# 2

# *HTML I/O*

This is the second of two supplemental chapters for *Java Swing*, 2nd edition, available from our web site, *http://www.oreilly.com/catalog/jswing2*. If you made it through the previous chapter on the `HTMLEditorKit`, you may be impressed with what it can handle. You may also be wondering how it loads and displays all that information. If so, this chapter is for you. If not (and it's perfectly reasonable not to care) you can skip over this chapter without penalty. You might still want to look at the HTML editor example at the end of the chapter, though. Whether you ever do it, it is still cool to see a fairly simple Java program that can create and save an HTML document that you can publish on the web.

Editor kits in general serve as a collection point for everything you need to read, write, and display some type of content. The `HTMLEditorKit` in particular serves as just such a collection point for HTML content. In addition to the `HTMLEditorKit` class proper, several supporting classes aid in the process of reading HTML, displaying it in a `JEditorPane`, and writing it out to a stream with a high amount of fidelity. Specifically, we examine the `javax.swing.text.html.parser` package, the `HTMLWriter` class, and several `View` classes for use with the `HTMLEditorKit`. The APIs for this section remain a bit opaque, but as we mentioned earlier, each new release of the SDK comes with increased support, functionality, and openness.

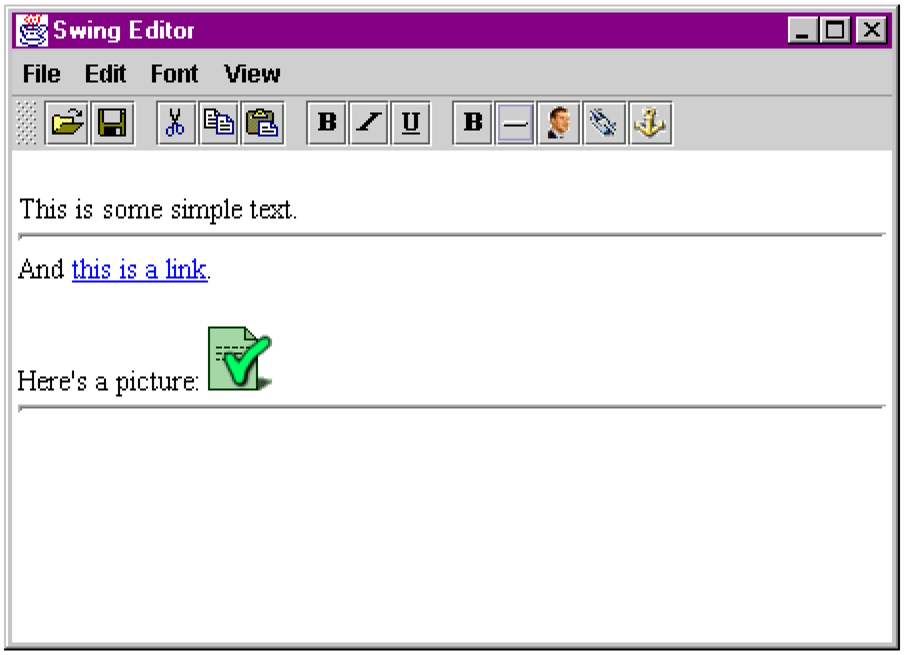
## *But I Only Want To Write HTML!*

If you're not interested in the hows and wheres of reading and writing HTML but still want to be able to edit HTML documents, go ahead and dive into this example. We extend the simple text editor from Chapter 23 of *Java Swing*, 2nd edition to support two things:

- HTML actions including horizontal rules, images, and hyperlinks

- A Save menu item writes the document as HTML

To get started, here’s a screenshot of our editor in action. The document you see was typed in from scratch.



This is some simple text.

And this is a link.

Here's a picture:

*Figure 2-1 The Swing Editor extended to support HTML.*

Here’s the HTML generated when you save the document:

```
<html>
  <head>

  </head>
  <body>
    <p>
      This is some simple text.
    </p>
    <hr>
    And <a href="http://www.ora.com">this is a link</a>.

Here's a picture:
<img src="http://the-moth.loyinc.pvt/images/correct.gif">
<hr>

  </body>
</html>
```

Not bad, eh? Not perfect, but certainly passable for prototypes and internal applications.

Let's look at the classes involved in this application. The editor is primarily the same as the `SimpleEditor` mentioned in Chapter 23 of our book. The primary change comes from the new actions we support. Rather than repeat the code for the editor (*HTMLEditor.java* in the source archive for these online supplements), we concentrate on three specific actions: `ImageAction`, `SaveAction`, and `TagAction`.

## Hyperlink Actions

The most interesting of the editor actions is the `TagAction` class. You can create actions for any HTML tag that uses one (primary) attribute. In our simple editor, the only such tag of interest is the `<A>` tag, but that is definitely a popular one. We create two separate instances of `TagAction` for the `<A>` tag: one for the `href` attribute version and one for the `name` attribute. Here's the code for creating the actions:

```
a = new TagAction(HTML.Tag.A, "URL", HTML.Attribute.HREF);
a.putValue(Action.SMALL_ICON, new ImageIcon("icons/link.gif"));
a.putValue(Action.NAME, "Anchor Link");
a = new TagAction(HTML.Tag.A, "Name", HTML.Attribute.NAME);
a.putValue(Action.SMALL_ICON, new ImageIcon("icons/anchor.gif"));
a.putValue(Action.NAME, "Anchor Name");
```

These actions can then be stuffed back into the action hash we use for all the predefined text actions found in the `StyledEditorKit`.

The code for the `TagAction` class itself is fairly straightforward. That's because we rely primarily on `StyledEditorKit.StyledTextAction` as a starting point.

```
public class TagAction extends StyledEditorKit.StyledTextAction {
  private HTML.Tag tag;
  private HTML.Attribute tagAttr;
  private String tagName;

  public TagAction(HTML.Tag t, String s, HTML.Attribute a) {
    super("anchor-link");
    tag = t;
    tagName = s;
    tagAttr = a;
  }

  public void actionPerformed(ActionEvent e) {
    JEditorPane editor = getEditor(e);
    // remember current selection to overcome focus bug in
    // JEditorPane
    if (editor != null) {
      int ss = editor.getSelectionStart();
      int se = editor.getSelectionEnd();
      String value = JOptionPane.showInputDialog(HTMLEditor.this,
                       "Enter " + tagName +":");
      editor.setSelectionStart(ss);
      editor.setSelectionEnd(se);
      StyledEditorKit kit = getStyledEditorKit(editor);
      MutableAttributeSet attr = kit.getInputAttributes();
      boolean anchor = attr.isDefined(tag);
      if (anchor) {
        attr.removeAttribute(tag);
      }
      else {
        SimpleAttributeSet as = new SimpleAttributeSet();
        as.addAttribute(tagAttr, value);
        attr.addAttribute(tag, as);
      }
      setCharacterAttributes(editor, attr, false);
    }
  }
}
```

This action starts by retrieving the editor (an instance of `JEditorPane`) associated with the event. "Associated with the event" means the editor pane with focus when the action occurred. There is no specific code required to attach an `Action` in a menu or toolbar to an editor. Once we have a valid editor, we use that to alter the selected text.

Anchor tags require at least one attribute, so we prompt the user for the proper information. (At the time of this writing, popping up and closing a `JOptionPane` removed the selection from the editor; the code before and after the prompt simply guarantees our selected text stays selected.) With the attribute information in hand, we use the editor's `StyledEditorKit` to make a proper `AttributeSet` for our tag. We use that `AttributeSet` in turn to apply our tag to the text in the editor.

## *Inserting Images*

The `ImageAction` class is similar to the `TagAction` class. We need to prompt the user for the image's source URL, but we don't "apply" the image to any selected text. We simply insert it at the current cursor location. Here's the code for the `ImageAction` class:

```
public class ImageAction extends StyledEditorKit.StyledTextAction {
  public ImageAction() {
    super("InsertIMG");
  }

  public void actionPerformed(ActionEvent ae) {
    JEditorPane editor = getEditor(ae);
    HTMLEditorKit kit = (HTMLEditorKit)editor.getEditorKit();
    HTMLDocument doc = (HTMLDocument)editor.getDocument();
    String value = JOptionPane.showInputDialog(HTMLEditor.this,
                       "Image file:");
    try {
      kit.insertHTML(doc, editor.getCaretPosition(),
               "<img src=\"" + value + "\">", 0, 0,
               HTML.Tag.IMG);
    }
    catch (Exception e) {
      JOptionPane.showMessageDialog(HTMLEditor.this,
      "Image Not Loaded", "ERROR", JOptionPane.ERROR_MESSAGE);
      e.printStackTrace();
    }
  }
}
```

Unlike the `TagAction` class, `ImageAction` requires the editor kit to be an `HTMLEditorKit`. The reason is that `HTMLEditorKit` has a convenience routine to insert a chunk of HTML as plain text. No building of tags with attribute sets here. We just insert the proper <`IMG`> tag with the recently retrieved source information and we're done.

## *Saving As HTML*

Ok, so we can insert HTML into our editor. You might have figured that out from the previous chapter. What we're really here for is reading and writing HTML. The new `SaveAction` shows off the ease of writing HTML from our editor. Here's the code:

```
class HTMLSaveAction extends AbstractAction {
  public HTMLSaveAction() {
    super("Save", new ImageIcon("icons/save.gif"));
  }

  // Query user for a filename and attempt to open and write the text
  // component's content to the file
  public void actionPerformed(ActionEvent ev) {
    String filename = (String) JOptionPane.showInputDialog(
      HTMLEditor.this, "Enter Filename", "Save As...",
           OptionPane.QUESTION_MESSAGE, null, null,
           currentFile == null ? "" : currentFile.getAbsolutePath());
    if (filename == null)
      return;

    FileWriter writer = null;
    try {
      writer = new FileWriter(filename);
      getTextComponent().write(writer);
    }
    catch (IOException ex) {
      JOptionPane.showMessageDialog(HTMLEditor.this,
        "File Not Saved", "ERROR", JOptionPane.ERROR_MESSAGE);
    }
    finally {
      if (writer != null) {
        try {
          writer.close();
        } catch (IOException x) {}
      }
    }
  }
}
```

The first part of this action simply prompts the user for a filename. As long as we get a non-null filename, we use that to create a standard FileWriter—nothing special yet. The special part actually comes from the JEditorPane. With the HTMLEditorKit installed, calling the usual write() method writes the text out using the HTMLWriter class. Very little code required on your part, just the way we like it.

## *Reading HTML*

Here's a diagram for the classes we examine in this section. For now we will tackle the input side of the HTMLEditorKit. Later sections examine the display and output classes.

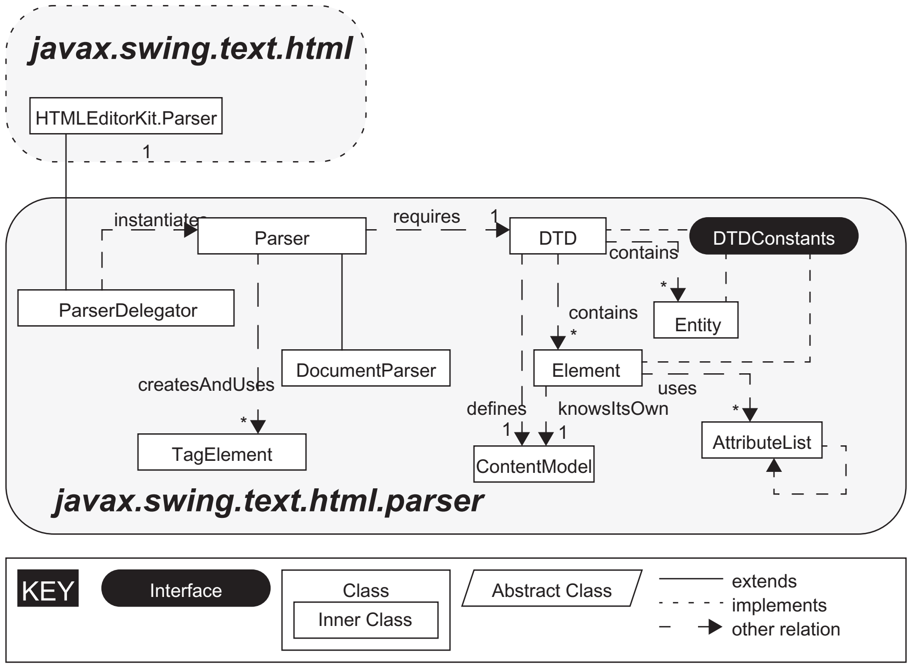


*Figure 2-2. HTMLEditorKit's parser class diagram.*

### *Document Parsers*

The first function involved in loading and displaying an HTML document is parsing it. The HTMLEditorKit class has hooks for returning a parser to do the job. The classes in the javax.swing.text.html.parser package implement a parser based on a Document Type Definition (DTD) for just this purpose. Since this parser is based on a DTD, it is extensible. However, we feel it is important once again to point you to the XML tools available for Java if you are serious about supplying custom markup.

## The Parser Class

The `Parser` class reads and parses an input stream. As noted before, it is DTD-based. By default, a parser with the HTML 3.2 DTD is built.

### Fields

Two fields appear in the `Parser` class. Neither field has a public accessor, but they are obviously accessible to subclasses.

*protected DTD dtd*
The DTD used for this parser. This field can be specified via the `Parser` constructor.

*protected boolean strict*
This flag determines whether the parser follows the DTD strictly. Since many HTML files do not perfectly adhere to the HTML DTD, the default value of false allows the parser to be lenient. The only way to change this value is to create a subclass.

### Properties

The properties for the `Parser` class listed in Table 2-1 are all protected. They provide access to basic state information in the parser.

*Table 2-1:* `Parser` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `attributes`[1] | `SimpleAttributeSet` | • | | | |
| `currentLine`[1] | `int` | • | | | 0 |
| `currentPos`[1] | `int` | • | | | 0 |

[1]This property has a protected accessor.

The `attributes` property contains a list of the parseable tags and their attributes. You can check for required attributes for a given tag through this property. The `current-Line` and `currentPos` properties disclose the line and character position in the stream being parsed. These values can be used for error messages or progress messages, for example.

### Constructors

Only one constructor is provided:

*public Parser(DTD dtd)*
The required DTD is used when the parser reads any input stream via the `read()` method. By default, the `HTMLEditorKit` builds a parser that uses the HTML 3.2

DTD. (The DTD is provided in the *rt.jar* file.) The `DTD` class is covered in detail later in this chapter.

## Public Parsing Methods

Two public methods exist for working with parsers:

*public void parse(Reader in) throws IOException*

This method reads and parses the input stream specified by `in`. It in turn calls the appropriate handler methods shown in the next section.

*public String parseDTDMarkup() throws IOException*

This method parses the DTD declaration (`<DOCTYPE>`) tags and returns the declaration. In SDK 1.3, valid declarations are simply ignored. Note that this method is not meant to produce the DTD declaration independently of parsing the document. The parser calls this method when it encounters declaration tags to ensure their validity. If the parser does not have an open document, calling this method results in an `IOException`.

## Protected Tag Methods

These protected methods accomplish the actual parsing tasks. The `parse()` method reads incoming tokens and calls one of these methods to handle the particular token at hand. Low-level parser support for new features could be added to subclasses. For example, you could override the `handleComment()` method to parse the contents of the comment for JavaScript or maybe proprietary revision information.

*protected TagElement makeTag(Element elem, boolean fictional)*
*protected TagElement makeTag(Element elem)*

These methods create new `TagElement` objects as tags are encountered in the input stream. The `elem` argument represents the element being tagged. The `fictional` argument is used when adding (implied) tags that do not really exist in the input stream. (For example, if you forget the `<HTML>` tag at the beginning of a document, a fictional `<HTML>` tag is inserted into the parsed document for you.) If you are creating a fictional `TagElement`, the `elem` argument comes directly from the DTD (see the `getElement()` methods of the `DTD` class later in this chapter). Otherwise, it is based on the content of the stream the parser is reading.

*protected void flushAttributes()*

This method clears the attributes in the parsing stack. For example, when starting a new tag, the attributes associated with the previously parsed tag are thrown out.

*protected void handleText(char[] text)*
*protected void handleTitle(char[] text)*
*protected void handleComment(char[] text)*

These methods handle standard text, title text, and comments, respectively. The text argument passed in comes from the input stream.

*protected void handleEOFInComment()*

This method handles the unexpected condition of reaching the end of the input stream while still in a comment. The default recovery mechanism is to treat only the first line of the comment as the real (and apparently unterminated) comment. Everything after that first line is passed back to the parser for reparsing.

*protected void handleEmptyTag(TagElement tag) throws ChangedCharSetException*
*protected void handleStartTag(TagElement tag)*
*protected void handleEndTag(TagElement tag)*

These methods handle the actual tags the parser encounters (empty tags, start tags, and end tags, respectively.) These tags can be actual or implied. Any document accounting information that goes along with each type of tag can be handled here, such as the attribute flushing for start tags mentioned in the `flushAttributes()` method.

*protected void handleError(int ln, String msg)*

This method deals with parsing errors. These errors are described by the text of `msg` and the line number (`ln`) in the input stream. This method can either try to recover from the error, or, more simply, it can notify the user that an error has occurred.

*protected void error(String err)*
*protected void error(String err, String arg1)*
*protected void error(String err, String arg1, String arg2)*
*protected void error(String err, String arg1, String arg2, String arg3)*

These methods all generate calls to the `handleError()` method with default information passed in for any unused arguments.

*protected void startTag(TagElement tag) throws ChangedCharSetException*
*protected void endTag(boolean omitted)*

These methods deal with the tag stack for new starting and ending tags. In turn, these methods call the `handleStartTag()` and `handleEndTag()` methods, respectively, to deal with the content of the particular tags. If the end tag was omitted from the original stream but is required by the DTD, an error is generated.

*protected void markFirstTime(Element elem)*

As its name implies, this method marks the first time an element has been seen in the current document. The only tags that the parser worries about by default are `<HTML>`, `<HEAD>`, and `<BODY>`.

*protected boolean parseMarkupDeclarations(StringBuffer strBuff) throws IOException*

This method serves as a wrapper for the `parseDTDMarkup()` method. If the text in `strBuff` is not DOCTYPE, `false` is returned. Otherwise, the `parseDTDMarkup()` method is called to make sure the declaration is syntactically correct.

## *The DocumentParser Class*

The `DocumentParser` class extends `Parser` and fills in several of the blank `handle()` methods. If you are looking to write your own parser, this class serves as a good starting point.

The `HTMLEditorKit` and `HTMLDocument` classes use `DocumentParser` to deal with HTML documents. In turn, this class parses an input stream and its `handle()` methods pass tokens to the appropriate callback object. (See the `HTMLEditorKit.ParserCallback` class in the previous chapter.)

### ***Constructors***

One public constructor exists:

*public DocumentParser(DTD dtd)*

This constructor creates a new `DocumentParser` object based on the supplied DTD. While you can technically supply any DTD, the documentation recommends using only the supplied HTML DTD. You can, of course, supply your own DTD to a subclass that expects such a DTD.

### ***Protected Parser Methods***

`DocumentParser` override some of the `Parser` class's `handle()` methods:

*protected void handleComment(char[] text)*
*protected void handleEmptyTag(TagElement tag) throws ChangedCharSetException*
*protected void handleEndTag(TagElement tag)*
*protected void handleError(int ln, String errorMsg)*
*protected void handleStartTag(TagElement tag)*
*protected void handleText(char[] data)*

The changes made to these methods in `DocumentParser` generally pass the incoming tag to the parser callback object. Some changes watch for tags such as <`META`> that might alter the remaining parsing.

### ***Public Parser Methods***

One public method is provided to start the parsing process:

*public void parse(Reader in, HTMLEditorKit.ParserCallback callback, boolean ignore-CharSet) throws IOException*

This method parses the text coming from `in` and sends tokens back to the `call-back` object. You can use the `ignoreCharSet` argument to, well, ignore the character set of the incoming stream. `HTMLDocument` objects ignore the character set by default. If you do not ignore it, a `ChangedCharSetException` is thrown if you encounter the `charset` meta tag. Of course, your `handle()` methods may choose to ignore the exception.

## The ParserDelegator Class

The last of the parser classes is the `ParserDelegator` class. It extends the `HTMLEd-itorKit.Parser` class and functions as a wrapper for the `DocumentParser` class. To install your own parser, you will probably subclass `ParserDelegator` (or create your own) and have your delegator point to your parser.

### Constructors

One public constructor exists:

*public ParserDelegator()*

This constructor builds a new `ParserDelegator` object and creates a DTD for HTML 3.2 through the `setDefaultDTD()` method if one has not already been instantiated.

### DTD Methods

Two static methods exist for creating a DTD to use with the parser:

*protected static void setDefaultDTD()*

This method sets up the HTML 3.2 DTD as the default and then uses `create-DTD()` to instantiate it. This default DTD is then used when creating new parsers. Notice that no arguments are passed to this method. Use of the HTML 3.2 DTD is hard-coded as of SDK 1.4, but that may change as this package matures.

*protected static DTD createDTD(DTD dtd, String name)*

This method fills in the given `dtd` from a resource file. This method assumes the resource can be found in the classpath and is a compiled DTD (*.bdtd*) file. The *bdtd* extension is appended to `name` to locate the appropriate resource.

### *Parse Methods*

*public void parse(Reader r, HTMLEditorKit.ParserCallback cb, boolean ignoreCharSet) throws IOException*

Not surprisingly, this method simply calls the `parse()` method with the same arguments as the delegated parser.

## *DTD-Based Parsing*

DTDs provide a way to describe the content of a document to a parser. DTDs are certainly not the only means, but they are very effective. You can create your own DTD and hand it over to the parser if you so choose. As we noted before, the new XML parsing tools may be more to your liking.

## *The DTD Class*

The starting point for building your own type definition comes in the form of the `DTD` class. (See Figure 2-2 for a diagram of how all the classes relate.) Instances of this class can be used to build parsers that understand markup documents. As you will see in the API, you can build a DTD from its more basic pieces of elements and entities or you can fill in a completed DTD from a stream. The HTML 3.2 DTD that the `HTMLEditorKit` class uses is *html32.bdtd* in the *classes.zip* file. Being in binary form, this DTD is not very user-friendly. If you want to see the real thing in human-readable form, check out *http://www.w3.org/TR/REC-html32.html*.

### *Properties*

The `DTD` class has one property shown in Table 2-2.

*Table 2-2:* `DTD` *properties*

| **Property** | **Data Type** | **get** | **is** | **set** | **Default Value** |
|---|---|---|---|---|---|
| `name` | `String` | • | | | `null` |

The `name` property is just what it suggests, the name of the DTD. The name for the DTD used for this editor kit is `"html32"`.

### *Fields*

Several fields defined for `DTD` are publicly available.

*public String name*

The name of the DTD (storing the value from the `name` property).

*public Vector elements*
*public Hashtable elementHash*
*public Hashtable entityHash*

These fields store the elements and entities comprising the DTD. The elements can either be accessed by name from the `elementHash` or as an entire list from the elements vector.

*public final Element applet*
*public final Element base*
*public final Element body*
*public final Element head*
*public final Element html*
*public final Element isindex*
*public final Element meta*
*public final Element p*
*public final Element param*
*public final Element pcdata*
*public final Element title*

These fields all represent `Element` objects that can be used to type information being parsed. For example, you can check to see if a particular element is equal to the `pcdata` field of your DTD.

*public static int FILE_VERSION*

This static field contains the version of the `DTD` class. The value in this field is used to verify incoming DTDs using the `read()` method described below. Currently, no penalty is incurred for providing alternate version numbers. (The current value is 1.)

## *Constructors*

One protected constructor exists:

*protected DTD(String name)*

This creates a new instance of the `DTD` class named `name`. The constructor adds a few general entities for white space and one “unknown” empty element as seeds for the DTD.

## *Element and Entity Methods*

A DTD consists of definitions for entities and elements. An entity represents a single piece of displayed information. A common example of an entity in the HTML DTD is the `&amp;` text. This entity produces an ampersand (&) when the document is displayed. Elements represent the structure and to some extent, the content, of the document. For example, this paragraph of text could be displayed in an HTML document `<P>`...`</P>` element.

*public Entity getEntity(String name)*
*public Entity getEntity(int ch)*

These methods return `Entity` objects based on the given `name` or character (`ch`), respectively. Note that these methods work only for previously defined entities. You can add entities to the DTD using one of the `define()` methods below.

*public Element getElement(String name)*
*public Element getElement(int index)*

These methods return `Element` objects based on the given `name` or `index`, respectively. Note that the second method works only for previously defined elements. An exception is thrown if you try to access an index outside the array. For the first method, if the named element does not exist, a new, empty `Element` is created and added to the hash (and the vector). This new element is returned.

*public Entity defineEntity(String name, int type, char[] data)*
*public Entity defEntity(String name, int type, int ch)*
*protected Entity defEntity(String name, int type, String str)*

These methods define entities for use in the DTD. The first version of the method checks to see if the entity given by `name` is already defined. If not, a new entity is created with the given `name`, `type`, and `data`. (For more information on valid values for `type` and `data`, see the `Entity` class later in this section.) The entity is added to the `entityHash` and also returned. The other versions simply build proper `char` arrays from the given character (`ch`) or string (`str`) and call the first version.

*public Element defineElement(String name, int type, boolean omitStart, boolean omitEnd, ContentModel content, BitSet exclusions, BitSet inclusions, AttributeList atts)*
*protected Element defElement(String name, int type, boolean omitStart, boolean omitEnd, ContentModel content, String[] exclusions, String[] inclusions, AttributeList atts)*

These methods define new elements for use in the DTD. These methods find the element with the given name and then assign all the attributes specified in the remaining arguments to that element. Note that this means you can overwrite the values given to previously defined elements. (For more information on the valid values for the remaining arguments, see the `Element` class later in this section.)

*public void defineAttributes(String name, AttributeList atts)*

This method assigns the attribute list `atts` to the element `name`. At the time of this writing, the assignment is simple; `atts` overwrites any attribute list previously associated with the element.

*protected AttributeList defAttributeList(String name, int type, int modifier, String value, String values, AttributeList atts)*

This method builds a new attribute list based on a (possibly) existing list and a specified new value given by a `name`, `type`, `modifier` and either a single value or a list of values. For more information on the contents of an attribute list, see the `AttributeList` class later in this section.

*protected ContentModel defContentModel(int type, Object obj, ContentModel next)*
Like the `defAttributeList()` method, this method builds on a (possibly) existing content model. The new content is described by `type` and `obj`. The next argument can be `null` if this is the first content information built. For more information on content models, see the `ContentModel` class later in this section.)

### *Public DTD Methods*

If you create several DTDs for use in your applications, these methods can make it easy to access those DTDs:

*public static void putDTDHash(String name, DTD dtd)*
This method adds a DTD (`dtd`) to a hash for quick retrieval by name.

*public static DTD getDTD(String name) throws IOException*
This method retrieves the DTD described by `name` from the DTD hash. If no DTD with the given `name` exists, a new one is created, added to the hash, and then returned. This is the preferred method of creating your own DTD if you don't intend to create a subclass.

*public void read(DataInputStream in) throws IOException*
This method fills in a DTD from the given data stream (`in`). The data read is assumed to be a well-formed *.bdtd* file (i.e., a correct DTD properly compiled to the bdtd format). The general format of the file contains the list of names (entities, attributes and elements) followed by the structure of the document.

## *The DTDConstants Interface*

As its name implies, the `DTDConstants` interface supplies you with several constants for use in building and describing a DTD (see Table 2-3). Most classes in the parser package implement this interface.

*Table 2-3. DTDConstants*

| | | |
|---|---|---|
| ANY | IDREF | NUMBERS |
| CDATA | IDREFS | NUTOKEN |
| CONREF | IMPLIED | NUTOKENS |
| CURRENT | MD | PARAMETER |
| DEFAULT | MODEL | PI |
| EMPTY | MS | PUBLIC |
| ENDTAG | NAME | RCDATA |
| ENTITIES | NAMES | REQUIRED |
| ENTITY | NMTOKEN | SDATA |

*Table 2-3. DTDConstants*

| | | |
|---|---|---|
| FIXED | NMTOKENS | STARTTAG |
| GENERAL | NOTATION | SYSTEM |
| ID | NUMBER | |

These constants are used primarily as valid types for entities, elements, and content models.

## *The AttributeList Class*

This class defines a package-specific list of attributes that can be attached to an `Element`. The name is a bit misleading since an `AttributeList` object is really one item in a linked list of attributes. The `next` property (shown below) points you to the next attribute in the list. You can attach an `AttributeList` to an `Element` using the `defineAttributes()` method of the `DTD` class.

### ***Fields***

Several public fields are available to give direct access to many of `AttributeList`'s properties. You are, of course, encouraged to use the public accessor methods.

*public int modifier*
: The modifier for this attribute. Valid modifiers (from the `DTDConstants` interface) include REQUIRED and IMPLIED.

*public String name*
: The name of the attribute.

*public AttributeList next*
: The next attribute in the list. The last attribute simply has `null` for this field.

*public int type*
: The type of this attribute. Common values include CDATA and NUMBER.

*public String value*
: The value (if any) stored in this attribute. The value of an attribute might include the name of a required attribute in a tag, for example.

*public Vector values*
: If an attribute stores multiple values (say, a list of possible valid types for a given element in the DTD), those values are stored here.

### *Properties*

Without exception, the properties of `AttributeList` (shown in Table 2-4) give you access to the fields described above through the more conventional `getProperty()` mechanism.

*Table 2-4:* `AttributeList` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `modifier` | `int` | • | | | 0 (no modifier) |
| `name` | `String` | • | | | from constructor |
| `next` | `AttributeList` | • | | | null |
| `type` | `int` | • | | | 0 (illegal type) |
| `value` | `String` | • | | | null |
| `values` | `Enumeration` | • | | | null |

Valid type and modifier values come from the `DTDConstants` interface. The values property can be `null` if no set of multiple values is defined for the attribute. Remember that although only the `get()` methods are defined for these properties, they are maintained in the class by public variables and can be modified if need be.

### *Constructors*

You can create your own attribute list starting with these constructors:

*public AttributeList(String name)*
*public AttributeList(String name, int type, int modifier, String value, Vector values, AttributeList next)*

These constructors create new `AttributeList` objects. The first constructor creates an empty attribute that you can fill by directly changing the values of its fields. You can also specify all the relevant pieces of information to the second constructor. Both `value` and `values` can be passed as `null`. Recall that this creates one link in a chain. To build a complete list, provide a valid reference in the `next` argument to point to the next attribute.

### *Conversion Methods*

Two class methods are provided as convenience routines to convert attribute types to human-readable names and vice-versa:

*public static int name2type(String nm)*
*public static String type2name(int tp)*

The `name2type()` method converts a name found in a DTD to its corresponding `DTDConstants` value. `CDATA` is returned for unknown or nonexistent names. The `type2name()` method performs the reverse lookup. Here, `null` is returned if the type in `tp` is not defined in the DTD.

## *The Element Class*

In a DTD, an `Element` is the main building block of the document structure. (This is separate from the notion of an element in the document content of the Swing `EditorKit` world.) They describe the main sections of a document. How those sections relate to each other is determined by the placement of the element in the DTD. If you build a DTD from scratch, you'll be building several `Element` objects along the way.

### *Field Detail*

The fields for `Element` are all public. Several of the fields do have public accessors and show up as properties for the class, but none of the properties have a `set()` method. With only a default constructor for the class, you'll have to set these values manually if you create a custom `Element`.

*public AttributeList atts*
The attributes associated with the element, such as whether the element is a required element or an implied element.

*public ContentModel content*
The content of the element represented as a restricted (deterministic) Backus-Naur Form (BNF) expression. (BNF is often used to describe computer languages.)

*public Object data*
The user data for the element.

*public int index*
The location in the overall document of this element.

*public BitSet inclusions*
*public BitSet exclusions*
The inclusion and exclusion lists for the element. These lists contain indexed references to other elements that can be included or should be disallowed while parsing.

*public String name*
The name of the element.

*public boolean oStart*
*public boolean oEnd*
Whether the parser allows the start or end tag to be omitted, respectively.

*public int type*
The type of the element. Valid types come from the `DTDConstants` interface and include RCDATA, EMPTY, MODEL and ANY.

### *Properties*

Almost all the properties listed in Table 2-5 are publicly accessible fields listed above. The exception, of course, is the `empty` property. This property indicates that the `type` of the element is EMPTY.

*Table 2-5:* `Element` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `name` | `String` | • | | | |
| `type` | `int` | • | | | |
| `content` | `ContentModel` | • | | | |
| `attributes` | `AttributeList` | • | | | |
| `index` | `int` | • | | | |
| `empty` | `boolean` | | • | | |

### *Attribute Methods*

These methods retrieve field values without explicitly referencing the fields.

*public AttributeList getAttribute(String name)*
*public AttributeList getAttributeByValue(String name)*

These methods return an `AttributeList` that starts at the attribute specified by `name`. The second version returns the attribute containing `name` as its value.

*public boolean omitStart()*
*public boolean omitEnd()*

These methods indicate whether the start and end tag for the element may be omitted. These directly reflect the contents of the `oStart` and `oEnd` fields.

### *Conversion Methods*

As with other classes in this package, a convenience routine for converting human-readable versions of type information exists:

*public static int name2type(String nm)*

This method returns a type from `DTDConstants` given a valid name `nm`. You can look up CDATA, RCDATA, EMPTY, and ANY by name. An invalid name returns `0`.

## *The TagElement Class*

While an `Element` object is the basis for a generic DTD, HTML documents need a little more information to be displayed correctly onscreen. For example, the `<BR>` tag indicates that the next piece of text should start on a new line. To make this type of information easily accessible, the `HTMLEditorKit` parser package uses the `TagElement` class to wrap regular `Element` objects. Note that `TagElement` does not extend `Element`, but a `TagElement` does keep an `Element` at its core.

## Properties

Table 2-6 shows the properties associated with the `TagElement` class.

*Table 2-6:* `TagElement` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `element` | `Element` | • | | | from constructor |
| `HTMLTag` | `HTML.Tag` | • | | | derived from element; `UnknownTag` if no tag can be found for element |
| `preformatted` | `boolean` | | • | | `false` |

The `element` property provides access to the real `Element`. The `HTMLTag` property relays the HTML tag associated with the element. For example, an `img` element would return `HTML.Tag.IMG` for this property. The `preformatted` property determines whether the whitespace in the element's text should be left intact. In HTML, text enclosed in `<PRE></PRE>` tags would fall into this category.

## Constructor Detail

You can create a new `TagElement` with one of two constructors:

*public TagElement(Element elem)*
*public TagElement(Element elem, boolean fictional)*

Both constructors build a `TagElement` with the default property values shown in Table 2-6. The second constructor allows you to supply a boolean value to indicate whether this tag is fictional. You might insert fictional tags to maintain internal document integrity when an optional tag is left out of the original source. For example, a paragraph of text might be enclosed in a fictional `<P>` element. The haphazard use of `<P>` tags in real-world HTML documents makes it important that a parser allow a single paragraph of text, but the internal representation of that text should still be consistent with the structure of the document according to the DTD.

## Miscellaneous Test Methods

Two other methods let you check the properties of the `TagElement` class, although they do not use the standard `get()` or `is()` accessor names:

*public boolean breaksFlow()*

This method determines whether this element breaks the flow of text and images. Elements that break the flow of text include `<BR>` and `<HR>`.

*public boolean fictional()*

This method indicates the fictional status of the element. The default is `false`. Using the second of the constructors presented above allows you to override the default.

## *The Entity Class*

Entities serve as the macros of the DTD. More specifically, parameter entities can be thought of as macros for use in a DTD. Formally, an entity is "a collection of characters that can be referenced as a unit" (ISO 8879: §4.120). Apart from parameter entities, DTDs can define general entities that show up in the markup content. A common example of such an entity is the string `&amp;`, which produces an ampersand (&) in the document. Certainly the general entities are more familiar, but if you write DTDs,* parameter entities can be very useful.

### ***Field***

As with the `Element` class, all of the `Entity` fields are public. To modify their values, access these fields directly. Unlike `Element`, you can specify all the field values directly to the constructor if you like.

*public String name*

The name of the entity.

*public int type*

The type of the entity. Valid types come from the `DTDConstants` interface and include PUBLIC, SDATA, PI, STARTTAG, ENDTAG, MS, MD, and SYSTEM.

*public char[] data*

The actual characters that make up the entity. For an ampersand, this field would include an array of characters like this: { 'a', 'm', 'p' }.

### ***Properties***

All the fields in `Entity` have public accessor methods, as listed in Table 2-7. However, in addition to the fields, a few other properties are defined.

* If you want more information on DTDs or SGML in general, you should check out *README.1ST* by Turner, Douglass, and Turner (Prentice-Hall PTR, 1996). A good reference for DTDs and XML is *XML: The Annotated Specification* by Bob DuCharme (Prentice Hall, 1999).

*Table 2-7:* `Entity` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `name` | `String` | • | | | from constructor |
| `type` | `int` | • | | | from constructor |
| `general` | `boolean` | | • | | derived from `type` |
| `parameter` | `boolean` | | • | | derived from `type` |
| `data` | `char[]` | • | | | from constructor |
| `string` | `String` | • | | | derived from `data` |

The `general` and `parameter` boolean properties are based on the type of the entity. They indicate whether any given entity is a general entity or a parameter entity, respectively. The `string` property serves as an alternate mechanism for looking at the `data` property.

### *Constructors*

One constructor exists:

*public Entity(String name, int type, char[] data)*
This constructor creates an `Entity` object with the given values for its `name`, `type`, and `data` properties.

### *Conversion Method*

As with the `Element` class, we have a conversion method available:

*public static int name2type(String nm)*
This method returns the `DTDConstants` equivalent of the given entity name (`nm`).

## *The ContentModel Class*

The last piece of the DTD puzzle is the `ContentModel` class. In building a DTD, you can specify that certain elements are required, are optional, must occur at least once, or can occur multiple times. If you have worked with regular expressions at all, these options should sound familiar. The `ContentModel` class serves as a tiny regular expression encapsulator for use in a DTD. With a `ContentModel` object, you can say things like `<P>` tags are allowed multiple times in an HTML document. In such a case, the content of the content model would be the `P` element and the type of the model would be `'*'`.

## *Fields*

As with the other DTD building blocks, `ContentModel` fields are public. For this class, however, no direct `get()` style methods are provided. You access elements and content models directly.

*public Object content*
The content of this model (an `Element` or a nested `ContentModel`).

*public ContentModel next*
In list models, this field points to the next piece of the complete `ContentModel`. In other models, this field is `null`.

*public int type*
The type of the model. Possible types include characters you might recognize from the world of regular expressions: '*', '?', '+', ',', '|', '&'. A single element model (one required occurrence) is represented by the value `0`.

## *Constructors*

The constructors for the `ContentModel` class are quite flexible:

*public ContentModel()*
Build an empty content model. You must fill in the fields by hand.

*public ContentModel(Element content)*
Build a single element content model, with a type of `0`.

*public ContentModel(int type, ContentModel content)*
Build a specific content model (`type`) with the given `content`.

*public ContentModel(int type, Object content, ContentModel next)*
Build a specific list model (`type`) with the given `content` and next piece of the model.

## *Methods*

The methods of the `ContentModel` class answer a few questions about the type of the model and its content. A parser can use these methods to handle an incoming stream of text more efficiently.

*public boolean empty()*
This method returns `true` if the model could match an empty input stream. For example, a P* model would return true, but a [0-9]+ model would return false.

*public Element first()*
This method returns the element that must come next in an input stream. The name might seem strange, but think of it as the first correct element of the stream.

*public boolean first(Object token)*

This method returns `true` if this model could be the first element (token) in an input stream.

*public void getElements(Vector elemVec)*

This method appends the elements of this model to `elemVec`. List models are traversed recursively.

## *Displaying the Document*

When displaying HTML, the `HTMLEditorKit` takes full advantage of the Swing text facilities. Like the display of other text documents from the `javax.swing.text` package, HTML display goes through a `ViewFactory`. The specific view factory used is the `HTMLViewFactory`, an inner class of `HTMLEditorKit`. In addition to using some views defined by the main text package, the HTML package defines several more.

Here's a class diagram for the view classes. Note that some of the package private classes mentioned in the `ViewFactory` discussion are not shown. Obviously they cannot be used except as source examples for your own programming efforts. (But they are great for that, if you get in the business of writing custom views.)

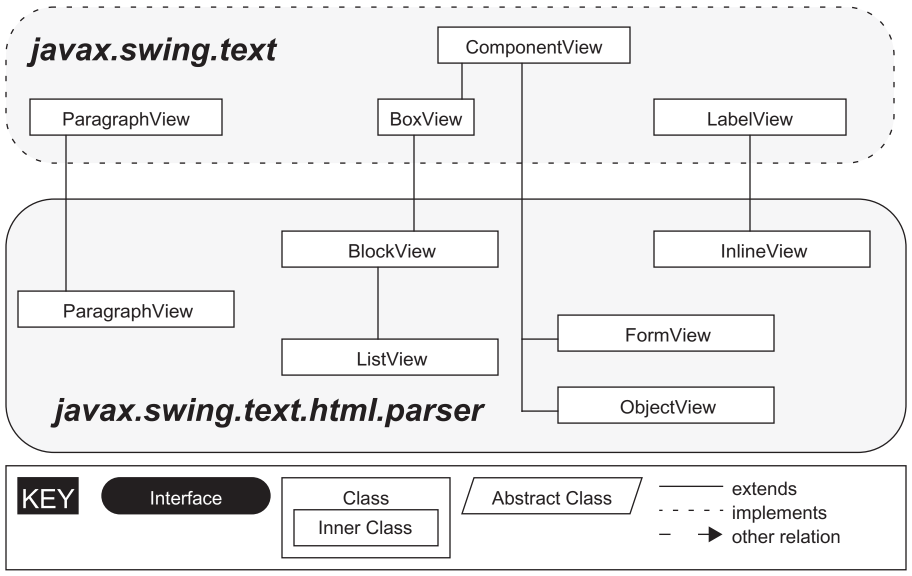


*Figure 2-3. The class diagram for HTML views.*

## *Basic Views for HTML*

HTML has types of paragraphs and elements not associated with normal text. There are several mundane topics like offset blocks of text and lists. There are also several fancier topics like forms and frames. `BlockView`, `InlineView`, `ListView`, `ObjectView`, and `ParagraphView` comprise the core set of basic HTML views. You can see their hierarchical relationship in Figure 2-3. In the descriptions of the classes, we present only the new methods associated with each class. The basic function of these views does not differ from that of the `View` class in the `javax.swing.text` package.

You can extend any of these views to create a custom view for your own documents. The mechanism for getting your views into play is shown in the "Internals Example" section of the HTML Editor Kit chapter. While that example creates only a "hidden" view, a valid bounding size and a reasonable `paint()` method are all that is required to render a real visual presence in the document.

You will notice that several of the views described below do not have any methods. Most of these views simply override the common methods for a generic view to provide the appropriate size and visual effect.

## *BlockView*

This straightforward class implements a view for a block of text where the attributes of the text are determined by the stylesheet in place for the parent HTML page.

### ***Constructors***

One public constructor for `BlockView` exists:

*public BlockView(Element elem, int axis)*

This constructor creates a `BlockView` for the given element (elem) aligned along the given axis. The valid values for axis are `View.X_AXIS` and `View.Y_AXIS`. The `axis` plays a role in determining the actual size of the view when modified by things like stylesheets. For example, `X_AXIS` blocks are affected by the left and right insets while `Y_AXIS` blocks consider top and bottom insets.

### ***Block-Specific Methods***

These protected methods are available:

*protected void setPropertiesFromAttributes()*

This method sets the display properties from the stylesheet for the document.

*protected StyleSheet getStyleSheet()*

This method returns the stylesheet from the enclosing `HTMLDocument`.

## *InlineView*

This view handles most of the real stuff you see on your screen. Technically, this is the view used for "content" elements in HTML. If you run the debugging version of the `HTMLEditorKit` from the previous chapter, you'll notice that most of the elements built when parsing the documents are tagged as `HTML.Tag.CONTENT` objects.

### ***Constructors***

As with most other views, a simple constructor is provided:

*public InlineView(Element elem)*
  This constructor creates a new `InlineView` for the given element (`elem`).

### ***Methods***

One protected method is introduced in this class:

*protected StyleSheet getStyleSheet()*
  As with the HTML views, you can use this method to retrieve the stylesheet in effect for the view.

## *ListView*

This view handles HTML lists—both ordered and unordered. The entire list is rendered in one shot using the `StyleSheet.ListPainte`r inner class to do the actual painting.

### ***Constructors***

One public constructor exists:

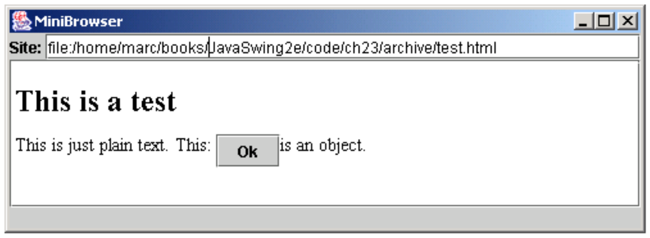
*public ListView(Element elem)*
  This constructor builds a list view for the given element (`elem`). The supplied element should be the main list tag (for example, a `<UL>` or `<OL>` tag), not an individual list item.

## *ObjectView*

This view handles the `<OBJECT>` tags. In SDK 1.4, support for this tag is still limited but you can build very simple objects as illustrated by the following code:

```
<object classid="javax.swing.JButton">
  <param name="text" value="OK">
</object>
```

Figure 2-4 shows the result of that snippet of HTML in our MiniBrowser application.


MiniBrowser
Site: file:/home/marc/books/JavaSwing2e/code/ch23/archive/test.html
Ok

This is a test

This is just plain text. This: is an object.

*Figure 2-4. A JButton rendered in an HTML document via the <OBJECT> tag.*

### Constructors

Again, one constructor is provided:

*public ObjectView(Element elem)*

This constructor builds an `ObjectView` and encapsulates the element elem with all the attributes for the element specified by the `<PARAM>` tags in the HTML document.

## ParagraphView

Last, but certainly not least, is the `ParagraphView` class. Text from an HTML document, either from explicit `<P>` tags or from implied tags, is presented on screen via this view.

### Constructors

One constructor exists:

*public ParagraphView(Element elem)*

This constructor creates a view for the given element (`elem`). The element should be a complete paragraph of content.

### Methods

Only one new method is introduced:

*protected StyleSheet getStyleSheet()*

As with the other HTML views, you can use this method to retrieve the stylesheet in effect for this view.

## *Package Private Views*

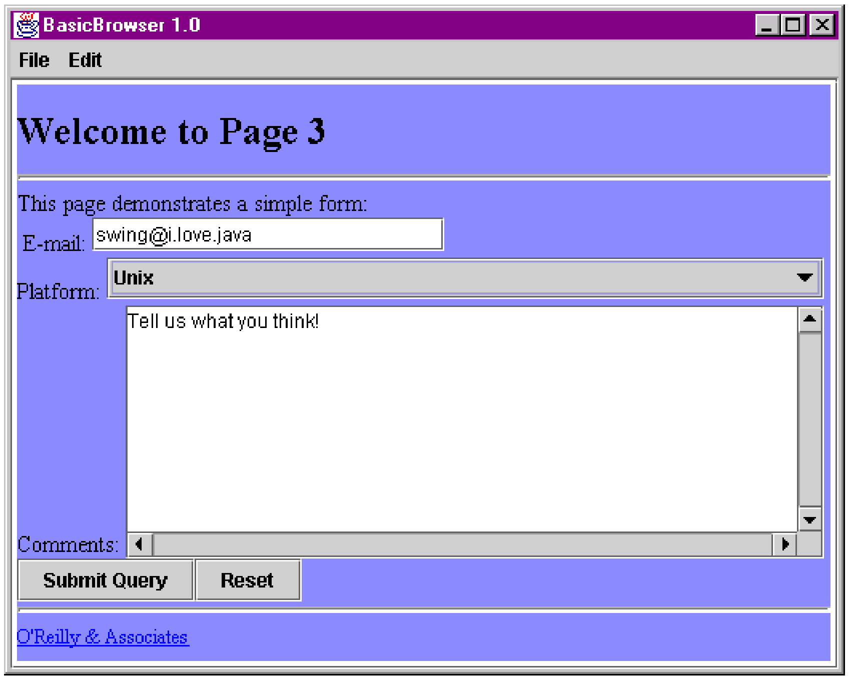
Table 2-8 lists other views used by the Swing `ViewFactory` to render HTML. These views are really package private or “friendly” classes so you do not have access to them as a developer. However, you might find the list useful as a reference if you plan to implement your own markup renderer.

*Table 2-8 Package Private HTML View Descriptions*

| View | Description |
|---|---|
| `BRView` | View for the `<BR>` tag |
| `CommentView` | View for HTML comments `<!-- -->` |
| `HRuleView` | View for the horizontal rule (`<HR>`) tag |
| `ImageView` | View for inline images |
| `IsindexView` | View for `<ISINDEX>` tag (a `JPanel` with a prompt and a `JTextField`) |
| `LineView` | View for a single line of text (used for text inside of `<PRE>` tags) |
| `TableView` | View for tables (with inner classes for table rows and cells) |
| `HiddenTagView` | View for hidden tags (subclass of `EditableView` so that you can edit the tags when your `JEditor-Pane` is set to editable) |

## *Form Views*

HTML forms allow users to enter information on a web page. The `HTMLEditorKit` has a few views devoted to producing the components and functionality you would expect from a standard web form. To display these components in a browser, you do not need to do anything special. The editor kit simply sets up the components based on the HTML page you give it. Figure 2-5 shows an example:



Welcome to Page 3

This page demonstrates a simple form:

E-mail: swing@i.love.java

Platform: Unix

Comments: Tell us what you think!

*Figure 2-5. An example of the built-in support for form <INPUT> components.*

## *FormView*

`FormView` is the main class for building forms. As an extension of the `ComponentView` class, it can produce the views for real components based on the input type. Table 2-9 lists the common form elements and the components produced by `FormView`.

*Table 2-9 FormView HTML To Component Mappings*

| **HTML Form Tag** | **Component created** |
|---|---|
| `<input type=button>` | `JButton` |
| `<input type=checkbox>` | `JCheckBox` |
| `<input type=file>` | `JTextField` |
| `<input type=image>` | `JButton` |
| `<input type=password>` | `JPasswordField` |
| `<input type=radio>` | `JRadioButton` |
| `<input type=reset>` | `JButton` |
| `<input type=submit>` | `JRadioButton` |
| `<input type=text>` | `JTextField` |

*Table 2-9 FormView HTML To Component Mappings*

| HTML Form Tag | Component created |
|---|---|
| `<select size=1> or <select>` | `JComboBox` |
| `<select size=">1"> or <select multiple>` | `JList (inside a JScrollPane)` |
| `<textarea>` | `JTextArea (inside a JScrollPane)` |

## *Inner Classes*

`FormView` has one inner class defined:

*protected class FormView.MouseEventListener extends MouseAdapter*

This inner class serves as a listener for mouse click events over input images. Even though images are presented via the `JButton` component (which could do a normal submission for simple clicks just like text-based buttons), this listener appends the mouse click coordinate information to the submission.

## *Submit and Reset Text Constants*

Two constants are defined in `FormView`. Both constants are deprecated and have been replaced by UI manager properties (`FormView.submitButtonText` and `FormView.resetButtonText`, respectively).

*public static final String SUBMIT*
*public static final String RESET*

Names for the text messages to be displayed in the default submit and reset form buttons.

To set these values properly through the UI manager, use code similar to this:

```
UIManager.put("FormView.submitButtonText", "Ok");
UIManager.put("FormView.resetButtonText", "Clear");
```

## *Constructor Detail*

As with other views, `FormView` has a single, simple constructor:

*public FormView(Element elem)*

This constructor creates a new `FormView` from the given element (`elem`). The form is built all at once after finding the closing `</FORM>` tag in the HTML.

## *Methods*

SDK 1.3 defined a few additional methods for the `FormView` class.

*public void actionPerformed(ActionEvent evt)*

This event handler method is called in response to events from the form. For example, events from a submit or reset button are processed here. This method also handles submitting the form from a text field—if the text field is the last field in the form (otherwise, the focus is simply sent to the next text field).

*protected void imageSubmit(String imageData)*
*protected void submitData(String data)*

These methods submit the form. The `imageSubmit()` method handles the extra mouse click coordinate information and then calls `submitData()`.

## *Option*

This class handles the option elements available for use with lists and combo box elements.

### *Properties*

The properties shown in Table 2-10 represent the display attributes common to `<FORM>` option elements.

*Table 2-10:* `Option` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `attributes` | `AttributeSet` | • | | | from constructor |
| `label` | `String` | • | | • | `null` |
| `selected` | `boolean` | | • | | `false` |
| `selection` | `boolean` | | | • | `false` |
| `value` | `String` | • | | | derived from `attributes` (specifically, the value attribute) |

The `attributes` property returns the HTML attribute set for this option element. You can get the `LABEL` attribute directly from the `label` property here. The information in the element can be retrieved from the `value` or `selected` properties depending on the type of input element. The `selection` property allows you to set the selected state of the option.

### *Constructors*

`Option` has one public constructor:

*public Option(AttributeSet attr)*

This constructor builds a new `Option` object with the given attributes (`attr`). The type of input element built depends on the attribute values.

## Package Private Views

As with the other views used with HTML, several package private views help with forms. Table 2-11 lists these classes and their uses.

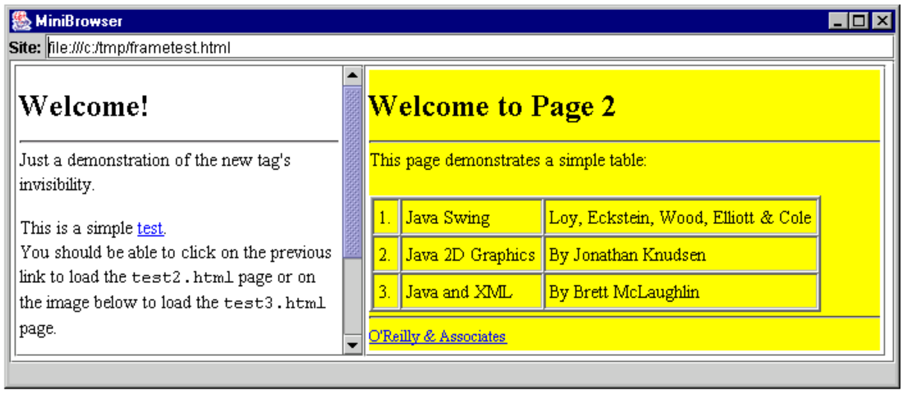
*Table 2-11 Package Private Form Views*

| View Class | Description |
|---|---|
| `EditableView` | A view that is visible only when the enclosing text component is editable. |
| `OptionComboBoxModel` | Model for <OPTION> with one item visible (combobox) |
| `OptionListModel` | Model for <OPTION> with multiple items visible (list) |
| `TextAreaDocument` | Model for text area component |

## Document Frames

Frames are a popular HTML design tool. They provide a relatively simple mechanism for keeping one area of a page static while allowing dynamic content in other areas. One effective example of frames is a site "toolbar" that provides the user with quick access links to the entire site at all times. SDK 1.3 and later provide basic support for HTML frames, although experience suggests that you should use them lightly. Figure 2-6 shows a simple frame demonstration using our simple HTML browser from the previous chapter. There is no new Swing code to present; the only thing interesting about this example is the HTML document that uses the standard `<FRAME>` tag. The `HTML-EditorKit` builds the frames for you.

We discuss handling hyperlink events and frames with our minibrowser in the next section.



Welcome!

Just a demonstration of the new tag's invisibility.

This is a simple test.
You should be able to click on the previous link to load the test2.html page or on the image below to load the test3.html page.

Welcome to Page 2

This page demonstrates a simple table:

| | | |
|---|---|---|
| 1. | Java Swing | Loy, Eckstein, Wood, Elliott & Cole |
| 2. | Java 2D Graphics | By Jonathan Knudsen |
| 3. | Java and XML | By Brett McLaughlin |

O'Reilly & Associates

*Figure 2-6. A demonstration of <FRAME> tag support in the HTMLEditorKit.*

### Package Private Views

The views that support frame functionality in the `HTMLEditorKit` are package private. It may still be useful for you to see the views listed in Table 2-12 in case you decide to enhance the frame functionality.

*Table 2-12 Package Private HTML Frame Views*

| View Class | Description |
|---|---|
| `FrameView` | Handles one frame (borders, margins, scrollbars, etc.) |
| `FrameSetView` | Handles setting up the frames |
| `NoFramesView` | Handles the <NOFRAMES> tag by ignoring the content unless the editor pane displaying the document is editable. |

## HTMLFrameHyperlinkEvent

Most of the effort work involved to implement frame comes in writing the HTML, not the Java code. One area that can affect your Java code is hyperlinks that are intended to display in a separate frame. To support this variation of a hyperlink, Swing includes an extension of the normal `HyperlinkEvent` called `HTMLFrameHyperlinkEvent`.

### Properties

HTMLFrameHyperlinkEvent supports the properties listed in Table 2-13.

*Table 2-13:* `HTMLFrameHyperlinkEvent` *properties*

| Property | Data Type | get | is | set | Default Value |
|---|---|---|---|---|---|
| `sourceElement` | `Element` | • | | | null |
| `target` | `String` | • | | | null |

The `sourceElement` property (when present) refers to the anchor element from the HTML document which caused the event. You can use this property to grab other interesting facts from the `<A>` tag, such as possible custom JavaScript information. The target property returns the target frame for the event. As of SDK 1.3, the following targets are handled the way you would expect:

`_self`
: The page referenced by the HREF attribute should appear in this frame.

`_top`
: The page referenced by the HREF attribute should appear in the top-level frame of this browser window.

`myframe`
: The page referenced by the HREF attribute should appear in the frame named "myframe." Any number of named frames can be created and used.

One common target, `_blank`, does not work. Since this target is meant to create a new browser window (which is really a `JFrame`-level component), the `HTMLEditorKit` would have to guess at what such a window looked like. However, if you design your browser to be a popup window with frame closing and visibility controls properly defined, you can watch for the `_blank` target and handle it yourself. Here's a modified version of the `hyperlinkUpdate()` method in the `SimpleLinkListener1` class that handles frame events properly—including `_blank` targets:

```
public void hyperlinkUpdate(HyperlinkEvent he) {
  HyperlinkEvent.EventType type = he.getEventType();
  String target = null;

  // Decide which event we got...
  if (he instanceof HTMLFrameHyperlinkEvent) {
    target = ((HTMLFrameHyperlinkEvent)he).getTarget();
    // Pop up a quick debugging confirmation message...
    JOptionPane.showMessageDialog(null, "Frame Event for " + target);
  }
  // Get the url, and if it's not null, switch to that
  // page in the correct pane and update the "site url" label.
  if (he instanceof HTMLFrameHyperlinkEvent) {
    if (target.equals("_blank")) {
      new MiniBrowser2(he.getURL().toString()).setVisible(true);
      return;
    }
    HTMLFrameHyperlinkEvent  evt = (HTMLFrameHyperlinkEvent)he;
    HTMLDocument doc = (HTMLDocument)pane.getDocument();
    doc.processHTMLFrameHyperlinkEvent(evt);
  } else {
    // Ok, it's just a normal event, go ahead and deal with it
    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();
    }
  }
}
```

You can see in this example how to use the `HTMLFrameHyperlinkEvent` class. We can retrieve the target directly from the event. If it is `_blank`, we pass the URL to the constructor of the `MiniBrowser2` class and create a new instance. If it is any other target, we ask the current document to handle the event. Even if the target is for a separate pane, the `doc.processHTMLFrameHyperlinkEvent()` call passes the URL to the proper frame. If it is not a frame event, we handle the event as we did before we started talking about frames.

Notice that this example is lacking one tiny feature: the default window listener for `MiniBrowser2` calls `System.exit()`. If we were designing a real browser that could deal with `_blank` targets, we would need to write a smarter listener that exits only when the last window closed.

### Constructors

Several constructors exist for building frame events:

*public HTMLFrameHyperlinkEvent(Object source, HyperlinkEvent.EventType type, URL targetURL, String targetFrame)*

*public HTMLFrameHyperlinkEvent(Object source, HyperlinkEvent.EventType type, URL targetURL, String desc, String targetFrame)*

*public HTMLFrameHyperlinkEvent(Object source, HyperlinkEvent.EventType type, URL targetURL, Element sourceElement, String targetFrame)*

*public HTMLFrameHyperlinkEvent(Object source, HyperlinkEvent.EventType type, URL targetURL, String desc, Element sourceElement, String targetFrame)*

These constructors allow you to specify the source of the event, its `type` (one of `entered`, `activated`, or `exited`), the destination URL (`targetURL`), and the `targetFrame` for the destination URL. Other options for the last three constructors include a description of the link (which can be used when dealing with malformed URLs), and the source element (see the `sourceElement` property above).

## *Editing HTML*

The `HTMLEditorKit` provides a quick, easy way to display HTML documents. But it is an editor kit, so you can also edit HTML. The `HTMLEditorKit` includes all the necessary tools to edit and generate HTML text. It's not perfect yet, but it is functional. As with the rest of Swing HTML-related classes, this area improves in capability and stability with each new release.

Editing HTML with the `HTMLEditorKit` is as simple as telling your `JEditorPane` to turn editing on:

```
JEditorPane jep = new JEditorPane(mypage);
jep.setEditable(true);
```

You can now edit the text and even provide actions for changing the font style as you did with the styled text editor.

For a complete program, review the example from the beginning of this chapter. It makes use of all the best features of the `HTMLEditorKit` with respect to editing. (The file is *HTMLEditor.java* with the source examples for the same archive.) If you don't really want to edit web pages, but you do want to save them, read on.

## *Outputting HTML*

Our example `HTMLEditor` program has a save function. That save function uses the `write()` method from `JTextComponent` to emit the document to a file stream. The code itself is quite simple:

```
// The filename came from a prompt to the user (via JOptionPane).
FileWriter writer = null;

try {
  writer = new FileWriter(filename);
  getTextComponent().write(writer);
}
catch (IOException ex) {
  JOptionPane.showMessageDialog(HTMLEditor.this,
  "File Not Saved", "ERROR", JOptionPane.ERROR_MESSAGE);
}
finally {
  if (writer != null) {
    try {
      writer.close();
    } catch (IOException x) {  /* yikes...nothing to do now */ }
  }
}
```

The `write()` method takes advantage of the writer installed as part of the `HTMLEdi-torKit`. In this case, that writer is the `HTMLWriter` class. The classes in this section both extend from the `AbstractWriter` class shown below:

### *HTMLWriter*

The `HTMLWriter` class outputs `HTMLDocument` objects to a stream. The writer keeps all tags intact (even ones it doesn't understand) and produces reasonably formatted HTML. Tags carry attributes exactly as you would expect them to. All attributes are enclosed in quotation marks (`<tag  attribute="value">`) to conform to the HTML specification—and to keep embedded spaces in attributes intact.

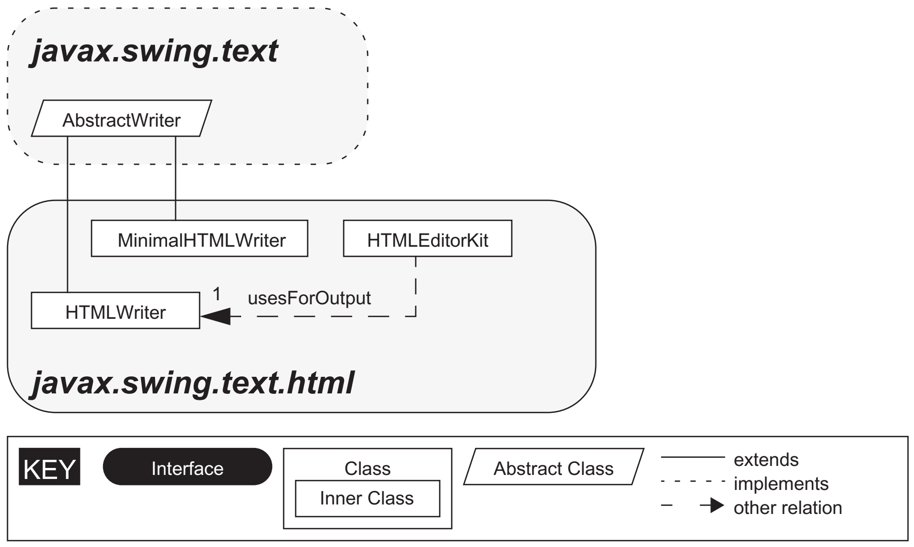


*Figure 2-7. The class diagram for HTML writers.*

## *Constructors*

Two constructors help build a writer from a given `HTMLDocument`:

*public HTMLWriter(Writer w, HTMLDocument doc)*
*public HTMLWriter(Writer w, HTMLDocument doc, int pos, int len)*

Both constructors require a `Writer` object (`w`) and an HTML document to write (`doc`). The writer can be any valid writer including a `FileWriter` or maybe an `OutputStreamWriter` chained on top of a `Socket`. The second version of the constructor creates a writer for a subset of `doc` specified by the starting offset `pos` and extending for `len` characters.

## *Writing Methods*

Only one public method exists:

*public void write() throws IOException, BadLocationException*

This method writes the document (or specified portion) to the given output stream. It loops through the element structure of the document and drives calls to the appropriate protected methods for the various elements it encounters. Comments are directed to the `comment()` method, for example.

## *Protected Tag Writing Methods*

The protected methods manage most of the work involved in writing the document out in proper (and nicely indented) HTML format. Many methods may look familiar to you if you have used the `AbstractWriter` class. Many overridden methods take pains to make sure content outside the normal ASCII range is treated correctly.

*protected void startTag(Element elem) throws IOException, BadLocationException*
*protected void endTag(Element elem) throws IOException*
These methods produce starting and ending tags for the given element.

*protected void writeAttributes(AttributeSet attr) throws IOException*
This method outputs any attribute information for a tag in its proper HTML format.

*protected void emptyTag(Element elem) throws BadLocationException, IOException*
This method handles "empty" tags that have no closing tag (such as comments).

*protected void writeEmbeddedTags(AttributeSet attr) throws IOException*
If the specified attribute set (`attr`) has any embedded HTML tags, those tags are recursively written with appropriate start and end tags.

*protected void closeOutUnwantedEmbeddedTags(AttributeSet attr) throws IOException*
If any accumulated tags do not have matching counterparts in `attr`, close them before writing the HTML for `attr`.

## *Protected Content Writing Methods*

Beyond the tags, several other protected methods are available.

*protected void comment(Element elem) throws BadLocationException, IOException*
This method emits HTML comments for the given Element `elem`. If `elem` is not a comment element, nothing is written.

*protected void output(char[] chars, int start, int length) throws IOException*
This method writes general content. It checks for entity replacement (<, >, &, ") before writing. (Other entities are written using the `&#num;` notation.) If no replacement is desired, the `output()` method from `AbstractWriter` is used.

*protected void selectContent(AttributeSet attr) throws IOException*
This method handles list (both `JList` and `JComboBox`) elements for `<SELECT>` tags in online forms. It calls `writeOption()` as needed for the individual items in the list.

*protected void text(Element elem) throws BadLocationException, IOException*
This method writes general text to the stream. Entities are replaced and line wrapping is allowed if the text is not inside some preformatted block.

*protected void textAreaContent(AttributeSet attr) throws BadLocationException, IOException*

This method outputs any text contained in a `<TEXTAREA>` tag in an online form. The text is properly indented with entity replacement turned on.

*protected void writeLineSeparator() throws IOException*

This method writes line separators, being careful to retain newline information (which is normally ignored).

*protected void writeOption(Option option) throws IOException*

This method writes out `<OPTION>` elements in online forms, one per line. The current selected status of the element in the editor is preserved.

### *Protected Testing Methods*

The protected testing methods allow quick checking of element information to make sure proper HTML is generated. If you build your own editor, you can override these tests to support your new tags.

*protected boolean isBlockTag(AttributeSet attr)*

This method returns `true` if `attr` contains an HTML block tag. If `attr` does not contain an HTML tag at all, it returns `false`.

*protected boolean synthesizedElement(Element elem)*

This method returns `true` if `elem` was "synthesized." Synthesized means an implied element was added to maintain correct document structure. Currently only paragraph tags (`<P>`) can be implied.

*protected boolean matchNameAttribute(AttributeSet attr, HTML.Tag tag)*

This method returns `true` if `attr` contains an HTML tag that matches `tag` and `false` in all other cases.

## *MinimalHTMLWriter*

If you have a simple `StyledDocument` that was not initially an HTML document, you can still emit it as HTML using the `MinimalHTMLWriter`. For example, we could add a "Save As..." option to our `StyledEditor` class to output the document as HTML in addition to the usual method. Here's the code for the new `Action`:

```
class SaveAsHtmlAction extends AbstractAction {
  public SaveAsHtmlAction() {
    super("Save As...", null);
  }
  // Query user for a filename and attempt to open and write the text
  // component's content to the file
  public void actionPerformed(ActionEvent ev) {
    String filename = JOptionPane.showInputDialog(
      SimpleEditor.this, "Enter HTML Filename");
    if (filename == null)
      return;
    FileWriter writer = null;
    try {
      writer = new FileWriter(filename);
            MinimalHTMLWriter htmlWriter = new MinimalHTMLWriter(writer,
                   (StyledDocument)textComp.getDocument());
      htmlWriter.write();
    }
    catch (IOException ex) {
      JOptionPane.showMessageDialog(SimpleEditor.this,
          "HTML File Not Saved", "ERROR", JOptionPane.ERROR_MESSAGE);
    }
    catch (BadLocationException ex) {
      JOptionPane.showMessageDialog(SimpleEditor.this,
          "HTML File Corrupt", "ERROR", JOptionPane.ERROR_MESSAGE);
    }
    finally {
      if (writer != null) {
        try {
          writer.close();
        } catch (IOException x) {}
      }
    }
  }
}
```

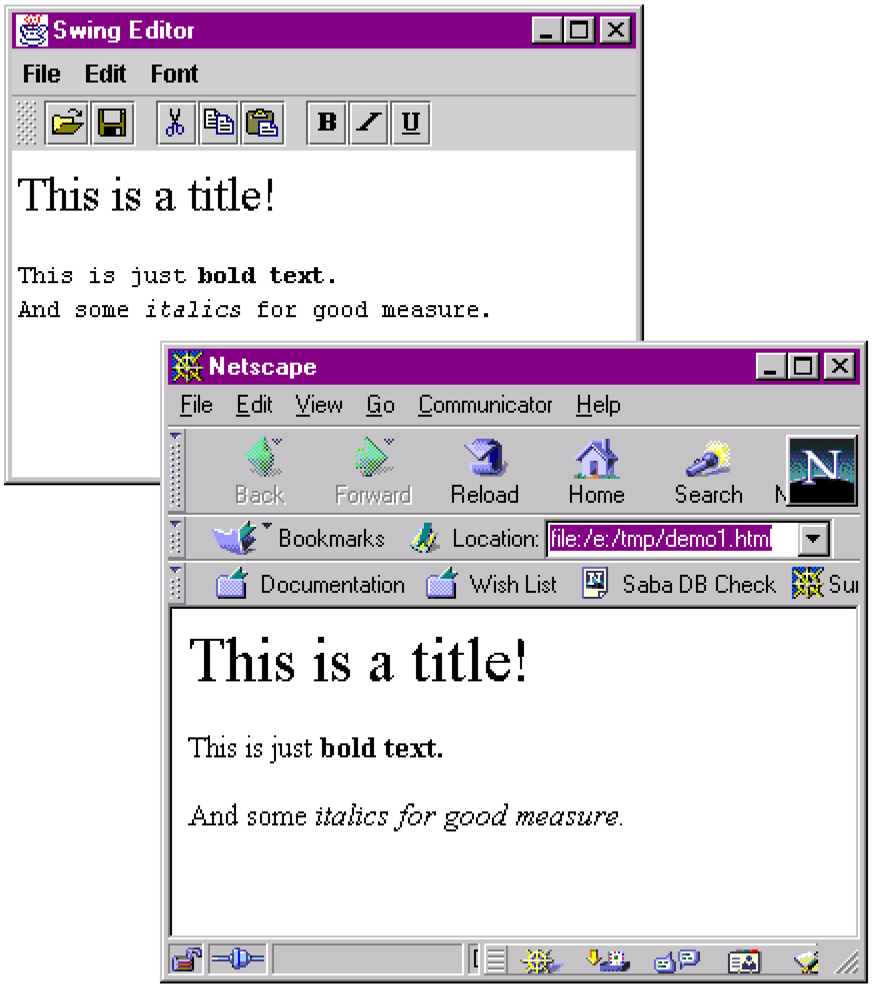
You should see two substantive changes to the `SaveAction` class we saw in Chapter 23 of *Java Swing,* 2nd edition. First, rather than let the `textComp` object (an instance of `JEditorPane`) write itself, we create our `MinimalHTMLWriter` from the `File-Writer` object and the document from our `textComp`. The `write()` method of the `htmlWriter()` spits out the HTML version of the document. Unfortunately, you still have to be careful with this writer—it's not perfect yet. Figure 2-8 shows a screenshot of a simple document in our styled editor, and the same document as HTML displayed inside a Netscape browser.

## Constructors

Similar to the `HTMLWriter` class, two constructors exist:

*public MinimalHTMLWriter(Writer w, StyledDocument doc)*
*public MinimalHTMLWriter(Writer w, StyledDocument doc, int pos, int len)*

These constructors build writers for the given `doc`. The second version writes only `len` characters of content starting from `pos`.


Swing Editor
File Edit Font
Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search
Bookmarks Location: file:/e:/tmp/demo1.html
Documentation Wish List Saba DB Check

This is a title!

This is just **bold text.**
And some *italics* for good measure.

Figure 2-8. The StyledEditor document saved as HTML and displayed in a browser.

## Public Writing Methods

One public method starts the writing process:

*public void write() throws IOException, BadLocationException*
This method writes the document (or the specified part of the document) to the writer given in the constructor. It drives calls to the various protected writing methods below.

## *Protected Tag Writing Methods*

*protected void endFontTag() throws IOException*
*protected void startFontTag(String style) throws IOException*

These methods start and end font tags that match the style given in the `startFontTag()` method. No nested font tags are allowed, so `startFontTag()` calls `endFontTag()` if necessary. Note that these methods are specifically for fonts that cannot be represented by standard HTML tags such as `<B>` and `<I>`.

*protected void writeAttributes(AttributeSet attr) throws IOException*

This method outputs any attribute information for a tag in its proper HTML format.

*protected void writeNonHTMLAttributes(AttributeSet attr) throws IOException*

This method handles font attributes not directly corresponding to standard HTML and drives calls to `startFontTag()`.

*protected void writeHTMLTags(AttributeSet attr) throws IOException*

This method handles standard HTML font attributes such as bold and italic. It also makes sure that no nested font tags occur while maintaining the correct styles for what really amounts to nested fonts. For example, writing out a word with a style of bold and italic in the middle of a segment of text that is already bold should not generate another <B> tag.

*protected void writeStartTag(String tag) throws IOException*
*protected void writeEndTag(String endTag) throws IOException*

These methods simply write the given tags (`tag` and `endTag`, respectively) with proper indenting to keep things looking pretty. The actual content of the tags is built by other methods. For example, `startFontTag()` uses `writeStartTag()` for its `<FONT>` tag.

## *Protected Content Writing Methods*

Like `HTMLWriter`, `MinimalHTMLWriter` has several methods for handling the basic content of the document. Unlike `HTMLWriter`, however, these methods are designed to map standard documents to HTML. You'll see some methods that obviously draw their names from the parts of a document rather than the parts of a web page.

*protected void text(Element elem) throws IOException, BadLocationException*

Similar to `HTMLWriter`, this method emits text. It removes any newline characters at the end of the `elem` (if they exist) before writing. Line breaks are handled by the indenting code in other methods.

*protected void writeBody() throws IOException, BadLocationException*

This method writes the body of the document into the `<BODY>` section of an HTML document.

*protected void writeComponent(Element elem) throws IOException*

This method is, to quote the API documentation, "deliberately unimplemented." If your document contains components, you'll need to subclass `MinimalHTML-Writer` and override this method to produce standard HTML form elements.

*protected void writeContent(Element elem, boolean needsIndenting) throws IOException, BadLocationException*

This method handles writing the attributes and content for a given element in proper HTML fashion. All the non-HTML (Swing implementation) attributes are handled first and then the standard attributes are written. Finally, the text of the element is written.

*protected void writeEndParagraph() throws IOException*
*protected void writeStartParagraph(Element elem) throws IOException*

These methods start and end a paragraph in HTML (`<P>`...`</P>`) for the given element `elem`. Fonts, indentation, and newlines are all handled here.

*protected void writeHeader() throws IOException*

This method produces a valid HTML `<HEAD>` block. The only information placed in the header is a `<STYLE>` for named styles in the document.

*protected void writeImage(Element elem) throws IOException*

As with `writeComponent()`, this method is not implemented. You'll need to override it for producing any `<IMG>` tags in your output. This is a nontrivial method to create. You must decide how to build the `SRC` attribute for the tag and handle any size requirements stored in the document.

*protected void writeLeaf(Element elem) throws IOException*

This method handles nontext elements such as images. As of SDK 1.3, this method handles only images and components, both of which have unimplemented `write()` methods.

*protected void writeStyles() throws IOException*

This method produces the styles stored in the HTML header block. Only paragraph styles (`p.styleName { attribute="value" ... }`) are written.

## Protected Testing Methods

A few convenience routines for testing the state of the writing or the element to be written also exist:

*protected boolean inFontTag()*

This method returns `true` if a `<FONT>` tag has been started but has not been ended. It monitors when tags should be closed before opening new ones.

*protected boolean isText(Element elem)*

This method returns `true` if `elem` is a text leaf (as opposed to a component or image leaf).

Although Swing's HTML support is less than ideal, it is absolutely sufficient to handle simple help systems and aid in rapid prototyping. Each release of the SDK improves the support and usability. It will only continue to get better. In the meantime, if you're desperate for serious markup language support, you really should check out XML. A great place to start is *Java and XML* by Brett McLaughlin—from O'Reilly, of course.