase(String source) {
      return Character.toUpperCase(source.charAt(0)) +
         source.substring(1);
   }
}

// Unchanged code from example 20.7

class OpenList extends JPanel
   implements ListSelectionListener, ActionListener
{
   protected JLabel m_title;
   protected JTextField m_text;
   protected JList m_list;
   protected JScrollPane m_scroll;

   public OpenList(String[] data, String title) {
      setLayout(null);
      m_title = new JLabel(title, JLabel.LEFT);
      add(m_title);
      m_text = new JTextField();
      m_text.addActionListener(this);
```

3 **Returns true if given word contains digits**

3 **Converts first character of given String to upper case**

4 **OpenList component modified to populate list with ResultSet data**

```
        add(m_text);
        m_list = new JList(data);
        m_list.setVisibleRowCount(4);
        m_list.addListSelectionListener(this);
        m_list.setFont(m_text.getFont());
        m_scroll = new JScrollPane(m_list);
        add(m_scroll);
    }

    public OpenList(String title, int numCols) {
        setLayout(null);
        m_title = new JLabel(title, JLabel.LEFT);
        add(m_title);
        m_text = new JTextField(numCols);
        m_text.addActionListener(this);
        add(m_text);
        m_list = new JList();
        m_list.setVisibleRowCount(4);
        m_list.addListSelectionListener(this);
        m_scroll = new JScrollPane(m_list);
        add(m_scroll);
    }

    public void appendResultSet(ResultSet results, int index,
     boolean toTitleCase)
    {
        m_text.setText("");
        DefaultListModel model = new DefaultListModel();
        try {
            while (results.next()) {
                String str = results.getString(index);
                if (toTitleCase)
                    str = Utils.titleCase(str);
                model.addElement(str);
            }
        }
        catch (SQLException ex) {
            System.err.println("appendResultSet: "+ex.toString());
        }
        m_list.setModel(model);
        if (model.getSize() > 0)
            m_list.setSelectedIndex(0);
    }
    // Unchanged code from example 20.7
}

// Unchanged code from example 20.7

class SpellChecker extends Thread {
    protected static String SELECT_QUERY =
        "SELECT Data.word FROM Data WHERE Data.word = ";
    protected static String SOUNDEX_QUERY =
        "SELECT Data.word FROM Data WHERE Data.soundex = ";
```

5 Custom thread that performs actual spell checking from current caret position down

```
protected HtmlProcessor m_owner;
protected Connection m_conn;
protected DocumentTokenizer m_tokenizer;
protected Hashtablem_ignoreAll;
protected SpellingDialog m_dlg;

public SpellChecker(HtmlProcessor owner) {
  m_owner = owner;
}

public void run() {
  JTextPane monitor = m_owner.getTextPane();
  m_owner.setEnabled(false);
  monitor.setEnabled(false);

  m_dlg = new SpellingDialog(m_owner);
  m_ignoreAll = new Hashtable();

  try {
    // Load the JDBC-ODBC bridge driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    m_conn = DriverManager.getConnection(
      "jdbc:odbc:Shakespeare", "admin", "");
    Statement selStmt = m_conn.createStatement();

    Document doc = m_owner.getDocument();
    int pos = monitor.getCaretPosition();
    m_tokenizer = new DocumentTokenizer(doc, pos);
      String word, wordLowCase;

    while (m_tokenizer.hasMoreTokens()) {
      word = m_tokenizer.nextToken();
      if (word.equals(word.toUpperCase()))
        continue;
      if (word.length()<=1)
        continue;
      if (Utils.hasDigits(word))
        continue;
      wordLowCase = word.toLowerCase();
      if (m_ignoreAll.get(wordLowCase) != null)
        continue;

      ResultSet results = selStmt.executeQuery(
        SELECT_QUERY+"'"+wordLowCase+"'");
      if (results.next())
        continue;

      results = selStmt.executeQuery(SOUNDEX_QUERY+
        "'"+Utils.soundex(wordLowCase)+"'");
      m_owner.setSelection(m_tokenizer.getStartPos(),
        m_tokenizer.getEndPos(), false);
      if (!m_dlg.suggest(word, results))
        break;
    }

    m_conn.close();
```

6 Before doing work, disables components

7 Looks at each word in the document

7 If word is found in the database then it is spelled correctly

8 If it is an unknown word try to find words that sound like it to suggest as replacements

```
            System.gc();
            monitor.setCaretPosition(pos);
        }
        catch (Exception ex) {
            ex.printStackTrace();
            System.err.println("SpellChecker error: "+ex.toString());
        }

        monitor.setEnabled(true);
        m_owner.setEnabled(true);
        m_owner.setCursor(Cursor.getPredefinedCursor(
            Cursor.DEFAULT_CURSOR));
    }

    protected void replaceSelection(String replacement) {
        int xStart = m_tokenizer.getStartPos();
        int xFinish = m_tokenizer.getEndPos();
        m_owner.setSelection(xStart, xFinish, false);
        m_owner.getTextPane().replaceSelection(replacement);
        xFinish = xStart+replacement.length();
        m_owner.setSelection(xStart, xFinish, false);
        m_tokenizer.setPosition(xFinish);
    }

    protected void addToDB(String word) {
        String sdx = Utils.soundex(word);
        try {
            Statement stmt = m_conn.createStatement();
            stmt.executeUpdate(
                "INSERT INTO DATA (Word, Soundex) VALUES ('"+
                word+"', '"+sdx+"')");
        }
        catch (Exception ex) {
            ex.printStackTrace();
            System.err.println("SpellChecker error: "+ex.toString());
        }
    }

    class SpellingDialog extends JDialog {
        protected JTextField m_txtNotFound;
        protected OpenList m_suggestions;

        protected String m_word;
        protected booleanm_continue;

        public SpellingDialog(HtmlProcessor owner) {
            super(owner, "Spelling", true);

            JPanel p = new JPanel();
            p.setBorder(new EmptyBorder(5, 5, 5, 5));
            p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
            p.add(new JLabel("Not in dictionary:"));
            p.add(Box.createHorizontalStrut(10));
            m_txtNotFound = new JTextField();
            m_txtNotFound.setEditable(false);
```

9 If the word was misspelled and the user accepted a replacement, this does the replacement

9 Adds a word to the “known” database

10 Dialog that prompts the user for an action on a misspelled word

11 Text field containing misspelled word

```
p.add(m_txtNotFound);
getContentPane().add(p, BorderLayout.NORTH);

m_suggestions = new OpenList("Change to:", 12);
m_suggestions.setBorder(new EmptyBorder(0, 5, 5, 5));
getContentPane().add(m_suggestions, BorderLayout.CENTER);

JPanel p1 = new JPanel();
p1.setBorder(new EmptyBorder(20, 0, 5, 5));
p1.setLayout(new FlowLayout());
p = new JPanel(new GridLayout(3, 2, 8, 2));

JButton bt = new JButton("Change");
ActionListener lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    replaceSelection(m_suggestions.getSelected());
    m_continue = true;
    setVisible(false);
  }
};
bt.addActionListener(lst);
bt.setMnemonic('c');
p.add(bt);

bt = new JButton("Add");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    addToDB(m_word.toLowerCase());
    m_continue = true;
    setVisible(false);
  }
};
bt.addActionListener(lst);
bt.setMnemonic('a');
p.add(bt);

bt = new JButton("Ignore");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_continue = true;
    setVisible(false);
  }
};
bt.addActionListener(lst);
bt.setMnemonic('i');
p.add(bt);

bt = new JButton("Suggest");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    try {
      m_word = m_suggestions.getSelected();
      Statement selStmt = m_conn.createStatement();
      ResultSet results = selStmt.executeQuery(
        SELECT_QUERY+"'"+m_word.toLowerCase()+"'");
```

11 List containing replacement suggestions

12 Replaces misspelled word with selected suggestion

13 Adds misspelled word to database

14 Adds words to suggestions list that "sound like" selected suggestion

```
            boolean toTitleCase = Character.isUpperCase(
              m_word.charAt(0));
            m_suggestions.appendResultSet(results, 1,
              toTitleCase);
          }
          catch (Exception ex) {
            ex.printStackTrace();
            System.err.println("SpellChecker error: "+
              ex.toString());
          }
        }
      };
      bt.addActionListener(lst);
      bt.setMnemonic('s');
      p.add(bt);

      bt = new JButton("Ignore All");
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          m_ignoreAll.put(m_word.toLowerCase(), m_word);
          m_continue = true;
          setVisible(false);
        }
      };
      bt.addActionListener(lst);
      bt.setMnemonic('g');
      p.add(bt);

      bt = new JButton("Close");
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          m_continue = false;
          setVisible(false);
        }
      };
      bt.addActionListener(lst);
      bt.setDefaultCapable(true);
      p.add(bt);
      p1.add(p);
      getContentPane().add(p1, BorderLayout.EAST);

      pack();
      setResizable(false);
      setLocationRelativeTo(owner);
    }

    public boolean suggest(String word, ResultSet results) {
      m_continue = false;
      m_word = word;
      m_txtNotFound.setText(word);
      boolean toTitleCase = Character.isUpperCase(
        word.charAt(0));
      m_suggestions.appendResultSet(results, 1, toTitleCase);
      show();
```

14 Adds words to suggestions list that "sound like" selected suggestion

15 Skips word, and will skip all ocurrences of word in this document

16 Called during spell checking to populate the dialog with a misspelled word and its replacement suggestion

```
            return m_continue;
        }
    }
}

class DocumentTokenizer {
    protected Document m_doc;
    protected Segmentm_seg;
    protected int m_startPos;
    protected int m_endPos;
    protected int m_currentPos;

    public DocumentTokenizer(Document doc, int offset) {
        m_doc = doc;
        m_seg = new Segment();
        setPosition(offset);
    }

    public boolean hasMoreTokens() {
        return (m_currentPos < m_doc.getLength());
    }

    public String nextToken() {
        StringBuffer s = new StringBuffer();
        try {
            // Trim leading separators
            while (hasMoreTokens()) {
                m_doc.getText(m_currentPos, 1, m_seg);
                char ch = m_seg.array[m_seg.offset];
                if (!Utils.isSeparator(ch)) {
                    m_startPos = m_currentPos;
                    break;
                }
                m_currentPos++;
            }

            // Append characters
            while (hasMoreTokens()) {
                m_doc.getText(m_currentPos, 1, m_seg);
                char ch = m_seg.array[m_seg.offset];
                if (Utils.isSeparator(ch)) {
                    m_endPos = m_currentPos;
                    break;
                }
                s.append(ch);
                m_currentPos++;
            }
        }
        catch (BadLocationException ex) {
            System.err.println("nextToken: "+ex.toString());
            m_currentPos = m_doc.getLength();
        }
        return s.toString();
    }
```

7 **Used like StreamTokenizer, but keeps track of the character position from each token**

18 **Returns the next token, starting at the position in the document**

```
    public int getStartPos() { return m_startPos; }

    public int getEndPos() { return m_endPos; }

    public void setPosition(int pos) {
      m_startPos = pos;
      m_endPos = pos;
      m_currentPos = pos;
    }
}
```

20.8.1 Understanding the code

Class HtmlProcessor

1. This class now imports the `java.sql` package to make use of JDBC functionality. The `createMenuBar()` method now creates a new menu item in the Tools menu titled Spelling... . This menu item can also be invoked with keyboard accelerator F7. When selected it creates and starts a `SpellChecker` thread, passing a reference to the main application frame as a parameter.

Class Utils

2. Three new static methods are added to this class. The `soundex()` method calculates and returns the SOUNDEX code of the given word. To calculate that code we use the first character of the given word and add a three-digit code that represents the first three remaining consonants. The conversion is made according to the following table:

Code	Letters
1	B,P,F,V
2	C,S,G,J,K,Q,X,Z
3	D,T
4	L
5	M,N
6	R
*	(all others)

3. The `hasDigits()` method returns `true` if a given string contains digits, and the `titleCase()` method converts the first character of a given string to upper case.

4. *Class OpenList*

This custom component receives new functionality for use in our new spell checker dialog. First, we add a new constructor which assigns a given number of columns to the text field, and does not initialize the list component.

Second, we add the `appendResultSet()` method which populates the list component with the data supplied in the given `ResultSet` instance at the given position. If the third parameter is set to `true`, this tells the method to convert all string data to the 'title case' (which means that the first letter is in upper case, and the rest of the string is unchanged). This is accomplished through use of the `Utils.titleCase()` method.

❺ *Class SpellChecker*

This class extends `Thread` to perform spell checking of the current document from the current caret position moving downward. Two class variables are declared:

- `String SELECT_QUERY`: SQL query text used to select a word equal to a given string.
- `String SOUNDEX_QUERY`: SQL query text used to select a word matching a given SOUNDEX value.

Five instance variables are declared:

- `HtmlProcessor m_owner`: a reference to the main application frame.
- `Connection m_conn`: JDBC connection to a database.
- `DocumentTokenizer m_tokenizer`: a custom object used to retrieve each word in a document.
- `Hashtable m_ignoreAll`: a collection of words to ignore in a search, added to with the Ignore All button.
- `SpellingDialog m_dlg`: our custom dialog used for processing spelling mistakes.

The `SpellChecker` constructor takes a reference to the application's frame as a parameter and stores it in the `m_owner` instance variable.

❻ The `run()` method is responsible for the most significant activity of this thread. To prevent the user from modifying the document during spell checking we first disable the main application frame and our text pane contained within it.

> **NOTE** Unlike AWT, Swing containers do not disable their child components when they themselves are disabled. It is not clear whether this is a bug, an intended feature, or an oversight.

Then we create a new `SpellingDialog` instance to provide the user interface, and instantiate the `m_ignoreAll` collection. In a try/catch block we process all JDBC interactions to allow proper handling of any potential errors. This code creates a JDBC connection to our Shakespeare database, retrieves the current caret position, and creates an instance of `DocumentTokenizer` to parse the document from the current caret position. ❼ In a `while` loop we perform spell checking on each word fetched until there are no more tokens. Words in all upper case, containing only one letter, or containing digits, are skipped (this behavior can easily be customized). Then we convert the word under examination to lower case and search for it in the `m_ignoreAll` collection. If it is not found, we try to find it in the database. ❽ If the SQL query does not return any results, we try to locate a similar word in the database with the same SOUNDEX value to suggest to the user in the dialog. The word in question is then selected in our text pane to show the user which word is currently under examination. Finally we call our `SpellingDialog`'s `suggest()` method to request that the user make a decision about what to do with this word. If the `suggest()` method returns `false`, the user has chosen to terminate the spell checking process, so we exit the loop. Once outside the loop we close the JDBC connection, restore the original caret position, explicitly call the garbage collector, and reenable the main application frame and our text pane editor contained within it.

❾ The following two methods are invoked by the `SpellingDialog` instance associated with this `SpellChecker`:

- `replaceSelection()` is used to replace the most recently parsed word with the given replacement string.
- `addToDB()` adds a given word and it's SOUNDEX value to the database by executing an insert query.

10 *Class SpellChecker.SpellingDialog*

This inner class represents a dialog which prompts the user to verify or correct a certain word if it is not found in the database. The user can select one of several actions in response: ignore the given word, ignore all occurrences of that word in the document, replace that word with another word, add this word to the database and consider it correct in any future matches, or cancel the spell check. Four instance variables are declared:

- `JTextField m_txtNotFound`: used to display the word under investigation.
- `OpenList m_suggestions`: editable list component to select or enter a replacement word.
- `String m_word`: the word under investigation.
- `boolean m_continue`: a flag indicating that spell checking should continue.

11 The `SpellingDialog` constructor places the `m_txtNotFound` component and corresponding label at the top of the dialog window. The `m_suggestions OpenList` is placed in the center, and six buttons are grouped to the right.

12 The Change button replaces the word under investigation with the word currently selected in the list or entered by the user. Then it stores `true` in the `m_continue` flag and hides the dialog window. This terminates the modal state of the dialog and makes the `show()` method return, which in turn allows the program's execution to continue (recall that modal dialogs block the calling thread until they are dismissed).

13 The Add button adds the word in question to the spelling database. This word will then be considered correct in future queries. In this way we allow the spell checker to "learn" new words (i.e., add them to the dictionary).

14 The Suggest button populates the `m_suggestions` list with all SOUNDEX matches to the word under investigation. This button is intended for use in situations where the user is not satisfied with the initial suggestions.

The Ignore button simply skips the current word and continues spell checking the remaining text.

15 The Ignore All button does the same as the Ignore, but also stores the word in question in the collection of words to ignore, so the next time the spell checker finds this word it will be deemed correct. The difference between Ignore All and Add is that ignored words will only be ignored during a single spell check, whereas words added to the database will persist as long as the database data does.

The Close button stores `false` in the `m_continue` flag and hides the dialog window. This results in the termination of the spell checking process (see the `suggest()` method).

16 The `suggest()` method is used to display this `SpellingDialog` each time a questionable word is located during the spell check. It takes a `String` and a `ResultSet` containing suggested substitutions as parameters. It sets the text of the `m_txtNotFound` component to the

String passed in, and calls appendResultSet() on the OpenList to display an array of suggested corrections. Note that the first character of these suggestions will be converted to upper case if the word in question starts with a capital letter. Finally, the show() method displays this dialog in the modal state. As soon as this state is terminated by one of the push buttons, or by directly closing the dialog, the suggest() method returns the m_continue flag. If this flag is set to false, this indicates that the calling program should terminate the spell checking cycle.

17 *Class DocumentTokenizer*

This helper class was built to parse the current text pane document. Unfortunately we cannot use the standard StreamTokenizer class for this purpose, because it provides no way of querying the position of a token within the document (we need this information to allow word replacement). Several instance variables are declared:

- Document m_doc: a reference to the document to be parsed.
- Segment m_seg: used for quick delivery of characters from the document being parsed.
- int m_startPos: the start position of the current word from the beginning of the document.
- int m_endPos: the end position of the current word from the beginning of the document.
- int m_currentPos: the current position of the parser from the beginning of the document.

The DocumentTokenizer constructor takes a reference to the document to be parsed and the offset to start at as parameters. It initializes the instance variables described previously.

The hasMoreTokens() method returns true if the current parsing position lies within the document.

18 The nextToken() method extracts the next token (a group of characters separated by one or more characters defined in the WORD_SEPARATORS array from our Utils class) and returns it as a String. The positions of the beginning and the end of the token are stored in the m_startPos and m_endPos instance variables respectively. To access a portion of document text with the least possible overhead we use the Document.getText() method which takes three parameters: offset from the beginning of the document, length of the text fragment, and a reference to an instance of the Segment class. (Recall from chapter 19 that the Segment class provides an efficient means of directly accessing an array of document characters.)

We look at each character in turn, passing over separator characters until the first nonseparator character is reached. This position is marked as the beginning of a new word. Then a StringBuffer is used to accumulate characters until a separator character, or the end of document, is reached. The resulting characters are returned as a String.

NOTE This variant of the getText() method gives us direct access to the characters contained in the document through a Segment instance. These characters should not be modified.

20.8.2 Running the code

Figure 20.19 shows our editor application with the spell checker dialog open. Open an existing HTML file and try running a complete spell check. Try adding some words to the dictionary and use the Ignore All button to avoid questioning a word again during that spell check.

Try using the Suggest button to query the database for more suggestions based on our SOUNDEX algorithm. Click Change to accept a suggestion or a change typed into the text field. Click Ignore to ignore the current word being questioned.

NOTE The Shakespeare vocabulary database supplied for this example is neither complete nor contemporary. It does not include such words as “software” or “Internet.” However, you can easily add them, when encountered during a spell check, by clicking the Add button.