1

The HTML Editor Kit

As Swing matures, the HTML support in the `JEditorPane` class improves. The support for reading, writing, and displaying HTML through the `JEditorPane` class is provided by the `HTMLEditorKit` and associated classes. This class is an extension of the generic `EditorKit` class and is devoted (not surprisingly) to HTML files. There are three basic areas of an editor kit: its parser, its file writer, and its association with a `ViewFactory`. While we have already looked at these pieces before (in Chapter 23 of *Java Swing*, 2nd edition), this chapter focuses on their implementation in `HTMLEditorKit`. Along the way, we'll see how to extend each of these pieces to do custom work within the context of an HTML document.

Before we begin, we should explain that this is the first of two supplemental chapters to *Java Swing*, 2nd edition. This chapter provides detailed information on `HTMLEditorKit`, augmenting the treatment in Chapter 23 of that book. The second supplemental chapter describes HTML I/O. Both chapters are available from our web site, *http://www.oreilly.com/catalog/jswing2* (OK, you probably know that already since you are reading this chapter!).

HTML and the JEditorPane

Before we get down to business on the API, here's a simple example of the `HTMLEditorKit`'s power. In a very small amount of code, we can create a web browser. This mini-browser contains nothing more than two labels, a text field, and a `JEditorPane` controlled by the `HTMLEditorKit`. With it, we can enter a URL, view that page, and even follow hyperlinks.

We'll be building on this simple browser throughout the chapter, but here's something to play with right away. We'll cover all the parts in detail, but if you can't wait to get started with your own browser, pay special attention to the *SimpleLinkListener1.java* file. That's

where most of the real work in this example occurs.* If all you want to do is display HTML text in your application, this is the example to look at.

Figure 1-1. MiniBrowser1 on a Mac OS X system displaying www.oreilly.com

And here's the code for this mini-browser. The first file, *MiniBrowser1.java*, sets up the basic Swing components you can see in Figure 1-1, namely a text field for entering new URLs, a `JEditorPane` for displaying the current pag,e and a label at the bottom of the page for the status bar. (The status bar reflects the "go to" URL when you mouse over a hyperlink.) The second file, *SimpleLinkListener1.java*, handles the hyperlink events.

```
/*
 * MiniBrowser1.java
 * A test bed for the JEditorPane and a custom editor kit.
 * This extremely simple browser has a text field for typing in
 * new urls, a JEditorPane to display the HTML page, and a status
 * bar to display the contents of hyperlinks the mouse passes over.
 */

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

* The original version of this browser appeared in Marc's article, "Patch the Swing HTMLEditorKit," *JavaWorld*, January 1999 (*http://www.javaworld.com/javaworld/jw-01-1999/jw-01-swing.html*). Our derivative code is used with permission. Thanks also to Mats Forslöf (at Marcwell, in Sweden) for his improvements to the original version. You should feel free to check out the article, but be aware that the patch mentioned fixes a pre-1.3 bug.

```
import java.awt.*;
import java.io.File;

public class MiniBrowser1 extends JFrame {

  private JEditorPane jep;

  public MiniBrowser1(String startingUrl) {
    // Ok, first just get a screen up and visible, with an appropriate
    // handler in place for the kill window command
    super("MiniBrowser");
    setSize(400,300);
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    // Now set up our basic screen components, the editor pane, the
    // text field for URLs, and the label for status and link information.
    JPanel urlPanel = new JPanel();
    urlPanel.setLayout(new BorderLayout());
    JTextField urlField = new JTextField(startingUrl);
    urlPanel.add(new JLabel("Site: "), BorderLayout.WEST);
    urlPanel.add(urlField, BorderLayout.CENTER);
    final JLabel statusBar = new JLabel(" ");

    // Here's the editor pane configuration.  It's important to make
    // the "setEditable(false)" call. Otherwise, our hyperlinks won't
    // work.  (If the text is editable, then clicking on a hyperlink
    // simply means that you want to change the text...not follow the
    // link.)
    jep = new JEditorPane();
    jep.setEditable(false);

    try {
      jep.setPage(startingUrl);
    }
    catch(Exception e) {
      statusBar.setText("Could not open starting page.  Using a blank.");
    }
    JScrollPane jsp = new JScrollPane(jep);

    // and get the GUI components onto our content pane
    getContentPane().add(jsp, BorderLayout.CENTER);
    getContentPane().add(urlPanel, BorderLayout.NORTH);
    getContentPane().add(statusBar, BorderLayout.SOUTH);

    // and last but not least, hook up our event handlers
    urlField.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        try {
          jep.setPage(ae.getActionCommand());
        }
        catch(Exception e) {
          statusBar.setText("Error: " + e.getMessage());
        }
      }
```

```
    });
    jep.addHyperlinkListener(new SimpleLinkListener1(jep, urlField,
                                                     statusBar));
  }

  public static void main(String args[]) {
    String url = "";
    if (args.length == 1) {
      url = args[0];
      if (!(url.startsWith("http:") || url.startsWith("file:"))) {
        // If it's not a fully qualified url, assume it's a file
        if (url.startsWith("/")) {
          // Absolute path, so just prepend "file:"
          url = "file:" + url;
        }
        else {
          try {
            // assume it's relative to the starting point...
            File f = new File(url);
            url = f.toURL().toString();
          }
          catch (Exception e) {
            url = "http://www.oreilly.com";
          }
        }
      }
    }
    else {
      url = "http://www.oreilly.com";
    }
    new MiniBrowser1(url).setVisible(true);
  }
}
```

And here's *SimpleLinkListener1.java*. Notice that the "entered" and "exited" event handlers do not mess with the mouse cursor. As of SDK 1.3, the mouse cursor is automatically updated as you enter and exit hyperlinks. All we do is update the status bar. Don't worry too much about the hyperlink handling; we'll tackle that topic in much more detail later in this chapter. Like we said, this is a good starting place if you can't wait. Hopefully, you have seen enough Java event-handling code to follow the example without reading the details yet.

```
/*
 * SimpleLinkListener1.java
 * A hyperlink listener for use with JEditorPane.  This
 * listener changes the cursor over hotspots based on enter/exit
 * events and also load a new page when a valid hyperlink is clicked.
 */

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

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

public class SimpleLinkListener1 implements HyperlinkListener {

  private JEditorPane pane;       // The pane we're using to display HTML

  private JTextField  urlField;   // An optional text field for showing
                                  // the current URL being displayed

  private JLabel statusBar;       // An optional label for showing where
                                  // a link would take you

  public SimpleLinkListener1(JEditorPane jep, JTextField jtf, JLabel jl) {
    pane = jep;
    urlField = jtf;
    statusBar = jl;
  }

  public SimpleLinkListener1(JEditorPane jep) {
    this(jep, null, null);
  }

  public void hyperlinkUpdate(HyperlinkEvent he) {
    HyperlinkEvent.EventType type = he.getEventType();
    if (type == HyperlinkEvent.EventType.ENTERED) {
      // Enter event.  Fill in the status bar
      if (statusBar != null) {
        statusBar.setText(he.getURL().toString());
      }
    }
    else if (type == HyperlinkEvent.EventType.EXITED) {
      // Exit event.  Clear the status bar
      if (statusBar != null) {
        statusBar.setText(" "); // must be a space or it disappears
      }
    }
    else {
      // Jump event.  Get the url, and if it's not null, switch to that
      // page in the main editor pane and update the "site url" label.
      if (he instanceof HTMLFrameHyperlinkEvent) {
        // ahh, frame event, handle this separately
        HTMLFrameHyperlinkEvent  evt = (HTMLFrameHyperlinkEvent)he;
        HTMLDocument doc = (HTMLDocument)pane.getDocument();
        doc.processHTMLFrameHyperlinkEvent(evt);
      } else {
        try {
          pane.setPage(he.getURL());
          if (urlField != null) {
            urlField.setText(he.getURL().toString());
          }
        }
```

```
        catch (FileNotFoundException fnfe) {
          pane.setText("Could not open file: <tt>" + he.getURL() +
                       "</tt>.<hr>");
        }
        catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

The HTMLEditorKit Class

The API for the `HTMLEditorKit` is more or less what you might expect from an editor kit if you made it through all the chapters on the Swing text package (in our book, *Java Swing*, Chapters 19 through 23). While Chapter 23 covers this editor kit, we'll take a deeper look here at the API and then go through an example of how to extend the kit with your own functionality.

The full class diagram for the various classes that make up the `HTMLEditorkit` is shown in Figure 1-3. As you might recall, the editor kit is not itself a Swing component. It helps the `JEditorPane` do its work. While we're not here to discuss the Rich Text Format (RTF) support available, the `RTFEditorKit` has a similar (thought somewhat simpler) layout.

Inner Classes

public static class HTMLEditorKit.HTMLFactory
: The view factory implementation for HTML.

public static class HTMLEditorKit.HTMLTextAction
: An action to easily insert HTML into an existing document.

public static class HTMLEditorKit.InsertHTMLTextAction
: An extension of `HTMLTextAction` that allows you to insert arbitrary HTML content.

public static class HTMLEditorKit.LinkController
: The event listener that translates mouse events into the (often) more desirable `HyperlinkEvent`s.

public static class HTMLEditorKit.Parser
: A parser to read an input stream of HTML.

public static class HTMLEditorKit.ParserCallback
: An implementation of a callback for use while loading an HTML document.

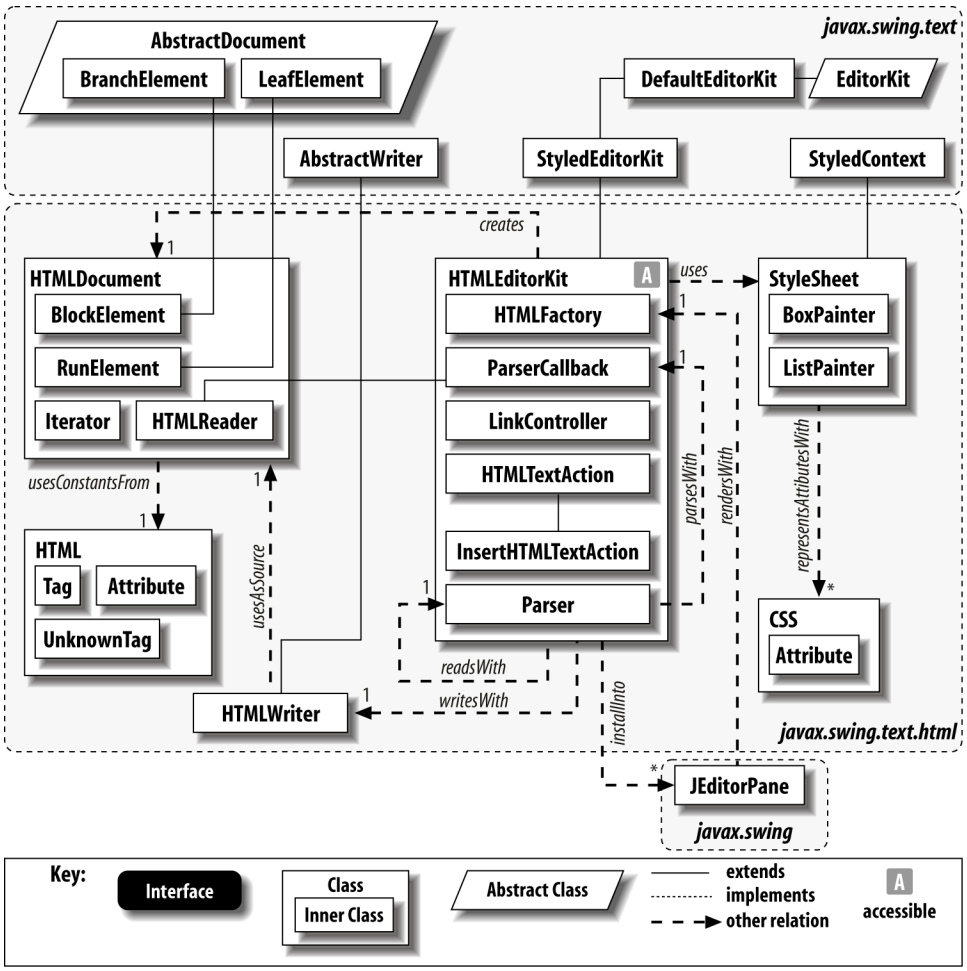

Figure 1-2. HTMLEditorKit class diagram.

Constants

The `HTMLEditorKit` defines several constants for use with actions and editing. With the exception of `DEFAULT_CSS`, the constants listed in Table 1-1 represent the names of actions for altering the appearance of content in an HTML document.

Table 1-1: `HTMLEditorKit` *constants*

Constant	Type	Description
`BOLD_ACTION`	`String`	Font bold action identifier
`COLOR_ACTION`	`String`	Font color action identifier

Table 1-1: `HTMLEditorKit` *constants*

Constant	Type	Description
`DEFAULT_CSS`	`String`	Name of the default cascading style sheet to load (e.g., default.css)
`FONT_CHANGE_BIGGER`	`String`	Bigger font size action identifier
`FONT_CHANGE_SMALLER`	`String`	Smaller font size action identifier
`IMG_ALIGN_BOTTOM`	`String`	Align image to bottom action identifier
`IMG_ALIGN_MIDDLE`	`String`	Align image to bottom action identifier
`IMG_ALIGN_TOP`	`String`	Align image to bottom action identifier
`IMG_BORDER`	`String`	Change image border action identifier
`ITALIC_ACTION`	`String`	Font italic action identifier
`LOGICAL_STYLE_ACTION`	`String`	Logical style change action identifier
`PARA_INDENT_LEFT`	`String`	Left-align paragraph action identifier
`PARA_INDENT_RIGHT`	`String`	Right-align paragraph action identifier

`HTMLEditorKit` also defines several inner classes. These inner classes play an integral part in the display and editing of HTML content. Because of their importance, we'll list them here as a quick reference but go into detail on their methods and use in the next section.

Properties

Apart from some of the obvious properties you might expect from an editor kit (the content type and parser, for example), the `HTMLEditorKit` class offers several other display properties, shown in Table 1-2.

Table 1-2: HTMLEditorKit *properties*

Property	Data Type	get	is	set	Default Value
`actions`	`Action[]`	•			standard text actions plus HTML specific actions[1]
`contentType`	`String`	•			`"text/html"`
`defaultCursor`	`Cursor`	•		•	system default cursor
`inputAttributes`	`MutableAttributeSet`	•			defined by *default.css*[2]
`linkCursor`	`Cursor`	•		•	system "hand" cursor
`parser`	`HTMLEditorKit. Parser`	•			`ParserDelegate()`
`styleSheet`	`StyleSheet`	•		•	defined by *default.css*[2]
`viewFactory`	`ViewFactory`	•			`HTMLFactory()`

[1] See Appendix B in *Java Swing*, 2nd edition for a list of these actions
[2] This file is in the *javax/swing/text/html* directory and is most often pulled from the *rt.jar* file

The `actions` property lists actions that can be applied to text. In addition to standard actions like making text bold or italic, the HTML additions let you insert images, hori-

zontal rules, and anchors (among other things). The editor kit stores the MIME type for the document in the `contentType` property. The `defaultCursor` and `linkCursor` properties dictate the cursor to be displayed over normal text (`defaultCursor`) and hyperlinks (`linkCursor`). The visual update of the cursor happens automatically as of SDK 1.3. The `inputAttributes` property returns the attribute set assocated with the current stylesheet. The `parser` property provides easy access to the installed parser that reads content. The `styleSheet` and `viewFactory` properties dictate the presentation of the document. Both of these can be overridden in subclasses.

Constructors

Only one constructor is provided for the `HTMLEditorKit`:

public HTMLEditorKit()
: Construct a simple instance of the `HTMLEditorKit` class. Useful for subclasses.

Editor Kit Methods

If you intend to create your own subclass of `HTMLEditorKit`, these methods will come in handy. Overriding them allows you to control exactly what steps from the parent class are performed (or not performed).

public void install(JEditorPane c)
: This method is called when the editor kit is associated with an editor pane. This is where the default link controller is attached to the editor pane so that hyperlink activity is reported to any `HyperlinkListener` objects correctly.

public void deinstall(JEditorPane c)
: This method can be called to remove an editor kit from an editor pane. This can happen if a new, non-HTML document is loaded into the pane, for example.

HTML Document Methods

These methods let you create, modify, and save HTML documents programmatically. Again, if you simply want to display HTML text in a `JEditorPane`, none of these are necessary.

public Document createDefaultDocument()
: This method creates a blank document that you could use to build an HTML page from scratch. HTML can be inserted using the `insertHTML` method below.

public void insertHTML(HTMLDocument doc, int offset, String html, int popDepth, int pushDepth, HTML.Tag insertTag) throws BadLocationException, IOException
: This method allows you to insert HTML into an existing document. You can insert images, formatted text, or even hyperlinks. The `doc` is the document where the HTML is to be inserted. The `popDepth` and `pushDepth` indicate how many closing

and opening tags, respectively, are required to insert the HTML. For just a simple insert, both can be `0`. The `insertTag` parameter dictates the tag associated with the HTML in the document hierarchy. This is obviously closely associated with the HTML you supply. Inserting content beyond the end of the document or before the beginning throws a `BadLocationException`.

public void read(Reader in, Document doc, int pos) throws IOException, BadLocationException

This method manually reads HTML text into an existing document starting at `pos`. The `Reader` supplied (`in`) should not have a parser attached to it as the parser from the editor kit is used.

public void write(Writer out, Document doc, int pos, int len) throws IOException, BadLocationException

This method allows you to save a portion of a document as HTML. To save the entire document, `pos` should start at `0` and `len` should be `doc.length()`. Examples of the `read()` and `write()` methods appear in the HTML I/O chapter.

HTMLEditorKit Inner Classes

As mentioned earlier, the inner classes for `HTMLEditorKit` play important roles in building and displaying HTML documents. If you plan on creating your own extension of `HTMLEditorKit` to add new functionality, you'll want to pay attention to the `HTMLFactory` and `LinkController` classes in particular. Recall that all of these inner classes are public and static, so you can subclass them at will.

The HTMLEditorKit.HTMLFactory Class

This class is the HTML implementation of the `ViewFactory` class. If you want to modify the display of particular elements or create new display functionality, this is the class to extend.

Constructors

public HTMLEditorKit.HTMLFactory()

A simple no-op constructor handy for subclasses.

Methods

public View create(Element elem)

This sole overridden method returns the `View` used by `JEditorPane` to display `elem`. You can return your own view or return a modified version of `super.create()`.

The HTMLEditorKit.HTMLTextAction Class

An extension of the `javax.swing.text.StyleEditorKit.StyledTextAction`, this class provides a convenient starting point for character-based actions. While you can also subclass `HTMLTextAction` to create your own actions, the `InsertHTMLTextAction` discussed next is a bit easier to use.

Constructors

HTMLEditorKit.HTMLTextAction(String name)

This sole constructor creates a new `Action` named `name`. The `name` is used in menu items or toolbars, as with other `Action` objects.

Methods

protected int elementCountToTag(HTMLDocument doc, int offset, HTML.Tag tag)

This method returns the number of elements from `offset` to the first occurrence of `tag` in `doc`.

protected Element findElementMatchingTag(HTMLDocument doc, int offset, HTML.Tag tag)

This method returns the deepest element matching `tag` beginning at `offset` in the given `doc`.

protected Element[] getElementsAt(HTMLDocument doc, int offset)

This method returns the array of `Element` objects in `doc` that contain `offset`.

protected HTMLDocument getHTMLDocument(JEditorPane e)
protected HTMLEditorKit getHTMLEditorKit(JEditorPane e)

These methods return the `HTMLDocument` and `HTMLEditorKit`, respectively, associated with the editor pane `e`. This can be useful for applying an action generically to text found in the "current" pane of multi-document interfaces.

The HTMLEditorKit.InsertHTMLTextAction Class

This extension of the `HTMLTextAction` class above provides easy access to creating text actions appropriate for an HTML editor.

Fields

protected HTML.Tag addTag
protected HTML.Tag alternateAddTag
protected HTML.Tag alternateParentTag
protected String html
protected HTML.Tag parentTag

All of these protected fields store the parameters passed to the constructors for easy access when the action is applied to an actual document.

Constructors

The constructors for `InsertHTMLTextAction` take parameters that fill out all the necessary pieces for inserting HTML formatting or content into an existing HTML document.

public HTMLEditorKit.InsertHTMLTextAction(String name, String html, HTML.Tag parentTag, HTML.Tag addTag)

For this constructor, `name` represents the name of the `Action` to be used in menus and toolbars. The text to be inserted is `html`. The `addTag` parameter represents the tag type for the inserted text, and `parentTag` represents the type for the content surrounding the inserted HTML. For example, if you were applying the bold font characteristic to some text, the `parentTag` might be `HTML.Tag.P` and `addTag` would be `HTML.Tag.B`. An example of creating such an action appears later in this section.

public HTMLEditorKit.InsertHTMLTextAction(String name, String html, HTML.Tag parentTag, HTML.Tag addTag, HTML.Tag alternateParentTag, HTML.Tag alternateAddTag)

This constructor is similar to the first version but allows for an alternate parent type and then an alternate tag type to be used if the original `parentTag` cannot be found at that offset. For example, if `parentTag` is `HTML.Tag.TABLE` but the action is still useful outside tables, the `alternateParentTag` could be `HTML.Tag.P`. In this case, the `addTag` and `alternateAddTag` might be the same type.

Methods

public void actionPerformed(ActionEvent ae)

This methods kicks off the insertion of HTML. The event `ae` is used to determine the editor pane to receive the inserted HTML.

protected void insertAtBoundary(JEditorPane editor, HTMLDocument doc, int offset, Element insertElement, String html, HTML.Tag parentTag, HTML.Tag addTag)
As its name implies, this protected method is used when HTML is inserted at a boundary. (A boundary in this case is an `offset` in `doc` that exactly matches the beginning offset of the `parentTag`.) It performs the extra work required to keep the tag stack in shape and then calls `insertHTML()`. The `editor` and `doc` arguments are the editor pane and document where the HTML should go. The `offset` argument represents the cursor location or selection start in `doc`. The `insertElement` and `parentTag` arguments are used to calculate the proper number of tag pops and pushes before inserting the HTML (via `html` and `addTag`, which are passed directly to `insertHTML()`).

protected void insertAtBoundry(JEditorPane editor, HTMLDocument doc, int offset, Element insertElement, String html, HTML.Tag parentTag, HTML.Tag addTag)
This method is apparently a deprecated misspelling of `insertAtBoundary()` above.

protected void insertHTML(JEditorPane editor, HTMLDocument doc, int offset, String html, int popDepth, int pushDepth, HTML.Tag addTag)
This method inserts the HTML. It calls the `HTMLEditorKit.insertHTML()` method. The editor argument is used to locate the proper editor kit; all other arguments are passed directly to the editor kit's `insertHTML()` method.

Here's a simple example of such an action that inserts a copyright notice. Here's the few lines we need to make a "copyright" button. (You can see this button in action in the code example of our first HTML editor in the next section.)

First, we define the string of HTML we want for our copyright line:

```
private final static String COPY_HTML =
      "<p>&copy; 1999, O'Reilly &amp; Associates</p>";
```

Next, we create the action for that text:

```
Action a = new HTMLEditorKit.InsertHTMLTextAction("InsertCopyright",
                 COPY_HTML, HTML.Tag.BODY, HTML.Tag.P);
a.putValue(Action.SMALL_ICON, new ImageIcon("icons/cpyrght.gif"));
a.putValue(Action.NAME, "Teletype Text");
```

That action can be added to a toolbar or menu, just like the others in the styled editor from Chapter 23 of *Java Swing*, 2nd edition. We end up with a one-touch button that can add our copyright information to the page.

The HTMLEditorKit.LinkController Class

This class controls the behavior of the mouse with regard to the hyperlinks present in a document. It manages the enter, exit, and activate events. As of SDK 1.3, all three event types are reported correctly, so you rarely need to override this class. However, if you are

building in any specific functionality, such as support for JavaScript, this is one of your first stops. We'll look at an example of providing our own `LinkController` later in this chapter.

Constructors

A single public constructor exists:

public HTMLEditorKit.LinkController()
: This constructor creates a new controller for use with an editor pane. By default the controller is built when the editor kit is installed on a particular pane, so programmers usually do not need to create instances of this class on their own.

Methods

The methods for this class come primarily from the MouseInputAdapter class with one protected method for firing the link events.

protected void activateLink(int pos, JEditorPane editor)
: This method creates and fires a HyperlinkEvent if the document is an instance of `HTMLDocument` and the `href` tag of the link is not `null`.

public void mouseClicked(MouseEvent e)
public void mouseDragged(MouseEvent e)
public void mouseMoved(MouseEvent e)
: These `MouseInputAdapter` methods generate mouse appropriate events around hyperlinks (entering, exiting, and activating).

The HTMLEditorKit.Parser Class

This inner class is responsible for parsing the HTML document. The parser drives the document processing using callbacks defined below. More details on the parser can be found with discussion of the `javax.swing.text.html.parser` package in the "Document Parsers" section of the HTML I/O chapter.

Constructors

One public constructor is available:

public HTMLEditorKit.Parser()
: This constructor creates an instance of the parser, but, as an abstract class, it cannot be used directly.

Methods

public abstract void parse(Reader r, HTMLEditorKit.ParserCallback cb, boolean ignoreCharSet)

This method should be used to parse the input stream from `r` and drive callbacks through the `cb` argument. The `ignoreCharSet` argument can be used to indicate that the stream should be parsed regardless of its character set.

The HTMLEditorKit.ParserCallback Class

This class defines a callback object that can be used in conjunction with the `Parser` inner class. This is the class that handles all the various tokens retrieved from the input stream.

Fields

public static Object IMPLIED

This field is used as an attribute for any HTML tags that are implied. The parser usually adds implied tags when it decides you forgot something in your input stream. A common example of an implied tag is the `<HTML>` tag at the beginning of a document.

Constructors

One no-argument constructor exists. An instance of `HTMLDocument.HTMLReader` (an extension of this callback class) is built for you when a new `HTMLEditorKit` is created, so you normally do not create these yourself—even if you have created your own parser.

public HTMLEditorKit.ParserCallback()

A simple no-op constructor for use by subclasses.

Methods

These methods handle the various types of tokens passed back by the parser.

public void flush()

This method can be overriden to output any remaining data. It is the last method called.

public void handleComment(char[] data, int pos)
public void handleEndOfLineString(String eol)
public void handleEndTag(HTML.Tag t, int pos)
public void handleError(String errorMsg, int pos)
public void handleSimpleTag(HTML.Tag t, MutableAttributeSet a, int pos)
public void handleStartTag(HTML.Tag t, MutableAttributeSet a, int pos)
public void handleText(char[] data, int pos)

These methods take the indicated text (comment, start tag, text, etc.) and get the proper information back into the `HTMLDocument`. In this inner class, the methods are all empty implementations meant to be overriden. The `handleEndOfLineString()` method was introduced in SDK 1.3 and can be overriden to emit the proper end-of-line string (`"\n"`, `"\r"`, or `"\r\n"`). It is called before `flush()`.

Extending HTMLEditorKit

As a quick example of how you might extend this class, let's look at an editor kit that spits out debugging information as we load documents. This example illustrates the steps involved in extending an editor kit and gives us a useful tool for implementing other extensions (such as custom tags and attributes).

The first step, of course, is to create your extended editor kit. In this example, we create a debugging editor kit that spits out the styles loaded and the individual tags it passes by. Here's the code:

```
/*
 * DebugHTMLEditorKit.java
 * A simple extension of the HTMLEditor kit that uses a verbose
 * ViewFactory.
 */

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
import java.net.*;

public class DebugHTMLEditorKit extends HTMLEditorKit {
  public static HTML.Tag ORA = new HTML.UnknownTag("ora");
  public static AttributeSet currentAnchor;

  public void install(JEditorPane paneEditor) {
    super.install(paneEditor);
    StyleSheet ss = getStyleSheet();
    java.util.Enumeration e = ss.getStyleNames();
    while (e.hasMoreElements()) {
      System.out.println(e.nextElement());
```

```
    }
  }

  public ViewFactory getViewFactory() {
    return new VerboseViewFactory();
  }

  public static class VerboseViewFactory extends HTMLEditorKit.HTMLFactory
  {
    public View create(Element elem) {
      System.out.print("Element: " + elem.getName());
      Object o=elem.getAttributes().
                        getAttribute(StyleConstants.NameAttribute);
      HTML.Tag kind = (HTML.Tag) o;
      System.out.println(" view as: " + o);
      dumpElementAttributes(elem);
      return super.create(elem);
    }

    private void dumpElementAttributes(Element elem) {
      AttributeSet attrs = elem.getAttributes();
      java.util.Enumeration names = attrs.getAttributeNames();
      while (names.hasMoreElements()) {
        Object key = names.nextElement();
        System.out.println("  " + key + " : " + attrs.getAttribute(key));
      }
      try {
        System.out.println("  " +
                elem.getDocument().getText(elem.getStartOffset(),
                            elem.getEndOffset()));
      } catch (Exception e) {
      }
    }
  }
}
```

The work of extending `HTMLEditorKit` happens in two methods. First, we override the `install()` method to print out our style information. Notice that we still call `super.install()`. Without that call, we lose the hyperlink functionality. (Sometimes you want that—we'll see an example of modifying hyperlink behavior later in the chapter.)

We next override `getViewFactory()` to return an instance of our own factory (written as an inner class). In our case, we use the `create()` method as a springboard to dump debugging information on each document element that passes through. At the end we return the standard view that `HTMLEditorKit` would have normally produced. It's in this method that you could send back any custom `ViewFactory` you like. If you need to support a specific look and feel through HTML that you can't get from a regular browser and stylesheets, a custom `ViewFactory` and `HTMLEditorKit` may be the ticket.

The HTMLDocument Class

While the editor kit in and of its own right is interesting, we can't do much beyond displaying web pages without looking at the `HTMLDocument` class. This class supports the basic structure of HTML pages. From this class you can view and manipulate the content of a given web page. Not coincidentally, this turns out to be handy for editing documents we plan to save as HTML.

Inner Classes

Several inner classes are defined for `HTMLDocument`. The majority of these classes stand on their own, but you might override the `HTMLReader` class if you wanted to provide your own parser. The `Iterator` class can come in very handy as a tool for examining documents.

public class HTMLDocument.BlockElement
: This class is a representation of a leaf in the HTML document tree structure. Branch elements are stored as `RunElement` objects.

public class HTMLDocument.HTMLReader
: An instance of the `HTMLEditorKit.ParserCallback` class, this class does the work of actually storing parsed data in an `HTMLDocument` structure. If you wanted to alter the default structure or support custom tags, this would be one place to start.

public abstract static class HTMLDocument.Iterator
: The only static inner class, this class can produce an iterator for all the HTML tags of a specific type. An example of using this iterator appears below in the "Internals Example" section.

public class HTMLDocument.RunElement
: This class is a representation of a branch in the HTML document tree structure. Leaf elements are stored as `BlockElement` objects.

Constants

Only one constant is defined for the `HTMLDocument` class:

public static final String AdditionalComments
: This constant is a key for the comments in a document not found in the body. The value associated with the key is a vector of `String` objects.

Properties

The properties for `HTMLDocument` listed in Table 1-3 help define the appearance and behavior of the document.

Table 1-3: `HTMLDocument` *properties*

Property	Data Type	get	is	set	Default Value
`base`	`URL`	•		•	`null`
`tokenThreshold`	`int`	•		•	`Integer.MAX_VALUE`
`styleSheet`	`StyleSheet`	•			
`preservesUnknownTags`	`boolean`	•		•	`true`

The `base` property reflects the base URL in effect for relative hyperlink references. If you use the `setPage()` method of `JEditorPane`, this property is defined automatically from the source of the page. In other cases (such as manually reading an input stream), you may need to define this value yourself. The `tokenThreshold` property determines when display of a page begins. If you want some of your page to show on the screen as soon as possible, pick a relatively low value (say, 50) for this property. The default is the maximum `int` value to effectively load the entire document. The read-only `styleSheet` property gives you access to the stylesheet installed from the editor kit. You can change the initial stylesheet during the construction of this class, but you can also change individual styles defined in the current stylesheet at any time. The last property, `preservesUnknownTags`, determines whether or not non-HTML tags are kept in the document. If you turn this feature off, writing your document to an HTML file expunges unrecognized tags. (See Table 1-6 for a list of "recognized" tags.)

Constructors

public HTMLDocument()
public HTMLDocument(StyleSheet styles)
public HTMLDocument(AbstractDocument.Content c, StyleSheet styles)

These constructors create new `HTMLDocument` objects. The first constructor uses the default stylesheet from the `HTMLEditorKit`. Alternatively, you can supply your own stylesheet or your own content model and stylesheet, respectively. (See Chapter 22, "Styled Text Panes," in *Java Swing*, 2nd edition for more information on `AbstractDocument.Content`.)

Protected Content Methods

Protected content methods construct or update a document. While you rarely need to extend `HTMLDocument` itself, doing so lets you override these methods and construct documents with a more intricate structure. In particular, you can enforce rules regarding the types of leaves allowed to hang off of a given branch.

protected void create(DefaultStyledDocument.ElementSpec[] data)

This method replaces the current document content with data.

protected AbstractDocument.AbstractElement createDefaultRoot()
This method creates a root element for the new document.

protected Element createBranchElement(Element parent, AttributeSet a)
This method returns an `HTMLDocument.BlockElement` object representing the attribute set `a` and attached to `parent`.

protected Element createLeafElement(Element parent, AttributeSet a, int p0, int p1)
This method returns an `HTMLDocument.RunElement` object attached to `parent` representing a run of text from `p0` to `p1`. The run has attributes described by `a`.

protected void insert(int offset, DefaultStyledDocument.ElementSpec[] data) throws BadLocationException
This method accomplishes edits in the document. New information can be batched into data and inserted at `offset` in one shot. This is preferable to locking the document to write changes one at a time as the user types. As of SDK 1.3, this method simply calls the super version from `DefaultStyledDocument`.

protected void insertUpdate(AbstractDocument.DefaultDocumentEvent chng, AttributeSet attr)
This method updates the document structure when an insert occurs (from the `create()` or `insert()` methods, for example).

Reader Methods

public HTMLEditorKit.ParserCallback getReader(int pos)
public HTMLEditorKit.ParserCallback getReader(int pos, int popDepth, int pushDepth, HTML.Tag insertTag)
These methods return an `HTMLReader` object capable of inserting parsed data at `pos`. The second version of the method details how many end tags (`popDepth`) and how many start tags (`pushDepth`) to handle before inserting text. The text begins with `insertTag`.

Public Content Methods

public Element getElement(String id)
public Element getElement(Element e, Object attribute, Object value)
These methods return an element matching the input parameters or `null` if one cannot be found. In the first version, an element with an attribute matching `id` is returned. The second version looks for a child of `e` that has an `attribute` with the given `value`.

public HTMLDocument.Iterator getIterator(HTML.Tag t)
This method returns an `Iterator` that can examine all of the tags of type `t`. An example of using this iterator appears later in this chapter.

public void setParagraphAttributes(int offset, int length, AttributeSet s, boolean replace)
This method sets the attributes (`s`) associated with the paragraph containing `offset`. If `replace` is `false`, `s` is merged with existing attributes. The `length` argument determines how many characters are affected by the new attributes. This is often the entire paragraph.

public void insertAfterEnd(Element elem, String htmlText) throws BadLocationException, IOException

public void insertAfterStart(Element elem, String htmlText) throws BadLocationException, IOException

public void insertBeforeEnd(Element elem, String htmlText) throws BadLocationException, IOException

public void insertBeforeStart(Element elem, String htmlText) throws BadLocationException, IOException
These methods insert `htmlText` at their respective points (after the end, before the start, etc.) relative to `elem`. Note that `elem` cannot be a leaf for the `insertAfterStart()` and `insertBeforeEnd()` methods.

public void setInnerHTML(Element elem, String htmlText) throws BadLocationException, IOException
This method replaces the children of `elem` with `htmlText`.

public void setOuterHTML(Element elem, String htmlText) throws BadLocationException, IOException
This method replaces `elem` in its parent with `htmlText`.

Event Methods

protected void fireChangedUpdate(DocumentEvent e)
This method flags a change in the document. See the `DocumentEvent` class in Chapter 22, "Styled Editor Panes," in *Java Swing*, 2nd edition for more information.

protected void fireUndoableEditUpdate(UndoableEditEvent e)
This method fires an event intended to be caught by `Undo` listeners. It simply calls the super version inherited from `DefaultStyledDocument`. With this method, an HTML editor could easily provide undo support.

public void processHTMLFrameHyperlinkEvent(HTMLFrameHyperlinkEvent e)
This method allows HTML-framed documents to stay inside their frames. Normal `JEditorPane` processing ignores the real target of the hyperlink event and behaves as if the target specified is `"_top"`. You can see an instance of this method in action in the *SimpleLinkListener.java* file defined in the first code example at the beginning of this chapter and in the *ORALinkListener.java* file later in this chapter.

HTMLDocument Inner Classes

Two of the inner classes mentioned for `HTMLDocument` can come in handy when dealing with document content directly: `HTMLReader` and `Iterator`. The other two classes—`BlockElement` and `RunElement`—are simple implementations of `AbstractDocument.BranchElement` and `AbstractDocument.LeafElement`, respectively. Both of these element classes define two read-only properties shown in Table 1-4.

Table 1-4: `HTMLDocument BlockElement and RunElement` *properties*

Property	Data Type	get	is	set	Default Value
`name`	`String`	•			`null`
`resolveParent`	`AttributeSet`	•			`null`

Element Constructors

public HTMLDocument.BlockElement(Element parent, AttributeSet a)

This constructor creates a new branch (`BlockElement`) in the document tree attached to `parent`. The `a` argument defines the new attributes encapsulated in this block.

public HTMLDocument.RunElement(Element parent, AttributeSet a, int offs0, int offs1)

This constructor creates a new leaf (`RunElement`) attached to `parent`. The text associated with the leaf is bounded by `offs0` at the start and `offs1` at the end. The `a` argument defines any style attributes specific to this leaf.

HTMLDocument.HTMLReader

The `HTMLReader` class extends the `HTMLEditorKit.ParserCallback` class. It does the real loading work involved in parsing a given HTML file. You can find more information on the specifics of parsing in the "Document Parsers" section in the HTML I/O chapter. For now, here's the basics of this reader class. Implementing your own parser is tedious, but not difficult. You can start by extending this class and overriding one of the `handle()` methods inherited from `ParserCallback`.

HTMLDocument.HTMLReader Inner Classes

These inner inner (!) classes define the high-level actions to take when inserting various elements into a document. The API for each action consists of two methods:

public void start(HTML.Tag t, MutableAttributeSet attr)

This method handles a new HTML tag, `t`. The attributes defined in `attr` are applied to the new element.

public void end(HTML.Tag t)
: This method handles the ending tag `t`. Presumably this tag matches a previously encountered `start()` call for `t`.

The `TagAction` class serves as the superclass to the remaining actions. The various HTML tags (such as `<A>`, `<IMG>`, `<TABLE>`, and so on) are mapped to one of the following eight actions:

public class HTMLDocument.HTMLReader.BlockAction
public class HTMLDocument.HTMLReader.CharacterAction
public class HTMLDocument.HTMLReader.FormAction
public class HTMLDocument.HTMLReader.HiddenAction
public class HTMLDocument.HTMLReader.IsindexAction
public class HTMLDocument.HTMLReader.ParagraphAction
public class HTMLDocument.HTMLReader.PreAction
public class HTMLDocument.HTMLReader.SpecialAction
: These classes handle the start and end tags for HTML documents. If a particular feature of HTML is not supported (such as the `<SCRIPT>` tag in SDK 1.3), the `HiddenAction` is used.

public class HTMLDocument.HTMLReader.TagAction
: This class is the parent class for the other actions listed above. By default, it ignores the tag.

Fields

Two simple fields provide temporary data storage as the parse progresses:

protected MutableAttributeSet charAttr
: This field simply stores the current attributes in play for document elements. It is continuously updated as starting and ending tags are encountered in the HTML being parsed.

protected Vector parseBuffer
: This field stores a vector of parsed elements. If the vector grows larger than the token threshold for the reader, a `flush()` is attempted.

Constructors

While `HTMLReader`s are normally produced by an `HTMLEditorKit` reading a stream of text, you can instantiate readers using one of two constructors:

public HTMLDocument.HTMLReader(int offset)
public HTMLDocument.HTMLReader(int offset, int popDepth, int pushDepth, HTML.Tag insertTag)

The first version of this constructor starts parsing a stream and inserting the parsed results at `offset`. The second version simply does the same with `popDepth`, `pushDepth`, and `insertTag` all specified. (The first version just calls the second constructor with push and pop depths of `0` and a null `insertTag`.) These constructors also register the handlers for all of the well-known HTML tags.

Control Methods

The following protected methods accomplish the work of getting content from the parsed stream into the actual document. These methods are often called from `ParserCallback`'s `handle()` methods.

protected void addContent(char[] data, int offs, int length)
protected void addContent(char[] data, int offs, int length, boolean generateImpliedPIfNecessary)

These methods are used to add content to the document. The `data` argument is inserted into the document beginning at offset `offs` for `length` characters. The second method uses `generateImpliedPIfNecessary` to create a new block of elements if needed. (The first version calls the second with `true` for this last argument.)

protected void addSpecialElement(HTML.Tag t, MutableAttributeSet a)

This method adds content for tag `t`, which is specified entirely by the attributes in `a`. The HTML comment `<!-- -->` is an example of a tag that maps through this method.

protected void blockOpen(HTML.Tag t, MutableAttributeSet attr)
protected void blockClose(HTML.Tag t)

These methods open and close a block elementof type `t` with the attributes specified by `attr`.

protected void preContent(char[] data)

This method adds content found inside a `<PRE>` tag. This tag ensures whitespace is left intact.

protected void pushCharacterStyle()
protected void popCharacterStyle()

These methods maintain a stack of character styles for handling nested character tags (for example, `<B>`Bold `<I>`and Italic`</I></B>`).

protected void registerTag(HTML.Tag t, HTMLDocument.HTMLReader.TagAction a)
: This method registers a `TagAction` (`a`) for a tag (`t`). All of the standard HTML tags have registered actions, but you can use this method to override those registrations. You can also register actions for custom tags.

protected void textAreaContent(char[] data)
: This method handles content designated for a `<TEXTAREA>` object in a form.

HTMLDocument.Iterator

The other inner class from `HTMLDocument` is a public static class named `Iterator`. You create an iterator for a specific type of tag, for example, the `<A>` tag. You can then retrieve a list of all occurrences of the tag in the document. This is quite useful for examining the structure of a document. An example of using this class (*ORABrowser.java*) appears in the next section.

This class is not associated with the `Iterator` found in the Java 2 Collections API, although it serves a similar purpose.

Properties

A majority of the information provided by the `Iterator` class comes from its properties, listed in Table 1-5. It is up to the programmer to provide the actual information in this abstract inner class. Instances of this class are retrieved through the `getIterator()` method of `HTMLDocument`. (`HTMLDocument` has a package private class called `LeafIterator` that extends this class.)

Table 1-5: `HTMLDocument.Iterator` *properties*

Property	Data Type	get	is	set	Default Value
`attributes`[1]	`AttributeSet`	•			
`endOffset`[1]	`int`	•			
`startOffset`[1]	`int`	•			
`tag`[1]	`HTML.Tag`	•			
`valid`[1]	`boolean`		•		

[1]These properties are represented by `abstract` methods in this class.

The `attributes` property returns the attributes associated with the current tag found by the `Iterator`. The `startOffset` and `endOffset` properties indicate the start and stop positions for the current tag (with the same attributes). The `tag` property indicates which type of tag is being searched for, and `valid` indicates whether or not the iterator has a valid tag.

Methods

Apart from the properties for `Iterator`, only one method is present:

public abstract void next()
: This method moves to the next occurrence of the tag associated with the `Iterator`. If no tag can be found, the `valid` property should return false.

We will see a more complete example of this class in the HTML I/O chapter, but here is an example of using the `Iterator` to find the images in a document:

```
HTMLDocument doc; // must obviously be built somewhere...
String results = "<br>Images:<br>";
Iterator it = doc.getIterator(HTML.Tag.IMG);
while (it.isValid()) {
  AttributeSet a = it.getAttributes();
  String src = a.getAttribute(HTML.Attribute.SRC).toString();
  results += ( "<a href=\"" + src + "\">" + src + "</a><br>" );
  it.next();
}
```

The `results` string now contains a valid HTML listing all of the images (with hyperlinks to same) for the entire document.

HTML

`HTMLEditorKit` and `HTMLDocument` are by far the most important classes in this package. If you are working on custom editors or browsers, you will become quite familiar with those two. Beyond those classes, though, are several supporting classes to be aware of. The `HTML` class is one of those small helper classes that is still integral to the use of `HTMLEditorKit` and `HTMLDocument`.

Inner Classes

The `HTML` class defines three inner classes. Tags and attributes each have their own subclass of these inner classes. The inner class APIs are shown later in this section.

public static class HTML.Attribute
: This class provides a type-safe enumeration for attributes found in HTML tags. Note that simply having an attribute in the list does not imply it is supported by the `HTMLEditorKit`.

public static class HTML.Tag
: This class provides a type-safe enumeration for tags found in HTML. Note that simply having a tag in the list does not imply it is supported by the `HTMLEditorKit`. For example, the `<APPLET>` tag is defined but not supported when displaying HTML documents.

public static class HTML.UnknownTag

This subclass of `HTML.Tag` lets the parser produce valid entries for custom tags without modifying the parser. Any tag not recognized is created as an `UnknownTag` with a name corresponding to that used in the HTML document. For example, we could add `<ORA>` tags to a document and we would see them pass through the editor kit as `UnknownTag` objects. We will see an example of using this class later in the chapter, but if you are serious about creating documents with custom tags, you should explore the XML tools available for Java. For a primer on Java and XML, check out *Java and XML* by Brett McLaughlin (O'Reilly).

Constants

Only one constant is defined:

public static final String NULL_ATTRIBUTE_VALUE

This constant is used to indicate that an attribute has no value associated with it rather than actually storing a null value.

Constructors

The `HTML` class does have a constructor but you will probably use its predefined tags and attributes.

public HTML()

This constructor builds a new instance of the `HTML` class. Again, we are usually after the static elements, so instances of this class are not often used.

Methods

The methods defined in the `HTML` class revolve around retrieving tags and attributes.

public static HTML.Tag getTag(String tagName)
public static HTML.Tag[] getAllTags()

These methods return tags by name or in one entire set, respectively. In the predefined tags, the `tagName` is intuitive; for example, the `<H3>` tag is named `H3`.

public static HTML.Attribute getAttributeKey(String attName)
public static HTML.Attribute[] getAllAttributeKeys()

These methods return attributes by name or in one entire set, respectively. Like the tags, in the predefined attributes, the `attName` is intuitive. For the `<IMG>` tag, the required attribute `SRC` is named `SRC`.

public static int getIntegerAttributeValue(AttributeSet attr, HTML.Attribute key, int def)

This convenience method retrieves the value of key from the attribute set `attr` as an `int`. The `def` argument is a default value to use in case `key` is not defined or an

error occurs converting its value to an integer. This can be useful for parsing things like widths and heights or font sizes.

The HTML.Tag Class

The `HTML.Tag` inner class defines a simple structure for storing a generic HTML tag. It tracks things like the name of the tag and whether or not the tag breaks the flow of the line. The class also contains several fields which are public static references to instances of `HTML.Tag`. There is one field for every tag supported in HTML (regardless of whether or not the tag is supported by `HTMLEditorKit`). These fields are shown in Table 1-6.

Table 1-6: `HTML.Tag` *fields*

A	FORM	OL
ADDRESS	FRAME	OPTION
APPLET	FRAMESET	P
AREA	H1	PARAM
B	H2	PRE
BASE	H3	S
BASEFONT	H4	SAMP
BIG	H5	SCRIPT
BLOCKQUOTE	H6	SELECT
BODY	HEAD	SMALL
BR	HR	STRIKE
CAPTION	HTML	STRONG
CENTER	I	STYLE
CITE	IMG	SUB
CODE	IMPLIED	SUP
COMMENT	INPUT	TABLE
CONTENT	ISINDEX	TD
DD	KBD	TEXTAREA
DFN	LI	TH
DIR	LINK	TITLE
DIV	MAP	TR
DL	MENU	TT
DT	META	U
EM	NOFRAMES	UL
FONT	OBJECT	VAR

The `COMMENT`, `CONTENT`, and `IMPLIED` tags are not part of the HTML specification but are used by `HTMLDocument` to tag comments, text content, and implied paragraphs for text, respectively.

Properties

The `HTML.Tag` class has two read-only properties defined in Table 1-7.

Table 1-7: `HTML.Tag` *properties*

Property	Data Type	get	is	set	Default Value
`block`	`boolean`		•		`false`
`preformatted`	`boolean`		•		`false`

The `block` property indicates whether or not this tag represents a block of content, such as a `<P>` or `<H1>` element. The `preformatted` property indicates whether or not the content is, well, preformatted, as in the case of the `<PRE>` tag.

Constructors

Three constructors exist but the only useful ones are meant for subclasses.

public HTML.Tag()

This sole public constructor can create an instance of an `HTML.Tag`, but the tag has no name or interesting properties.

protected HTML.Tag(String id)
protected HTML.Tag(String id, boolean causesBreak, boolean isBlock)

These methods create new `Tag` objects with the given `id`. The second version allows you to specify the block property (via `isBlock`) and whether or not this tag causes a break in the flow of content (via `causesBreak`). No constructor exists for specifying the `preformatted` property—you must alter the return value of the `isPreformatted()` method directly in your subclass.

Methods

Apart from the methods to read its properties, `HTML.Tag` defines two other methods:

public boolean breaksFlow()

This method returns true if this tag causes a break in the flow of content (for example, the `<BR>` and `<HR>` tags).

public String toString()

This overriden `Object` method returns the tag's name. The `<IMG>` tag, for example, returns `"IMG"`.

The HTML.Attribute Class

The `HTML.Attribute` class serves the same purpose as the `HTML.Tag` class but represents the attributes associated with the various tags rather than the tags themselves. As with the `Tag` class, the majority of information in `Attribute` is stored as fields; each field is an instance of the `Attribute` class. You can see the list of these attributes in Table 1-8.

Table 1-8: `HTML.Attribute` *fields*

ACTION	DUMMY	PROMPT
ALIGN	ENCTYPE	REL
ALINK	ENDTAG	REV
ALT	FACE	ROWS
ARCHIVE	FRAMEBORDER	ROWSPAN
BACKGROUND	HALIGN	SCROLLING
BGCOLOR	HEIGHT	SELECTED
BORDER	HREF	SHAPE
CELLPADDING	HSPACE	SHAPES
CELLSPACING	HTTPEQUIV	SIZE
CHECKED	ID	SRC
CLASS	ISMAP	STANDBY
CLASSID	LANG	START
CLEAR	LANGUAGE	STYLE
CODE	LINK	TARGET
CODEBASE	LOWSRC	TEXT
CODETYPE	MARGINHEIGHT	TITLE
COLOR	MARGINWIDTH	TYPE
COLS	MAXLENGTH	USEMAP
COLSPAN	METHOD	VALIGN
COMMENT	MULTIPLE	VALUE
COMPACT	N	VALUETYPE
CONTENT	NAME	VERSION
COORDS	NOHREF	VLINK
DATA	NORESIZE	VSPACE
DECLARE	NOSHADE	WIDTH
DIR	NOWRAP	

Methods

Unlike the `Tag` class, the `Attribute` class has no public constructors and only one method:

public String toString()

This overriden `Object` method returns the attribute's name. The `SRC` attribute, for example, displays as `"SRC"`.

The HTML.UnknownTag Class

This class allows the parser to deal with custom tags without any programmer specific modifications. Custom tags are wrapped in this class. If you want to use custom tags without writing your own parser, this class will help.

Construtors

`UnknownTag` has one public constructor:

public HTML.UnknownTag(String id)

This constructor allows you to build an instance of an `UnknownTag` with a specific id. In turn, you can use your `UnknownTag` object to compare against tags coming out of the normal parser process. An example of this technique appears at the end of the next section.

Methods

`UnknownTag` overrides two methods from `Object`: `hashCode()` and `equals()`. The `hashCode()` method makes sure that your `UnknownTag` objects still play nice with the default hash mechanisms available in Java. The `equals()` method allows you to compare your tag to other tags. A comparison succeeds when the compared object is also an `UnknownTag` and has the same `id`.

The StyleSheet Class

The last big piece of the HTML display puzzle comes in the form of a `StyleSheet` class. This class defines the mechanisms for supporting cascading style sheets (CSS) in HTML. A stylesheet basically lists the styles available in a document along with the display characteristics of those styles. For example, all `<P>` paragraphs should use the Helvetica font at 14 points, all `<H1>` text should use Helvetica at 28 points, and all `<H2>` text should use italicized Helvetica at 20 points. This is, of course, an incomplete list, but you get the idea.

Stylesheet support is still incomplete as of SDK 1.4. Each successive release of the `swing.text.html` package has included remarkable improvements. If you rely on this package for part of your application, you should make sure you have the latest possible version of Java. However, despite the incomplete nature of stylesheet support, you can still put it to use in your own applications. We will look at an example of modi-

fying the look of a standard HTML document through stylesheets at the end of this section.

Inner Classes

The `StyleSheet` class has two inner classes that help render HTML:

public static class StyleSheet.BoxPainter
public static class StyleSheet.ListPainter

Both classes format and render information on the page. `BoxPainter` maintains inset and background integrity when rendering. `ListPainter` renders individual items in a list. `ListPainter` knows which item is being rendered so that numbered (or lettered) lists can be produced. Both classes serve as delegates of `Views`. Neither have public (or even protected) constructors.

Properties

The `StyleSheet` class defines a few properties as shown in Table 1-9.

Table 1-9. `StyleSheet` *properties*

Property	Data Type	get	is	set	Default Value
`baseFontSize`	`int, String`[1]			•	`4 (from CSS)`
`base`	`URL`	•		•	`null`
`styleSheets`	`StyleSheet[]`	•			

[1]This property can be set as either an int or a String.

The `baseFontSize` property allows you to dictate the base font size for styles. The base size is used for normal `<P>` text, with larger fonts used for headers and smaller fonts used for sub- and superscript. This font size is also the base for relative font sizes specified via `<FONT>` tags. The `base` property is the URL root for any relative URLs defined in the document. This property is normally set via the `setPage()` method of `JEditorPane` or the `<BASE>` header tag but can be set manually if you are building a new document from scratch. The `styleSheets` property returns an array of contained stylesheets. Stylesheets can be arranged in a hierarchy to make construction and maintenance easier.

Constructors

One simple constructor exists:

public StyleSheet()

This constructor creates a new, empty stylesheet. You can then customize the sheet before installing it for use with `HTMLEditorKit` objects. (You can also retrieve

the current stylesheet directly from the editor kit and modify only the parts that interest to you.)

Rule and Style Methods

Several methods are defined for reading and writing new style rules into a `Sheet`. The `Style` type mentioned in several methods below comes from the interface `javax.swing.text.Style`.

public Style getRule(HTML.Tag t, Element e)
public Style getRule(String selector)

These methods return the style associated with a tag (`t`) contained in an element (`e`) or by a `selector`, respectively. The `selector` is a string that represents the tag hierarchy for the desired style. For example, a valid selector for normal paragraphs might be given as `"html body p"`.

public void addRule(String rule)

This method allows you to add a rule (or a set of rules) to the stylesheet. The rule argument should be a valid CSS rule. For example, here's a rule that makes all level one headers blue and italicized:

```
h1 {color: blue; font-style: italic}
```

public void loadRules(Reader in, URL ref) throws IOException

This method allows you to add a series of rules from an input stream (`in`). The `ref` argument specifies the location of the input stream. If the stream is generated and did not come from a specific URL, `ref` can be `null`. However, any relative URLs encountered are based on `ref`.

public void removeStyle(String nm)

This method allows you to remove a style named `nm`. The name follows the same pattern as the selector argument for the `getRule()` method above.

public void addStyleSheet(StyleSheet ss)
public void removeStyleSheet(StyleSheet ss)
public void importStyleSheet(URL url)

These methods allow you to add, remove, and import stylesheets respectively. For the import method, `url` should point to a valid CSS document.

AttributeSet Methods

Beyond dealing with entire stylesheets, you can also manipulate individual attributes of given styles. As most styles are implemented with attribute sets, these methods make it easier to tweak styles instead of creating them from scratch.

public AttributeSet addAttribute(AttributeSet old, Object key, Object value)
public AttributeSet addAttributes(AttributeSet old, AttributeSet attr)
public AttributeSet removeAttribute(AttributeSet old, Object key)
public AttributeSet removeAttributes(AttributeSet old, Enumeration names)
public AttributeSet removeAttributes(AttributeSet old, AttributeSet attrs)

These methods allow you to add and remove attributes from an `old` set and return the new one.

public void addCSSAttribute(MutableAttributeSet attr, CSS.Attribute key, String value)
public boolean addCSSAttributeFromHTML(MutableAttributeSet attr, CSS.Attribute key, String value)

These methods add attributes to a `MutableAttributeSet` rather than creating a new set based on an old one. Both versions require the set to modify, a valid `key`, and a value to associate with that `key`. The `addCSSAttributeFromHTML()` method parses the value argument from HTML based on `key`. It returns `true` if it finds a valid value for the given key, and false otherwise.

public AttributeSet getDeclaration(String decl)

This method translates a valid CSS declaration, `decl`, into an attribute set. If `decl` is `null`, an empty attribute set is returned.

public AttributeSet getViewAttributes(View v)

This method grabs an attribute set for use when displaying the view `v`.

public AttributeSet translateHTMLToCSS(AttributeSet htmlAttrSet)

Similar to `addCSSAttributeFromHTML()`, this method adds a series of CSS attributes by translating a set of HTML attributes. `StyleSheet` uses this method to convert HTML attributes into proper CSS attributes.

protected StyleContext.SmallAttributeSet createSmallAttributeSet(AttributeSet a)
protected MutableAttributeSet createLargeAttributeSet(AttributeSet a)

These protected methods create new `AttributeSet` objects that have differing properties. The `SmallAttributeSet` should be a compact set intended to be shared. The `LargeAttributeSet`, on the other hand, should provide quick access to a large number of attributes and should not be shared. Neither of these methods are currently used but are provided as hooks for subclasses that want to alter the standard use of `SimpleAttributeSet` for storage.

Helper Methods

Just to make your life easier—although we question why you are playing with this class if you have an easy life—these helper methods provide convenient access to specific parts of attributes sets and other properties common to styles.

public Font getFont(AttributeSet a)
public Color getForeground(AttributeSet a)
public Color getBackground(AttributeSet a)

These methods return the font, foreground color, and background color associated with the given attribute set `a`. Again, this is for convenience; you can certainly retrieve these values directly from the attribute set if you want to.

public StyleSheet.BoxPainter getBoxPainter(AttributeSet a)
public StyleSheet.ListPainter getListPainter(AttributeSet a)

These methods return new `BoxPainter` and `ListPainter` objects for use with the given attribute set `a`. As `BoxPainter` and `ListPainter` have no public constructors, this is the preferred mechanism for retrieving them.

public float getPointSize(int index)
public float getPointSize(String size)

These methods return the `float` value of a font size specified by `index` (pulled from a size map based on the `baseFontSize`) or `size` (converted from a string first to see if it is a relative or absolute size). For the second version, relative sizes are specified by a preceding + or −.

public static int getIndexOfSize(float pt)

This static method is the reverse of the `getPointSize()` methods above. Given a point size, this method returns the nearest index in the standard size map.

public Color stringToColor(String string)

This method returns a `Color` object corresponding to `string`. The string passed in can be any of the names listed in Table 1-10 or a hexadecimal RGB value in the format "#RRGGBB". If the color can't be found, `null` is returned.

Table 1-10: Color names understood by `StyleSheet.stringToColor()`.

Aqua	Gray	Navy	Silver
Black	Green	Olive	Teal
Blue	Lime	Purple	White
Fuchsia	Maroon	Red	Yellow

The CSS Class

Stylesheets in the Swing HTML package are predicated on the use of CSS. The `CSS` class defines attributes found in the HTML 4.0 specification. Although stylesheet support was incomplete at the time this book went to press, any implementation of CSS would presumably make extensive use of this class. In the current (1.4 SDK) implementation, CSS provides several package private helper methods that do make some style support available. Most of this support comes from hard-coding editor kits, not from `<STYLE>` tags defined in the HTML text itself.

Constructors

The `CSS` class has one public constructor:

public CSS()

This constructor creates a new `CSS` object. The `StyleSheet` class normally builds this for you and keeps a package private reference to that instance.

Methods

Despite having a public constructor, only two static methods are available to programmers:

public static final CSS.Attribute getAttribute(String name)
public static CSS.Attribute[] getAllAttributeKeys()

These methods retrieve an individual attribute or an array of all attribute keys, respectively. The `CSS.Attribute` class is similar to the `HTML.Tag` and `HTML.Attribute` classes and is defined below.

The CSS.Attribute Class

Obviously the `CSS` class is not very interesting except that it supplies you with a list of attributes. Similar to the `HTML.Attribute` class, those attributes come from the closed set of constants defined in the `Attribute` inner class. These attributes are listed in Table 1-11. Again, some of the attributes are defined but not necessarily supported. We noted the supported attributes as reported by the JavaDoc for `CSS`.

Table 1-11: CSS attributes defined as constants in `CSS.Attribute`

BACKGROUND[s]	FONT_WEIGHT[s]
BACKGROUND_ATTACHMENT[m]	HEIGHT[m]
BACKGROUND_COLOR[se]	LETTER_SPACING[m]
BACKGROUND_IMAGE[s]	LINE_HEIGHT[m]
BACKGROUND_POSITION[s]	LIST_STYLE[m]
BACKGROUND_REPEAT[s]	LIST_STYLE_IMAGE
BORDER[m]	LIST_STYLE_POSITION[s]
BORDER_BOTTOM[m]	LIST_STYLE_TYPE[s]
BORDER_BOTTOM_WIDTH[m]	MARGIN[s]
BORDER_COLOR	MARGIN_BOTTOM[s]
BORDER_LEFT[m]	MARGIN_LEFT[s]
BORDER_LEFT_WIDTH[m]	MARGIN_RIGHT[s]
BORDER_RIGHT[m]	MARGIN_TOP[s]
BORDER_RIGHT_WIDTH[m]	PADDING

[m]*modeled but not rendered*
[s]*supported;* [se]*supported with exceptions;* [sr]*support includes relative units*

Table 1-11: CSS attributes defined as constants in `CSS.Attribute`

BORDER_STYLE[se]	PADDING_BOTTOM[s]
BORDER_TOP[m]	PADDING_LEFT[s]
BORDER_TOP_WIDTH[m]	PADDING_RIGHT[s]
BORDER_WIDTH[m]	PADDING_TOP[s]
CLEAR[m]	TEXT_ALIGN[se]
COLOR[s]	TEXT_DECORATION[se]
DISPLAY[m]	TEXT_INDENT[m]
FLOAT[m]	TEXT_TRANSFORM[m]
FONT[s]	VERTICAL_ALIGN[se]
FONT_FAMILY[s]	WHITE_SPACE[m]
FONT_SIZE[sr]	WIDTH[m]
FONT_STYLE[s]	WORD_SPACING[m]
FONT_VARIANT	

[m]*modeled but not rendered*
[s]*supported;* [se]*supported with exceptions;* [sr]*support includes relative units*

Properties

Two properties are defined for each attribute as shown in Table 1-12.

Table 1-12. `CSS.Attribute` *properties*

Property	Data Type	get	is	set	Default Value
`defaultValue`	`String`	•			defined in constructor
`inherited`	`boolean`		•		defined in constructor

The `defaultValue` property should return the common default for the given attribute. The default value is specified to the (private) constructor. The inherited property indicates whether or not the attribute is an inherited attribute. Attributes such as color and font family are inherited.

Methods

Also similar to the `HTML.Attribute` class, `CSS.Attribute` overrides the `toString()` method from the `Object` class:

public String toString()

This method returns a readable version of the attribute, e.g., its name. The BACKGROUND_COLOR attribute, for example, returns `"background-color"`.

Internals Example

Before we tackle the business of reading and writing HTML directly, let's put some of the new classes we have available to use. In this example we're going to build another rudimentary version of a browser from scratch with the following key features:

- The <H1> tag now uses a 48-point sans-serif font
- A custom <ORA> tag is recognized; text found inside the <ORA> and </ORA> tags is hidden
- Hyperlink events are overriden to include information from their <A> tag

This functionality is mostly embedded in the *ORAEditorKit.java* file listed later in this chapter. However, to make proper use of the new functionality, we need to create a browser that uses our editor kit and a link listener that shows we really did receive the new information.

First, here's the new *ORABrowser.java* file:

```
// ORABrowser.java
//

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

public class ORABrowser extends JFrame {

  JEditorPane htmlPane;

  public ORABrowser() {
    this("http://www.oreilly.com/");
  }

  public ORABrowser(String url) {
    super("ORABrowser 1.0");
    setSize(400,500);
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    if (!url.startsWith("http:")) {
      try {
        url = (new java.io.File(url)).toURL().toString();
      } catch (java.net.MalformedURLException mfe) {
        System.err.println("Invalid url: " + url);
        System.err.println("Using default of www.oreilly.com");
        url = "http://www.oreilly.com/";
      }
    }
```

```
    try {
      htmlPane = new JEditorPane();
      htmlPane.setEditable(false);

      // Here's where we force the pane to use our new editor kit
      htmlPane.setEditorKitForContentType("text/html",
                                          new ORAEditorKit());

      // And add our smarter listener
      htmlPane.addHyperlinkListener(new ORALinkListener(htmlPane));

      htmlPane.setPage(url);
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    JScrollPane jsp = new JScrollPane(htmlPane);
    getContentPane().add(jsp);

    // set up a menubar
    JMenuBar jmb = new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    fileMenu.add(new ExitAction());
    JMenu viewMenu = new JMenu("View");
    viewMenu.add(new InfoAction());
    jmb.add(fileMenu);
    jmb.add(viewMenu);
    setJMenuBar(jmb);
  }

  public static void main(String args[]) {
    ORABrowser bb = null;
    if (args.length == 1) {
      bb = new ORABrowser(args[0]);
    }
    else if (args.length == 0) {
      bb = new ORABrowser();
    }
    else {
      System.err.println("Usage is: java ORABrowser [url]");
      System.exit(1);
    }
    bb.setVisible(true);
  }

  // All the action classes
  public class ExitAction extends AbstractAction {
    public ExitAction() {
      super("Exit");
    }
    public void actionPerformed(ActionEvent ae) {
      System.exit(0);
    }
  }
```

```
  public class InfoAction extends AbstractAction {
    public InfoAction() {
      super("View Page Info...");
    }
    public void actionPerformed(ActionEvent ae) {
      if (htmlPane != null) {
        HTMLDocument doc = (HTMLDocument)htmlPane.getDocument();
        if (doc != null) {
          new HTMLDocInfoFrame(doc);
        }
        else {
          System.err.println("Null document, cannot display info.");
        }
      }
      else {
        System.err.println("Null pane, cannot display info.");
      }
    }
  }
}
```

Notice the `InfoAction` inner class. (We'll look at the `HTMLDocInfoFrame` in a few pages.) This action pops up a window that displays the links and images contained in the current document. The hyperlink listener we use now is the *ORALinkListener.java* file:

```
/*
 * ORALinkListener.java
 * A hyperlink listener for use with JEditorPane.  This
 * listener changes the cursor over hotspots based on enter/exit
 * events and also load a new page when a valid hyperlink is clicked.
 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class ORALinkListener implements HyperlinkListener {

  private JEditorPane pane;       // The pane we're using to display HTML
  private JTextField  urlField;   // An optional text field for showing
                                  // the current URL being displayed
  private JLabel statusBar;       // An optional label for showing where
                                  // a link would take you
  private HashMap localKeys = new HashMap();

  // ... skip over the stuff that we picked up from the
  // ... SimpleLinkListener in the previous example
```

```
public void hyperlinkUpdate(HyperlinkEvent he) {
  HyperlinkEvent.EventType type = he.getEventType();
  // Ok.  Decide which event we got...
  if (he instanceof HTMLFrameHyperlinkEvent) {
    JOptionPane.showMessageDialog(null, "Frame Event");
  }
  if (type == HyperlinkEvent.EventType.ENTERED) {
    // Enter event.  Go the the "hand" cursor and fill in the status bar
    AttributeSet anchor = ORAEditorKit.currentAnchor;
    Object att = getLocalAttributeKey("onmouseover", anchor);
    if (att != null) {
      // Ok, at least there's one onmouseover event...
      String request = (String)anchor.getAttribute(att);
      if (request != null) {
        handleORARequest(request);
      }
    }
    if (statusBar != null) {
      statusBar.setText(he.getURL().toString());
    }
  }
  else if (type == HyperlinkEvent.EventType.EXITED) {
    // Exit event.  Go clear the status bar
    if (statusBar != null) {
      statusBar.setText(" "); // must be a space or it disappears
    }
  }
  else {
    // Jump event.  Get the url, and if it's not null, switch to that
    // page in the main editor pane and update the "site url" label.
    if (he instanceof HTMLFrameHyperlinkEvent) {
      HTMLFrameHyperlinkEvent  evt = (HTMLFrameHyperlinkEvent)he;
      HTMLDocument doc = (HTMLDocument)pane.getDocument();
      doc.processHTMLFrameHyperlinkEvent(evt);
    } else {
      try {
        pane.setPage(he.getURL());
        if (urlField != null) {
          urlField.setText(he.getURL().toString());
        }
      }
      catch (FileNotFoundException fnfe) {
        pane.setText("Could not open file: <tt>" + he.getURL() +
                     "</tt>.<hr>");
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

```
  // Just a simple helper method to display "alert" boxes.
  // We understand the following request:
  //  <a ... onmouseover="orascript:alert('Message');" ... >
  private void handleORARequest(String request) {
    String req = request.toLowerCase();
    int start = req.indexOf("orascript:alert");
    if (start != -1) {
      // Hooray! Something we can deal with.
      int end = request.lastIndexOf("'");
      String msg = request.substring(start + 17, end);
      JOptionPane.showMessageDialog(null, msg);
    }
    else {
      JOptionPane.showMessageDialog(null, "Unknown ORAScript:\n" +
                                    request);
    }
  }
}
```

This new link listener class does one important thing: in the hyperlinkUpdate() method, we check for an active current anchor from the new editor kit. If we find one, we can retrieve the <A> tag that was activated and look at any of its attributes—including attributes not specifically supported by the `HTMLEditorKit` class. In this example we look for an "onmouseover" attribute. If we find that attribute, we pass it to the `handleORARequest()` method. If it's a simple `alert()` call, we parse it; otherwise we just show the user what the unknown request was. Not a complete JavaScript implementation, but it's a start.

The real work in this example comes from the `ORAEditorKit` class defined below. Be sure to look at the `install()` method and the new `LinkController` inner class.

```
/*
 * ORAEditorKit.java
 * Another extension of the HTMLEditor kit that uses a verbose
 * ViewFactory. This implementation also installs a custom
 * <H1> style and watches for <ORA> entries.
 */

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;
import java.io.Serializable;
import java.net.*;

public class ORAEditorKit extends HTMLEditorKit {
  // We're going to add a custom ORA tag and hide any enclosed text
  // from the end user.
  public static HTML.Tag ORA = new HTML.UnknownTag("ora");
```

```
// We also track the current <a> tag, in an
// admittedly non-thread-friendly manner.
public static AttributeSet currentAnchor;

public void install(JEditorPane paneEditor) {
  LinkController linkController = new LinkController() ;

  // Use our patched link controller for both mouse clicks and motions
  paneEditor.addMouseListener(linkController) ;
  paneEditor.addMouseMotionListener(linkController) ;

  // Manually set our H1 tags to use a 48-pt font.
  StyleSheet ss = getStyleSheet();
  Style s = ss.getStyle("h1");
  StyleConstants.setFontSize(s, 48);
}

// the new patched link controller
public static class LinkController extends HTMLEditorKit.LinkController
  implements Serializable
{
  public void mouseClicked(MouseEvent me) {
    JEditorPane jep = (JEditorPane)me.getSource();
    Document doc = jep.getDocument();
    if (doc instanceof HTMLDocument) {
      HTMLDocument hdoc = (HTMLDocument) doc;
      int pos = jep.viewToModel(me.getPoint());
      Element e = hdoc.getCharacterElement(pos);
      AttributeSet a = e.getAttributes();

      // all that work gets us the attribute set associated with
      // the current <a> tag, which we now store:
      currentAnchor = (AttributeSet) a.getAttribute(HTML.Tag.A);
    }
    super.mouseClicked(me);
  }

  // and ditto for the mouseMoved method as we pass over links...
  public void mouseMoved(MouseEvent me) {
    JEditorPane jep = (JEditorPane)me.getSource();
    Document doc = jep.getDocument();
    if (doc instanceof HTMLDocument) {
      HTMLDocument hdoc = (HTMLDocument) doc;
      int pos = jep.viewToModel(me.getPoint());
      Element e = hdoc.getCharacterElement(pos);
      AttributeSet a = e.getAttributes();
      currentAnchor = (AttributeSet) a.getAttribute(HTML.Tag.A);
    }
    super.mouseMoved(me);
  }
}

// We're overriding some of the view information, so supply our
// own version of the factory.
```

```
public ViewFactory getViewFactory() {
  return new VerboseViewFactory();
}

public static class VerboseViewFactory extends HTMLEditorKit.HTMLFactory
{
  public boolean hideText = false;

  // This view factory spits out lots of debugging information and also
  // hides any content between <ORA> and </ORA> tags.
  public View create(Element elem) {
    System.out.print("Element: " + elem.getName());
    Object o =
      elem.getAttributes().getAttribute(StyleConstants.NameAttribute);
    HTML.Tag kind = (HTML.Tag) o;
    System.out.println(" view as: " + o);
    dumpElementAttributes(elem);
    if (kind.equals(ORA)) {
      hideText = !hideText;
      System.out.println("Found ORA tag, hiding: " + hideText);
    }
    return hideText ? new NoView(elem) : super.create(elem);
  }

  private void dumpElementAttributes(Element elem) {
    AttributeSet attrs = elem.getAttributes();
    java.util.Enumeration names = attrs.getAttributeNames();
    while (names.hasMoreElements()) {
      Object key = names.nextElement();
      System.out.println("  " + key + " : " + attrs.getAttribute(key));
    }
    try {
      System.out.println("  " +
                    elem.getDocument().getText(elem.getStartOffset(),
                                  elem.getEndOffset()));
    } catch (Exception e) {
    }
  }
}

// Our own custom "no view" class to hide any content found inside
// <ORA> and </ORA> tags.  Basically, we return a 0x0 bounding box
// and do nothing in the paint() method.

public static class NoView extends View {
  public NoView(Element elem) {
    super(elem);
    setSize(0.0f, 0.0f);
  }

  public int viewToModel(float fx, float fy, Shape a,
                         Position.Bias[] bias)
  {
    return 0;
```

```
    }
    public Shape modelToView(int pos, Shape a, Position.Bias b)
      throws BadLocationException
    {
      return new Rectangle(0, 0);
    }
    public float getPreferredSpan(int axis) {
      return 0.0f;
    }
    public void paint(Graphics g, Shape allocation) {
    }
  }
}
```

This class helps accomplish all three of our original goals: the new `H1` style, the new information in hyperlink events, and the support for an `<ORA>` tag.

We specify the new `<H1>` style in the `install()` method. Before leaving `install()`, we grab the current stylesheet associated with the document and modify the attributes for the `"h1"` style. (Figure 1-3 shows off the new font.) Other changes to default styles could be applied similarly. Regrettably, there is no mechanism for reading this information from the HTML source itself.

The hyperlink information is bundled up in the form of a `LinkController` inner class. In the original `HTMLEditorKit` class, a link listener is built and installed for us. However, since we never call `super.install()` here, we have complete control over what gets added. We build our own `LinkController` and use it to store the current `<A>` tag whenever a hyperlink event is reported. The anchor tag can then be accessed by other listeners (like the `ORALinkListener` above) as we see fit. Now those hyperlink listeners can access any target or JavaScript-like attributes that may be present in a link.

The `<ORA>` tag support happens in three steps. First, we build our own ORA tag using the `HTML.UnknownTag` class. This gives us our second piece—something with which to compare to the incoming elements in the `create()` method of our custom `ViewFactory`. If we find a matching tag, we toggle the viewability of content. When we return the recently parsed content, if we are hiding text because of the `<ORA>` tag, we return the third piece—a "no view" object that does not render on the page. You could certainly use a similar structure to recognize and render your own custom tags.

Figure 1-3 shows a screenshot of the simple browser and the alert message from running the mouse over the "test" link.

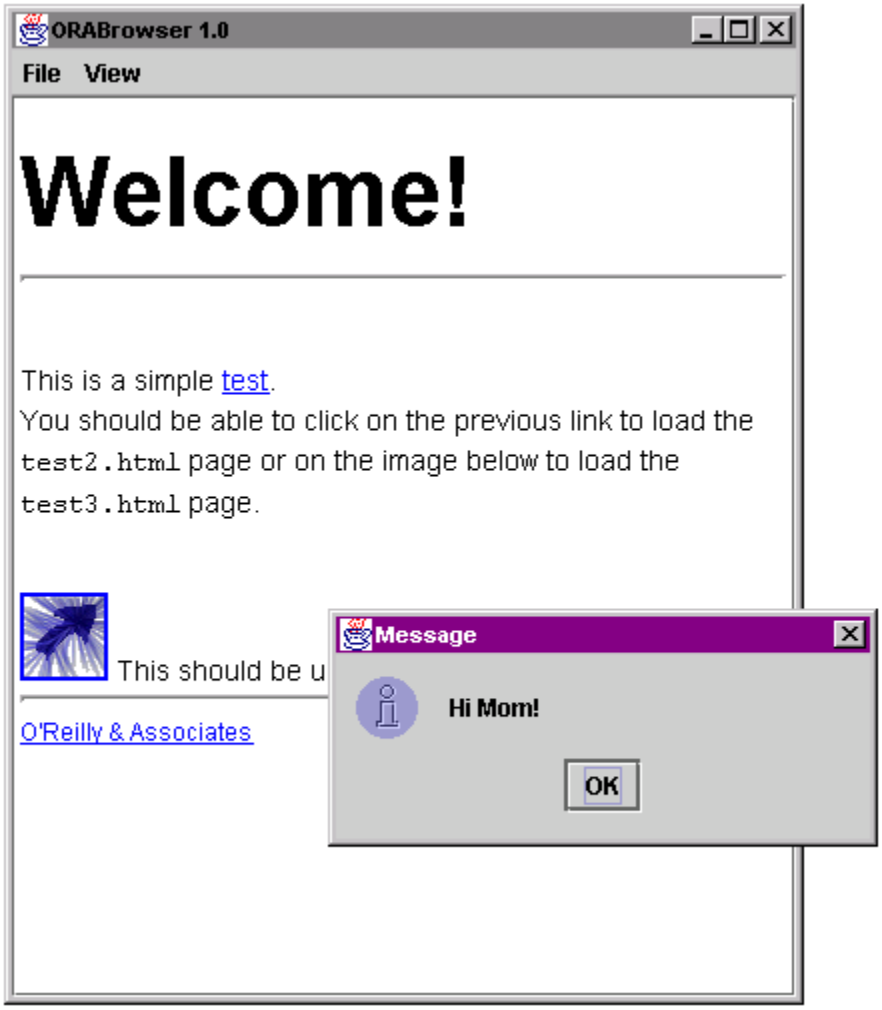

Figure 1-3. The new ORABrowser with big <H1> tags and minimal "ORAScript" support

Document Structure

One other interesting bit of information we can gather from our `HTMLDocument` objects is the list of tags present. With the help of the `HTMLDocInfoFrame` class listed below, we can produce a page similar to the "View Page Info..." page produced by popular browsers such as Netscape Navigator. We use the `HTMLDocument`.Iterator and a custom iterator to show the `<A>`, `<IMG>`, and `<ORA>` tags found in a document.

```
/*
 * HTMLDocInfoFrame.java
```

```
 * A display tool for the structure of an HTMLDocument.
 * Using iterators, you can display any tags you like.
 */

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.text.html.*;
import java.awt.BorderLayout;
import java.util.Enumeration;

public class HTMLDocInfoFrame extends JFrame {

  public HTMLDocInfoFrame(HTMLDocument doc) {
    super("Page Info");
    setSize(400,300);

    JEditorPane jep = new JEditorPane();
    getContentPane().add(new JScrollPane(jep), BorderLayout.CENTER);
    jep.setEditable(false);

    // Now fill the text area
    String results = "<html><body><h1>Page Structure</h1><hr><br>\n";
    results += "Links:<br>";
    HTMLDocument.Iterator it = doc.getIterator(HTML.Tag.A);
    while (it.isValid()) {
      String href = it.getAttributes()
        .getAttribute(HTML.Attribute.HREF).toString();
      results += ("<a href=\"" + href + "\">" + href + "</a><br>\n");
      it.next();
    }

    // Now look for <IMG> tags using our custom iterator
    results += "<br>Images:<br>";
    TagIterator it2 = new TagIterator(HTML.Tag.IMG, doc);
    while (it2.isValid()) {
      AttributeSet a = it2.getAttributes();
      String src = a.getAttribute(HTML.Attribute.SRC).toString();
      results += ( "<a href=\"" + src + "\">Image: " + src + "</a><br>" );
      it2.next();
    }

    // Now look for <ORA> tags with the custom iterator
    results += "<br><hr><br>Unknowns:<br>";
    it2 = new TagIterator(new HTML.UnknownTag("ora"), doc);
    while (it2.isValid()) {
      AttributeSet a = it2.getAttributes();
      Enumeration enum = a.getAttributeNames();
      results += "<ul>";
      while (enum.hasMoreElements()) {
        Object o = enum.nextElement();
        results += ( "<li> " + o + " (" + o.getClass() + ") : " +
                a.getAttribute(o));
      }
      results += "</ul><br>";
```

```
      it2.next();
    }

    jep.setContentType("text/html");
    jep.setText(results);
    HTMLDocument newDoc = (HTMLDocument)jep.getDocument();
    newDoc.setBase(doc.getBase());

    setVisible(true);
  }

  // Our own version of an Iterator that allows us to look at non-leaf
  // tags as well.
  public static class TagIterator {

    private HTML.Tag tag;
    private ElementIterator pos;

    TagIterator(HTML.Tag t, Document doc) {
      tag = t;
      pos = new ElementIterator(doc);
      next();
    }

    /**
     * Fetch the attributes for this tag.
     */
    public AttributeSet getAttributes() {
      Element elem = pos.current();
      if (elem != null) {
        AttributeSet a =
          (AttributeSet)elem.getAttributes().getAttribute(tag);
        if (a == null) {
          return elem.getAttributes();
        }
        return a;
      }
      return null;
    }

    public void next() {
      for (pos.next(); isValid(); pos.next()) {
        Element elem = pos.current();
        if (elem.getName().equals(tag.toString())) {
          break;
        }
        AttributeSet a = pos.current().getAttributes();
        if (a.isDefined(tag)) {
          // we found the next one
          break;
        }
      }
    }
```

```
    public HTML.Tag getTag() {  return tag; }

    public boolean isValid() {
      return (pos.current() != null);
    }
  }
}
```

Notice how the attributes associated with the tags can be used to dissect the document. You could use similar tricks to debug the creation of a document or to show off a fancier version of this document info frame. Figure 1-4 shows the resulting frame for our simple test page.

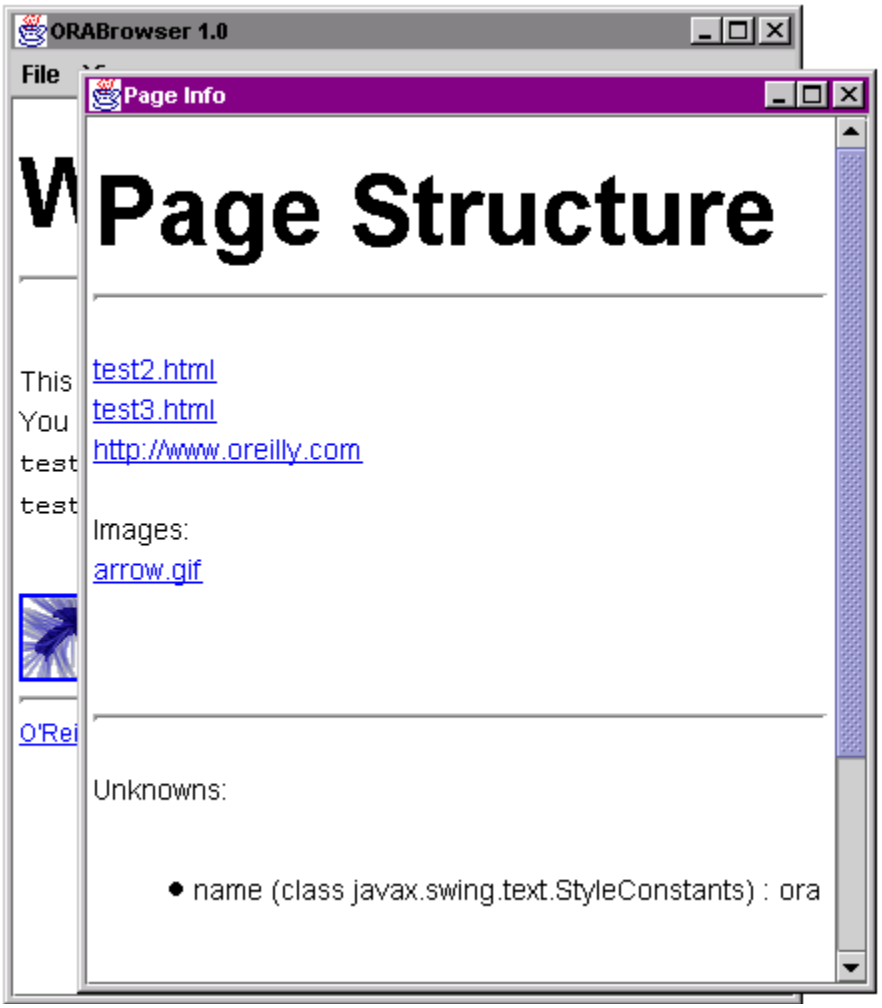

Figure 1-4. The HTMLDocInfoFrame in action on a simple HTML page.

HTML I/O

The business or reading and writing HTML is covered in the HTML I/O chapter, which of course you can download from our web site, *http://www.oreilly.com/catalog/jswing2*, just as you did this chapter.