BEA WebLogic Server

Programming WebLogic htmlKona (Deprecated)

WebLogic Server 6.0
Document Date March 20, 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic htmlKona (Deprecated)

Document Edition	Date	Software Version
	March 20, 2001	BEA WebLogic Server Version 6.0

Contents

About This Document

This documenthtmlKona which provides an object-oriented interface to the HTML environment that allows you to format an HTML document with objects. This product is deprecated and is no longer supported.

The document is organized as follows:

- Chapter 1, “Introduction to WebLogic htmlKona,” is an overview of WebLogic’s implementation of JavaServer Pages (JSP) and its architecture.
- Chapter 2, “Using htmlKona,” describes WebLogic’s htmlKona functions and features.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

BEA WebLogic's htmlKona product is deprecated. WebLogic now provides support for JSPs with WebLogic JavaServer Pages (JSP). In keeping with the Java 2 Enterprise Edition standard, JSPs are deployed as part of a *Web Application*. A Web Application is a grouping of application components, such as HTTP servlets, JavaServer Pages (JSP), static HTML pages, images, and other resources. For more information on JSPs, see Programming WebLogic JSP.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and

running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

Convention	Usage
monospace italic text	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Introduction to WebLogic htmlKona

Deprecation of WebLogic htmlKona

WebLogic htmlKona is deprecated in this release of WebLogic Server 6.0. For support for your JSPs, use WebLogic JavaServer Pages (JSP). For more information on JSPs, see Programming WebLogic JSP.

The following sections provide an overview of WebLogic htmlKona and an explanation of how to use JSP with WebLogic Server. It is not intended as a comprehensive guide to programming with JSP.

- What Is htmlKona?
- WebLogic Implementation of JSP
- How JSP Requests Are Handled
- Additional Information

What Is htmlKona?

JavaServer Pages (JSP) is a Sun Microsystems specification for combining Java with HTML to provide dynamic content for Web pages. When you are creating dynamic content, JSPs are more convenient to write than HTTP servlets because they allow you to embed Java code directly into your HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code. JSP is part of the Java Two Enterprise Edition (J2EE).

JSP allows you to separate the dynamic content of a Web page from its presentation. It caters to two different types of developers: HTML developers, who are responsible for the graphical design of the page, and Java developers, who handle the development of software to create the dynamic content.

Because JSP is part of the J2EE standard, JSPs may be deployed on a variety of platforms, including WebLogic Server. In addition, third-party vendors and application developers can provide JavaBean components and define custom JSP tags that can be referenced from a JSP page to provide dynamic content.

WebLogic Implementation of JSP

BEA WebLogic JSP supports the JSP 1.1 specification (see `http://java.sun.com/products/jsp/download.html`) from Sun Microsystems. JSP 1.1 includes support for defining custom JSP tag extensions. (See Programming JSP Extensions at http://e-docs.bea.com/wls/docs60/taglib/index.html.)

WebLogic Server also supports the Servlet 2.2 specification (`http://java.sun.com/products/servlet/download.html#specs`) from Sun Microsystems.

How JSP Requests Are Handled

WebLogic Server handles JSP requests in the following sequence:

1. A browser requests a page with a `.jsp` file extension from WebLogic Server.
2. WebLogic Server reads the request.
3. WebLogic Server converts the JSP into a servlet class that implements `javax.servlet.jsp.JspPage` using the JSP compiler. The JSP file is compiled only when the page is first requested, or when the JSP file has been changed. Otherwise, the previously compiled JSP servlet class is re-used, making subsequent responses much quicker.
4. The generated `JspPage` servlet class is invoked to handle the browser request.

It is also possible to invoke the JSP compiler directly without making a request from a browser. Because the JSP compiler creates a Java servlet as its first step, you can look at the Java files it produces, or even register the generated `JspPage` servlet class as an HTTP servlet.

Additional Information

- *JavaServer Pages Tutorial from Sun Microsystems* at `http://java.sun.com/products/jsp/docs.html`
- JSP product overview from Sun Microsystems at `http://www.java.sun.com/products/jsp/index.html`
- JSP 1.1 Specification from Sun Microsystems at `http://java.sun.com/products/jsp/download.htm`
- *Programming JSP Extensions* at `http://e-docs.bea.com/wls/docs60/taglib/index.html`.
- *Programming WebLogic HTTP Servlets* at `http://e-docs.bea.com/wls/docs60/servlet/index.html`.
- Deploying and Configuring Applications at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html`
- Writing Web Application Deployment Descriptors at `http://e-docs.bea.com/wls/docs60/programming/webappdeployment.html`

2 Using htmlKona

The following sections provide an overview of WebLogic htmlKona and a description of the features.

- Using Deprecated htmlKona
- Overview of htmlKona
- using htmlKona to generate HTML documents

Using Deprecated htmlKona

Although this document is not being updated, major differences between this release and WebLogic Server Version 5.1 are noted in the following table:

Table 2-1 Resources for Deprecated htmlKona

Use this feature . . .	To replace . . .	As described here . . .
myDriver.connect()	DriverManager.getConnection()	`DriverManager.getConnection()` is a synchronized method, which can cause your application to hang in certain situations. For this reason, BEA recommends that you substitute the **`Driver.connect()`** method for `DriverManager.getConnection()`.
Administration Console	weblogic.properties file	Use the Administration Console to set attributes. This replaces the weblogic.properties file.

Overview of htmlKona

The htmlKona classes simplify the task of programmatically generating complex HTML documents. htmlKona, which supports other scripting languages and many browser extensions, is useful both in interactive HTTP servlet environment and for periodic generation of static HTML pages.

htmlKona is a powerful complement to other database- and event-related WebLogic products because of its ability to generate dynamic pages from queries. In a multitier environment, htmlKona can retrieve multimedia objects like audio clips and GIF/JPEG images from heterogeneous databases for incorporation into HTML pages. htmlKona creates class files that, when run against a Java-enabled HTTP server like WebLogic, produce HTML pages for a client browser.

htmlKona provides an object-oriented interface to the HTML environment that allows you to format an HTML document with objects. In htmlKona, there are two basic types of objects: *elements* (derived from weblogic.html.HtmlElement) that are added to a *page* (derived from weblogic.html.WebPage). htmlKona treats a WebPage like a canvas. Instead of using a tagged syntax for mark up, you create a page object and add htmlKona elements to it. When completed, it is rendered by calling one of its `output()` methods.

Features of htmlKona elements:

- Some htmlKona elements are single-part, like the physical and logical elements for text markup; for example, BoldElement, ItalicElement, StrongElement, etc. Others elements, like an UnorderedList, are multipart. You create a multipart element without any contents, and then build with specific single-part elements. An UnorderedList, for example, is built by adding ListItems to it.
- Some htmlKona elements have attributes (like FontElement, for which you can set color, size, and fontname attributes). Attribute themselves may be encapsulated, as the names of the 16 HTML colors are encapsulated in HtmlColor, and the various ways of aligning objects are encapsulated in AlignType.
- Some htmlKona elements have components (like an HtmlContainer).

This document assumes that you have a good working knowledge of both HTML and of the browser(s) that will display your pages; it is not an HTML tutorial or primer. htmlKona provides support for many browser extensions, but you are expected to know which extensions your browser supports.

htmlKona Objects and their classes

The hierarchical arrangement of elements in htmlKona applies the structure and power of an object-oriented environment to the creation of HTML. There are two major descendencies in htmlKona: from WebPage and HtmlElement.

The starting point for every htmlKona page is the creation of an htmlKona WebPage subclass—mostly likely, you will be using htmlKona within an HTTP servlet, and you will create a **ServletPage**. After creating an htmlKona page, you add htmlKona HtmlElements to its head (with `getHead()`) or its body (with `getBody()`) containers, each of which is an weblogic.html.HtmlContainer. The elements are ordered and may be named inside one or more HtmlContainers, which is itself an HtmlElement.

Pages: htmlKona's canvas

All htmlKona pages descend from the class weblogic.html.WebPage. There are five types of htmlKona WebPages:

- Audio pages
- Image pages
- Plain pages
- HTML pages, of which a special case is **ServletPage**

htmlKona is most often used with HTTP servlets that subclass javax.servlet.http.HttpServlet.

Elements: htmlKona's paint tools

All htmlKona elements descend from the class weblogic.html.HtmlElement. There are five general kinds of htmlKona Elements:

1. ElementWithAttributes

2. FileElement (Used rarely)
3. HtmlContainer (Which is itself an HtmlElement)
4. MarkupElement (Encapsulation for various markup elements)
5. StringElement (A special case of HtmlElement)

weblogic.html.ElementWithAttributes is further subclassed into two groups: weblogic.html.MultiPartElement and weblogic.html.SinglePartElement. ElementsWithAttributes includes most of the elements you will use to build a WebPage.

using htmlKona to generate HTML documents

Classes in htmlKona can be organized by functionality or by a hierarchy of descendent relationships. In this portion of the documentation, the classes are grouped together functionally, that is, all of the elements you might need to create a list, for example, are discussed under the same topic.

- Setting up a page
  - Creating an HtmlPage
  - Adding the HEAD elements
  - Adding attributes to the BODY element
  - Building the document body
  - Example
- Other page-defining elements
- Setting block attributes
- Setting physical and logical attributes
- Other text-level classes

- Encapsulation classes
- Creating anchors
- Using lists
- Using images
- Using serverside and client-side image maps
- Using frames
- Using tables
- Setting up forms
- Adding a script to a page
- Adding an applet to a page
- Embedding a file on a page
- Using htmlKona to display dynamic data
- Using htmlKona with Java-enabled servers
  - Java Server serverside servlet example
- Shortcuts for testing output

Setting up a page

The first step in using htmlKona to create a web page is to construct a WebPage object and, if needed, set its HEAD elements and other attributes. Then you build the document by adding HtmlElements to the WebPage. After you compile the htmlKona file, you place it in the proper directory on a Java-enabled server, restart the server (in some cases), and request the page from the server with a URL from your browser.

There are five types of WebPages. They are all subclassed from the parent class weblogic.html.WebPage.

- weblogic.html.HtmlPage. Creates a common HTML-tagged document that is built by adding HTML elements to it. Adds a Content-type header to the page,

and allows you to set other header information with methods from its superclass, if desired.

- weblogic.html.ServletPage. Creates a page appropriate for serverside Java, that is, without a Content-type header. With serverside Java, header information is provided with other methods that depend upon the server you are using.
- weblogic.html.AudioPage. Constructed with the audiotype and the byte array that is the audio data for creating the page.
- weblogic.html.ImagePage. Constructed with the imagetype and the byte array that is the image data for creating the page.
- weblogic.html.PlainPage. Creates a non-HTML-tagged page. It displays text exactly as it exists, without any HTML formatting.

HtmlPages are of content-type “text/html", and are most commonly used. AudioPages and ImagePages are used to create multimedia HTML documents with Blob audio or visual images retrieved from a database. PlainPages are used for displaying unformatted text. ServletPages are subclassed from HtmlPage, but the content-type is not set. The only difference in these pages is the content-type that is passed to the HTTP server.

There are methods in the WebPage superclass for setting content encoding, length, and type; for setting location, server, referer, and pragma; for setting expiration and last modified dates, and other attributes.

There are methods in each type of WebPage to manipulate the types of HtmlElements you will manage.

Creating an HtmlPage

The most commonly used WebPage object is a ServletPage, which is the subclass of HtmlPage that is used with Java-Servlet-API style HTTP servlets. An HtmlPage is the htmlKona canvas where you lay out a description of the page with HtmlElements; when you are finished, you call the output() method and the HTML for the page is automatically generated.

The first step is to create an ServletPage:

```
ServletPage hp = new ServletPage();
```

You can also construct the page with its title as a String argument.

To set the codeset for a page, use the ServletPage constructor that takes a Codeset as an argument, for example:

```
ServletPage sp = new ServletPage(new CodeSet("SJIS"));
```

Adding the HEAD elements

htmlKona supports all of the HTML HEAD elements, which provide information to both the server and the browser. Add these to the page by calling the `HtmlPage.getHead()` method, and then using the `addElement()` methods:

```
hp.getHead()
  .addElement(new TitleElement("htmlKona test2"))
  .addElement(new MetaElement(MetaElement.nameType,
                              MetaElement.nameDescription,
                              "WebLogic test meta element"))
  .addElement(new MetaElement(MetaElement.equivType,
                              MetaElement.httpEquivExpires,
                              new java.util.Date()))
  .addElement(new MetaElement(MetaElement.nameType,
                              MetaElement.nameRobots,
                              MetaElement.indexNoFollow))
  .addElement(new LinkHeadElement(LinkHeadElement.relTag,
                              LinkHeadElement.relHome,
                              "www.weblogic.com",
"WebLogic Home"));
```

The htmlKona classes for HEAD elements are:

- weblogic.html.TitleElement (The only element required on an HTML page)
- weblogic.html.IsIndexElement
- weblogic.html.LinkHeadElement
- weblogic.html.MetaElement
- weblogic.html.ScriptElement
- weblogic.html.StyleElement
- weblogic.html.BaseElement
- weblogic.html.BaseFontElement (Browser extension)

Both MetaElement and LinkHeadElement contain constants that encapsulate possible names and values for these elements. Also supported is the BaseFontElement, a browser extension to the page head that allows you to set a default font size for the entire document.

Adding attributes to the BODY element

After setting up the HEAD portion of the page, call the `HtmlPage.getBodyElement()` method to set attributes for the BODY elements. You set attributes using the `setAttribute()` method from the BodyElement class, which also provides a set of constants for the current BODY attributes.

```
hp.getBodyElement()
   .setAttribute(BodyElement.bgColor, "#FFFFFF")
   .setAttribute(BodyElement.backgroundImg,
                 "images/wtbkg.gif")
   .setAttribute(BodyElement.linkColor, HtmlColor.fuchsia)
   .setAttribute(BodyElement.aLinkColor, "#080808")
   .setAttribute(BodyElement.vLinkColor, "#808080");
```

Also supported are ONLOAD and ONUNLOAD attributes for scripts. Note that the 16 HTML color names are encapsulated in the weblogic.html.HtmlColor class. All htmlKona operations that involve a color also provide methods that take a java.awt.Color object as an argument.

Building the document body

Once you have created an HtmlPage, added its HEAD elements, and set its BODY attributes, you can add HtmlElements to it with various `addElement()` methods. When you are finished, use one of the `HtmlPage.output()` methods to output the page to the proper source.

Example

Here is a very simple example of the complete code from a class written using htmlKona. This class is written with a `main`, to allow you to output the HTML without running the class against a Java-enabled server.

```
import java.io.*;
import weblogic.html.*;

public class helloworld {
```

```
  public static void main(String argv[])
    throws HtmlException, IOException
  {
    HtmlPage hp = new HtmlPage();
    hp.getHead()
      .addElement(new TitleElement("The Hello World Page"));
    hp.getBodyElement()
      .setAttribute(BodyElement.bgColor, HtmlColor.white)
      .setAttribute(BodyElement.backgroundImg,
                    "images/mylogo.gif");
    hp.getBody()
      .addElement(MarkupElement.HorizontalRule)
      .addElement(new StringElement("Hello World!")
                  .asBoldElement())
      .addElement(MarkupElement.HorizontalRule);
    hp.output();
  }
}
```

Here is what the HTML for this example will look like: **Hello World!**

To test this example:

1. Copy it into a file named `helloworld.java`.
2. Compile it.
3. Place the class file in your `CLASSPATH`.
4. Use `java helloworld > test.html` to create the HTML output.
5. View the HTML file in a browser.

Other page-defining elements

You can also create frames with htmlKona. Not all browsers will display frames. htmlKona provides extensive support for frames.

Setting block-level elements

The HTML specification divides markup elements into two general classes: those that cause paragraph breaks (block-level), and those that do not (text-level, both physical and logical elements). htmlKona makes an additional distinction between block-level elements that have multiple parts—like lists and forms—and those that contain only

text or other text-level kinds of elements. This document is arranged to track as closely as possible the organization that HTML itself uses, so that if you are familiar with HTML, you will find htmlKona easy to understand. Consequently, the first simple block-level elements are covered, then physical and logical elements, followed by more complex elements like tables and forms.

Simple, single-part block-level elements in htmlKona include the following:

- weblogic.html.BlockquoteElement
- weblogic.html.BreakElement
- weblogic.html.CenteredElement
- weblogic.html.DivElement
- weblogic.html.HeadingElement
- weblogic.html.HorizontalRuleElement
- weblogic.html.LiteralElement
- weblogic.html.ParagraphElement

Check the methods in each class for additional attributes you can set for these objects. You use these objects very simply: by creating a new object and adding it to an HtmlContainer, like the body portion of an HtmlPage. For example, here is how you would add a heading (H2) and a horizontal rule to the body portion of an HtmlPage "hp":

```
hp.getBody()
  .addElement(new HeadingElement("Breaking New Ground", 2))
  .addElement(new HorizontalRuleElement());
```

Many of these elements, in particular those that do not require an ending tag, are encapsulated in the general htmlKona class, weblogic.html.MarkupElement. For example, you can insert a horizontal rule into your HtmlPage in two ways:

```
hp.getBody()
  .addElement(new HorizontalRuleElement()
              .setAlign(AlignType.center)
              .setWidth("50%"))
```

or merely

```
hp.addElement(MarkupElement.HorizontalRule)
```

The first way, by creating a HorizontalRule object, allows you to use other methods in the HorizontalRule class to set additional attributes for the object. The second way provides a convenient short-cut for adding a markup element to your page, if you do not intend to set any further attributes.

Setting physical and logical attributes

Classes for physical and logical attributes:

- weblogic.html.BigElement
- weblogic.html.BoldElement
- weblogic.html.CiteElement
- weblogic.html.CodeElement
- weblogic.html.CommentElement
- weblogic.html.DefineTermElement
- weblogic.html.EmphasisElement
- weblogic.html.FontElement
- weblogic.html.ItalicElement
- weblogic.html.KeyboardElement
- weblogic.html.SampleElement
- weblogic.html.SmallElement
- weblogic.html.StrikeElement
- weblogic.html.StrongElement
- weblogic.html.SubscriptElement
- weblogic.html.SuperscriptElement
- weblogic.html.TeletypeElement
- weblogic.html.UnderlineElement

- weblogic.html.VariableElement

Physical and logical attributes are handled in a generic way. You construct a new object and add it to the HtmlPage (or any HtmlContainer), using its `setXXX()` methods to set additional attributes. In some cases, you can cast an HtmlElement as another type of element, in order to use methods in another class to set additional attributes.

In this example, create a FontElement, and set its size and color:

```
hp.getBody()
  .addElement(new FontElement(5, HtmlColor.red,
                              "To the Stars!"));
```

You can also cast another HtmlElement as a FontElement in order to use methods in the FontElement class on it. For example, here you center the element and set its color.

```
hp.getBody()
  .addElement(new HeadingElement(filename, 2)
              .setAlign(TextAlignType.center)
.asFontElement("3", HtmlColor.fuchsia));
```

Other text-level classes

There are two other text-level classes that do not cause paragraph breaks and are not specifically a physical or logical element:

- weblogic.html.AddressElement
- weblogic.html.SpacerElement

AddressElement

The AddressElement class identifies the author of the page, and also has a constructor for creating a hyperlink for contact information. It is often displayed in italic text. For example:

```
hp.getBody()
  .addElement(new AddressElement("http://www.weblogic.com",
                "BEA Systems Inc."));
```

If you use an AddressElement to add a multiline address to a page, you should add a BreakElement (or a MarkupElement.Break) between each line. For example:

```
HtmlContainer ae = new HtmlContainer();
ae.addElement("BEA Systems Inc.")
  .addElement(MarkupElement.Break)
  .addElement("180 Montgomery Street, Suite 1240")
  .addElement(MarkupElement.Break)
  .addElement("San Francisco, California 94104")
  .addElement(MarkupElement.Break);
hp.getBody()
  .addElement(new AddressElement(ae));
```

SpacerElement

The SpacerElement (supported in some browsers) allows you to set horizontal, vertical, or block whitespace between objects on a page. Either the size attribute, or the width, height, and alignment attributes are set. A SpacerElement constructor may use a weblogic.html.**SpacerType** object to set the SpacerElement's type attribute.

Encapsulation classes

htmlKona provides several classes for encapsulating certain kinds of attributes that are used often by other objects. Except for the Codeset and the HtmlColor interface, encapsulation classes are all subclassed from MarkupElement. Classes that are in general use, like AlignType and HtmlColor are discussed here; other classes that are associated with a particular element or set of elements are discussed with that element at other (linked) places in this document. Encapsulation classes:

- CodeSet (used with ServletPage and HtmlPage constructors)
- Align Type
- AnchorType (used with AnchorElement)
- BorderstyleType (used with TableElement in some browsers)
- ClearType
- FrameType (used with TableElement in some browsers)
- FieldType (used with InputElement)
- HtmlColor

- RulesType (used with TableElement in some browsers)
- ScrollType (used with FrameElements)
- SpacerType (used with SpacerElement)
- TextAlignType
- WindowName
- WrapType (used with TextAreaElement)

Codeset

Use a weblogic.html.Codeset

object as an argument to either ServletPage or HtmlPage to set the document's codeset. The default codeset if unset is "8859_1", which is the Latin character set used by English, German, and the Romance languages. An example:

```
ServletPage sp = new ServletPage(new Codeset("SJIS"));
```

which sets the codeset to Shift-JIS, Japanese. Codesets are defined in the JavaSoft internationalization specification documents at JavaSoft.

AlignType

Use a weblogic.html.AlignType

object as an argument to the various setAlign() methods to set alignment of HtmlElements like AppletElement, HorizontalRuleElement, and the various Table*Elements. The variables in this class set the alignment.

Note: TextAlignType, not AlignType, is used to align text in a container.)

Here is an example of using an AlignType object to set alignment for parts of a table:

```
TableElement table = new TableElement();
table.setCaption(new TableCaptionElement("Usage Statistics")
                  .setAlign(AlignType.bottom))
     .setBorder(1)
     .addElement(new TableRowElement()
                 .addElement(new TableDataElement(reg.getID())
             .setVAlign(AlignType.top))
                 .addElement(new TableDataElement(
               new TableElement(reg.getMain())
```

```
                                .setCaption(
        new BoldElement("N1"))
                                .setBorder(1))
                            .setVAlign(AlignType.top)
    .setAlign(AlignType.right)));
```

TextAlignType

A weblogic.html.TextAlignType object is used to set the alignment for the text contents of HtmlElements likeParagraphElement, DivElement, and HeadingElement. The variables in this class include center, justify, left, and right. Use TextAlignTypeobjects as arguments for the setAlign() methods of text containers.

ClearType

A weblogic.html.ClearType

object sets the HTML CLEAR attribute, which moves an HtmlElement (BreakElement or a DivElement) down unconditionally after a figure or a table, rather than flowing around the figure or table.

HtmlColor

The 16 HTML color names are encapsulated in the class weblogic.html.HtmlColor as String elements, so that methods and constructors that take a color as an argument can use either a variable from the HtmlColor class (like HtmlColor.white), or can use the six-letter string hex RGB value (like

"#FFFFFF").

Note: There are usually identical methods and constructors that also take a java.awt.Color object as an argument.

WindowName

Use a weblogic.html.WindowName object to set a target window for a link with the setTarget() methods in various htmlKona classes. Variables in the class are analogous to the HTML magic target names _blank, _parent, _self, and _top. Targets for links are often used with frames, images and image maps, and anchor elements.

Creating anchors

Classes for creating anchors:

- AnchorElement
- AnchorType

Use a weblogic.html.AnchorElement object to set hypertext links in your HtmlPage. (It supersedes an older class,LinkElement, that is still supported for backwards compatibility.)There are two real uses for anchors; one use is with an HREF attribute to set a hyperlink to another document or an internal link, and the other use is with a NAME attribute to identify an internal link. For instance, this paragraph is marked with the anchor <ANAME=

"anchorelementdef">, which allows you to jump to this paragraph from a link within this document that is marked with the anchor `<A HREF=#anchorelementdef>`.

`Consequently, the constructors for this class are of two types, one set that is primarily for hyperlinking text with an HREF attribute, and the other that is primarily for marking text with a NAME attribute. The constructors in this class also use objects from the encapsulated class weblogic.html.` **AnchorType** as arguments for the constructor, to remove the ambiguity in using simple Strings for the HREF and NAME attributes. You can, in fact, use both the HREF and NAME attributes in the same anchor, to provide an internal destination, and also to link to another document. If you view the source of this paragraph, you will see that the hyperlink above for the AnchorType class name uses both attributes.

You can also use a weblogic.html.WindowName object in one of the AnchorElement constructors to direct the specified URL into a different browser window.

Here are some AnchorElement uses:

```
hp.getBody()
  .addElement(
    new AnchorElement("http://www.acme.com/copyright.html",
                      "© 1996, Acme Inc."))
  .addElement(
    new AnchorElement(AnchorType.href,
                      "http://www.acme.com/copyright.html",
                      "© 1996, Acme Inc."))
  .addElement(
    new AnchorElement(AnchorType.name,
```

```
                        "topic1",
                        "The most important topic"))
    .addElement(
      new AnchorElement("http://www.acme.com/copyright.html",
                        "© 1996, Acme Inc.",
        WindowName.parent));
```

You can also set an internal NAME anchor in with the HtmlPage.asAnchorElement() method, as in this example:

```
  hp.getBody()
    .addElement(new StringElement("This text is an " +
                                  "internal anchor")
                .asAnchorElement("anchor1"))
    .addElement(new StringElement(" in this document."));
```

Using lists

There are two types of lists: DefinitionLists and ListElements. htmlKona supports several subclasses of ListElement, including OrderedList and UnorderedList, MenuList, and DirList. Note that HTML 3.2 deprecates both MenuList and DirList to be replaced by the more general UnorderedList. htmlKona supports these for backwards compatibility, but not all browsers may continue to display such lists.

Classes for lists:

- weblogic.html.DefinitionList contains one or more weblogic.html.DefinitionItems
- weblogic.html.ListElement contains one or more weblogic.html.ListItems and comes in these flavors:

weblogic.html.DirList

- weblogic.html.MenuList
- weblogic.html.OrderedList
- weblogic.html.UnorderedList)

DefinitionList

DefinitionLists are made up of DefinitionItems, which are added with the addElement() method. A DefinitionItem has two parts, a term and a definition. The DefinitionItems in a List are accessible by index position.

ListElement

There are several kinds of ListElements that correspond to HTML ordered and unordered lists. A ListElement is made up of ListItems or strings. The items in a list are accessible by index position. htmlKona supports several additional attributes for list elements.

Here is an example of building an unordered list:

```
UnorderedList ul = new UnorderedList();
ul.setType(UnorderedList.circle)
  .addElement(new ListItem("$500"))
  .addElement(new ListItem("$250"))
  .addElement(new ListItem("$100"));
```

This produces the following output:

- $500
- $250
- $100

htmlKona supports the type attribute for both ordered and unordered lists. Valid types for each kind of list are encapsulated as String constants in each class.

htmlKona also supports the start argument for OrderedLists and for ListItems added to OrderedLists. This allows you to control the numbering sequence, as in this example:

```
OrderedList ol = new OrderedList();
ol.setType(OrderedList.largeRoman)
  .setStart(13)
  .addElement("500")
  .addElement("600")
  .addElement("435")
  .addElement(new ListItem("250")
              .setType(OrderedList.smallRoman)
              .setValue("7"))
  .addElement("195")
  .addElement("100");
```

This would be displayed as follows:

1. 500
2. 600
3. 435

4. 250
5. 195
6. 100

Using images

Classes for image management:

- weblogic.html.ImageElement

Use the ImageElement to place an inline image on an HtmlPage. For example:

```
hp.getBody()
  .addElement(new ImageElement("http://website/images/bar.gif"));
```

The ImageElement class has methods for setting height, width, Hspace, Vspace, source, border, alternate text, and alignment for an ImageElement. In addition, htmlKona supports both serverside and client-side image maps, for which there is an additional ImageElement methods, setUseMap() and setIsMap(boolean) methods.

You can also use a weblogic.html.ImagePage to display an image retrieved as a byte array (byte[]) from a database.

Using serverside and client-side image maps

Classes for serverside image maps:

- weblogic.html.AnchorElement
- weblogic.html.ImageElement

Serverside image maps run a CGI script on the server to respond to mouse events over regions of an image. The additional attribute ISMAP is required for an ImageElement to act as an image map. To create a serverside image map, you should use an ImageElement as an argument for an AnchorElement that sets the CGI script as its URL. For example:

```
// Create the ImageElement
ImageElement ie =
  new ImageElement("http://www.weblogic.com/images/h.gif");
// Set the ISMAP attribute
ie.setIsMap(true);
hp.getBody()
  // Construct an AnchorElement with the CGI script as its URL
  // and the ImageElement as its display
  .addElement(new AnchorElement("/cgi-bin/imagemap", ie));
```

Several browsers now also support client-side image maps, which allow you to use local information associated with an image instead of serverside information (a serverside image map). htmlKona supports client-side image maps, but you should make sure the browser intended for display supports them.

Classes for client-side image maps:

- weblogic.html.AreaElement
- weblogic.html.ImageElement
- weblogic.html.MapElement

A MapElement, which is constructed by setting its name, is a multipart element that is created by adding AreaElements to it. Use an AreaElement to set coordinates for each "hotzone" on the image map.

An AreaElement is used to set coordinates for a region of the image that is mapped to a particular URL. Valid region shapes, encapsulated (as String constants) in the AreaElement class, or java.awt.Rectangle and java.awt.Polygon, can be used with the constructors and methods. The class contains other methods for setting coordinates and URLs. Note that setting the ALT attribute is highly recommended, to give the user some information when the image is not present or not displayed.

This example shows how to build a simple MapElement that provides links to help from a screen image. First, set up some arrays that will be used in some of the constructors; then add an ImageElement; and finally, create the MapElement.

```
int[] xcoords = {353, 475, 353};
int[] ycoords = {116, 163, 163};
int ncoords = 3;
int[] rectcoords = {353, 33, 475, 53};

hp.getBody()
  // Create a new ImageElement and call its setUseMap() method
  .addElement(
    new ImageElement(
```

```
            "http://www.weblogic.com/images/ipscreen.gif")
               .setAlign(AlignType.center)
               .setUseMap("#MyMap", true))
        // Create a new MapElement
        .addElement(new MapElement("MyMap")
                 // Add an AreaElement. This one uses an
                 // array of ints as its coordinates.
                 .addElement(new AreaElement()
                             .setShape(AreaElement.rectangle)
                             .setHref("htmlimgmap.html#ok")
                             .setCoordinates(rectcoords))
                 // This constructor takes a string
         //as its coordinates
                 .addElement(new AreaElement(AreaElement.rectangle,
                              "353, 60, 475, 82",
                              "htmlimgmap.html#cancel"))
         // This constructor takes an
         // AWT Rectangle as an argument
                 .addElement(new AreaElement(new java.awt.Rectangle(
                       353,88,475,100),
                                       "htmlimgmap.html#dns"))
         // This constructor takes an AWT Polygon
         // as an argument
                 .addElement(
    new AreaElement(new java.awt.Polygon(xcoords,
                                         ycoords,
                          ncoords),
                                       "htmlimgmap.html#advanced"))
                    .addElement(
       new AreaElement(AreaElement.circle,
                                       "220, 193, 40",
                                       "htmlimgmap.html#circle")));

      hp.output();
```

Using frames

Classes for frames:

- weblogic.html.FrameElement
- weblogic.html.FrameSetElement
- weblogic.html.NoFramesElement
- weblogic.html.ScrollType

FrameElement

Use one FrameElement for each frame on the page. There are methods in this class for setting frame margins, the name and source for the target window, the scrolling behavior, and whether or not the frame can be resized.

Note: If you set the noresize attribute of a frame to true, other frames on the page with which it shares a side will also automatically be set to noresize.

htmlKona also supports browser extensions for setting bordercolor and frameborder, as well as the onLoad and onUnload attributes for using scripts.

FrameSetElement

Use a FrameSetElement as a container for FrameElements to design an HTML page that uses frames. Construct a new FrameSetElement and then use its addElement() methods to add FrameElements to it.

Each FrameSetElement may contain two FrameElements or other FrameSetElements. You can use multiple FrameSetElements to position frames within frames. Use the setCols() and setRows() to divide the screen size (in pixels or percentage) between the two objects contained in the FrameSetElement. htmlKona also supports browser extensions for bordercolor, frameborder, and border attributes. The elements in a FrameSetElement are accessible by integer index.

NoFramesElement

Use a NoFramesElement to set the contents of a page with frames that should be displayed if the browser cannot display frames.

ScrollType

The ScrollType class encapsulates valid scrolling attributes for frames in htmlKona. Use a ScrollType to set the scrolling characteristics for FrameElement. You use a ScrollType object as an argument for the setScrolling() method in the FrameElement class.

Example

This example shows how to construct one FrameSetElement *fs* and set the size of its panels to 30% and 70% of the page. Next step is to set the smaller side of the FrameSetElement to the FrameElement *Frame1*. Then, construct another FrameSetElement *fs2* containing two FrameElements (*Frame2* and *Frame3*). The entire FrameSetElement *fs2* is added to the first FrameSetElement as the larger side of the page.

```
  String base = "/ns-home/frameExample";
  FrameSetElement fs = new FrameSetElement()
                       .setRows("30%,70%")
.setBorderColor(HtmlColor.fuchsia)
.setFrameBorder(true);
  fs1.addElement(new FrameElement().setName("Frame1")
                       .setSource(base + "?name=Frame1")
                .setScrolling(ScrollType.no));
  FrameSetElement fs2 = new FrameSetElement()
                       .setCols("50%,50%");
  fs2.addElement(new FrameElement().setName("Frame2")
                       .setSource(base + "?name=Frame2"));
  fs2.addElement(new FrameElement().setName("Frame3")
                       .setSource(base + "?name=Frame3"));
  fs1.addElement(fs2);
  hp.setFrameSetElement(fs1);
```

Using tables

Classes for tables:

- weblogic.html.TableElement
- weblogic.html.TableCaptionElement
- weblogic.html.TableHeadingElement
- weblogic.html.TableRowElement
- weblogic.html.TableDataElement

The chief table object in htmlKona is the **TableElement**. Construct the TableElement, sets its caption and attributes, and then add rows of headings or data to it.

TableElements are generically composed of **TableRowElements**, which are generically composed of **TableDataElements**, or **TableHeadingElements**, a special kind of TableDataElement. A TableDataElement corresponds to a single table cell; you can set the contents of a TableDataCell as well, after it is constructed. TableRowElements and TableDataElements can also be built with strings or with other HtmlElements. For instance, you can use an ImageElement to display an image in the cell of a table. You use **TableCaptionElements** to set a caption for the table.

You can also use a dbKona DataSet to construct a TableElement. Using htmlKona to display data from a dbKona DataSet is convenient because the Schema object associated with a DataSet can automatically supply the structure of a TableElement when the TableElement is constructed. Likewise, you can construct a TableRowElement with a single Record from a dbKona DataSet.

The TableElement class has methods for adding, getting, and setting rows and cells. Rows and cells in a TableElement are accessible by index position.

The TableElement class has also methods for setting attributes such as border, caption, cellpadding, cellspacing, and width. htmlKona supports certain browser extensions for tables like background color and image attributes, and Mosaic's BORDERSTYLE (with encapsulated constants from the weblogic.html.**BorderstyleType** class) attribute. It also supports FRAME and RULES attributes that have been proposed for adoption in HTML3.5; use encapsulated constants from the weblogic.html.**FrameType** and weblogic.html.**RulesType** classes to set these attributes. Set alignment for table elements with an AlignType objects.

Example

This example shows how to construct two tables, one nested inside the other. Note that in some cases the TableRowElement is constructed by adding arbitrary HtmlElements; in other cases, a TableDataElement is added. You can use any HtmlElement as an argument to TableRowElement.addElement(), but if you want to set any characteristics on the table cells, such as alignment or background color, you must add the cell as a TableDataElement, and use methods in that class to set cell characteristics.

```
// Create the table and set attributes and caption,
// which will display at the bottom of the
// table in an italic font
TableElement tab = new TableElement()
  .setBorder(1)
  .setCellPadding(5)
  .setWidth("50%")
  .setCaption(new TableCaptionElement(
```

```
            new ItalicElement("Fall Registration Schedule"))
           .setAlign(AlignType.bottom));
// Add a table row and build it with table headings.
// Note that we set the first table heading as an
// internal anchor by creating it as an AnchorElement.
// We also set a different background
// color for the second column heading.
tab.addElement(new TableRowElement()
            .addElement(
   new TableHeadingElement(new AnchorElement(
                              AnchorType.name,
                                      "topic",
                                      "Topic")))
            .addElement(new TableHeadingElement("Scheduled")
                .setBgColor("#A93B6F")));
// Add a second row. In the first column we call
// a method to get an ID, and the second column
// we make a nested table. We align the nested
// table at the top of the cell by calling
// the setVAlign() method on the TableDataElement.
tab.addElement(new TableRowElement()
             .addElement(new TableDataElement(reg.getID())
      .setVAlign(AlignType.top))
  // The nested table is the output of a
  // method that retrieves a dbKona DataSet
             .addElement(new TableDataElement(
      new TableElement(reg.getMain())
                        .setCaption(new BoldElement("N1"))
                        .setBorder(1))
                        .setVAlign(AlignType.top)));
// Add a row with simple strings
tab.addElement(new TableRowElement()
             .addElement("RegID 609E")
  .addElement("N2 - 10AM MWF"));
// Finally, add the table as a centered
// element to the HtmlPage.
hp.getBody()
    .addElement(new CenteredElement(tab));
```

Setting up forms

Classes for forms:

- weblogic.html.FormElement
- weblogic.html.InputElement, which uses the class weblogic.html.FieldType

- weblogic.html.SelectElement, which uses the class weblogic.html.OptionElement
- weblogic.html.TextAreaElement, which uses the class weblogic.html.WrapType

To produce a form in htmlKona, you construct a FormElement and add InputElements, SelectElements, and/or TextAreaElements to it. Then you use HtmlPage.addElement() to add the FormElement to the page.

As in HTML, FormElements can be nested inside TableElements to make it easy to lay out the form. When you use a TableElement to set up the layout for a form, make sure you are operating on the TableRowElements or TableDataElements when setting table attributes, and not on the contents of your FormElement. (This is further illustrated in the example code below.)

The type of InputElement is set in the constructor with a weblogic.html.**FieldType** object. The FieldType class encapsulates all of the valid types of HTML input elements, including checkbox, textarea, radio, submit buttons, etc. The class also supports the following:

- Button
- Checkbox
- File
- Hidden
- Image (for using an image with a submit button)
- Password
- Radio
- Reset
- Submit
- Text

A SelectElement, which creates a dropdown- or scrolling-type box of selectable items on a form, is made up of OptionElements. You construct a SelectElement and use the addElement() methods to add options, which can be strings or OptionElements. Use an OptionElement if you will need to use any of the methods in the OptionElement class to set attributes. An OptionElement has a Value that is not displayed, as well as a string that is displayed.

TextAreaElements are multiline text boxes for user input. You can present the user with some default text in the textbox with the setContent() method. (For a single-line textbox, use an InputElement of FieldType.text.) Set height and width of the textbox with the setRows() and setCols() methods. htmlKona also supports extensions like the attributes ONFOCUS, ONBLUR, ONSELECT, and ONCHANGE.

You set the text-wrap characteristics for a TextAreaElement with a **WrapType** object. Use the TextAreaElement.setWrap() method with a WrapType object to set the text wrap to off, virtual, or physical wrap.

Getting form data

When you process your htmlKona FormElement against a Java-enabled server, you must retrieve information entered by the user with the Submit button, i.e., the Common Gateway Interface (CGI) variables. Very simply, when the user presses a Submit button on your form, the same class that displayed the new form is invoked again, this time retrieving and using information that the user entered on the form. As a rule, CGI variables are referenced to the NAME attribute of the FormElement; the contents of the variable depend upon the type of input element, that is, a textarea element will return a string and a select element will return the index position of the selected option.

With both Java Server and Netscape servers, CGI variables are stored as a Hashtable object, and you use the get() methods to retrieve the contents of the variables.

The examples that follow show real-life uses for serverside Java against both Java Server and Netscape servers, including implementation details for Java Server and Netscape.

Example

Here is an example that shows the several steps involved in creating a FormElement. This example also uses a TableElement to line up the user input areas on the form. There are six steps shown in this example:

1. Construct an HtmlPage and set its title. You can add other attributes to the head portion, and you can call HtmlPage.getBodyElement() to set other attributes for the body potion of the page.
2. Construct a FormElement.
3. Construct a TableElement to organize the various pieces of the form.

4. Add rows to the table that contain InputElements.
5. Add the TableElement and any other InputElements you want to add to the FormElement.
6. Add the FormElement to your HtmlPage, along with other HtmlElements.

```
import weblogic.html.*;
import java.io.*;

public class myform {

  public static void main(String argv[])
      throws HtmlException, IOException {

    // Create a page and set its title
    HtmlPage hp      = new HtmlPage();
    hp.getHead()
      .addElement(new TitleElement("Creating a Form with
htmlKona"));

    // Create a form
    FormElement form = new FormElement("/servlet/myform",
                                       "POST");

    // Create a table and use it to organize form elements
    TableElement tab = new TableElement().setBorder(0);

    // Add input elements and labels for the form
    // as rows in the table
    tab.addElement(new TableRowElement()
            .addElement(new TableHeadingElement("Your name"))
    .addElement(new TableHeadingElement("Password"))
    .addElement(""));
    tab.addElement(new TableRowElement()
    .addElement(new InputElement("namefield",
                         FieldType.text)
        .setValue("Your name")
        .setSize(20)
.setMaxlen(30))
    .addElement(new InputElement("password",
                        FieldType.password)))
.addElement(new TableRowElement()
    .addElement(new TableDataElement(
                   new InputElement("submit",
                              FieldType.submit)
           .setValue("Check my account"))
.setColspan(2)));
```

```
    // Add a TextAreaElement and the table to the form
    form.addElement(new TextAreaElement("notes",
                                        "Enter your comments")
    .setCols(30))
        .addElement(tab);

    // Add the form to the page
    hp.getBody()
      .addElement(MarkupElement.HorizontalRule)
      .addElement(new HeadingElement("Feedback Form", 2))
      .addElement(form)
      .addElement(MarkupElement.HorizontalRule);
    hp.output();
  }
}
```

Adding a script to a page

Classes for use with scripts:

- weblogic.html.ScriptElement

Note: Script functions are called from event handlers within the page that can be set with methods in many htmlKona classes for attributes ONBLUR, ONCHANGE, ONLOAD, ONUNLOAD, etc.

Using a ScriptElement

A ScriptElement can be added to the head portion of a document, after calling the HtmlPage.getHead() method. A ScriptElement can also be added with the addElement() methods in several classes.

Example

This Netscape example shows a simple JavaScript script to be executed as part of a form with the GET action. (Note that Netscape serverside Java doesn't support package names.) The ScriptElement is added after calling the HtmlPage.getHead() method. This example uses a ScriptElement to evaluate an expression and return an answer:

```
import netscape.server.applet.HttpApplet;
import java.io.*;
import weblogic.html.*;

public class adder extends HttpApplet {

  public synchronized void run() throws IOException {
    try {
      // Housekeeping for a Netscape serverside
      // Java class
      this.returnNormalResponse("text/html");

      // Construct the page
      ServletPage hp = new ServletPage("Example 2d");

      // Construct a FormElement for user input
      FormElement form = new FormElement("", "GET");

      // Use a table to line up the elements on the form
      // and add it to the form
      TableElement tab = new TableElement();
      form.addElement(tab);

      // Populate the table with InputElements. We create
      // a row with 3 cells
      tab.addElement(new TableRowElement()
              .addElement(new InputElement("input")
                        .setValue("9 + 3")
                        .setSize(20)
                        .setMaxlen(20))
              .addElement(new InputElement("button1",
                          FieldType.button)
                        .setValue("Evaluate")
                        .setOnClick("compute(this.form)"))
               .addElement(new InputElement("answer")
                        .setValue("The answer is: 12")
                        .setSize(20)
                        .setMaxlen(20)));
      // Use the getBodyElement() method to set
      // attributes for the BODY tag
      hp.getBodyElement()
        .setAttribute(BodyElement.bgColor, HtmlColor.white);

      // Call getBody() to add various elements to
      // the page, including the ScriptElement.
      hp.getBody()
        .addElement(MarkupElement.HorizontalRule)
        .addElement(
   new ScriptElement(
```

```
            "document.write(\"This is a simple JavaScript "
                    + "expression evaluator.\")"))
         .addElement(MarkupElement.BeginParagraph)
         .addElement(
   new ScriptElement(
            "document.write(\"Enter an expression on the " +
                  "left and click on Evaluate.\")"))</font><font
color=#A93B6F face="Courier New">

// Note that we use "\n" to force newlines in the
// ScriptElement, which makes the JavaScript
// line up nicely.
         .addElement(new ScriptElement(
   "function compute(form) {\n" +
            "  if (confirm(\"Evaluate Expression \" +
                  form.input.value + \" ?\"))\n" +
            "    form.answer.value = \"The answer is: \" +
                  eval(form.input.value);\n" +
            "  else\n" +
            "    alert(\"Try again and choose OK.\");\n" +
            "}"))

// Finally, add the form to the page
         .addElement(form)
         .addElement(MarkupElement.HorizontalRule);

      // We invoke the ServletPage.output() method with the
      // results of HttpApplet's getOutputStream() method
      hp.output(getOutputStream());
    }
    catch (Exception e) {
      e.printStackTrace());
    }
  }
}
```

The documentation for the Netscape HttpApplet classes can be found at Netscape's <a href=http://developer.netscape.com/viewsource/index_frame.html?content=husted_js / site.

Adding an applet to a page

Classes for applets:

- weblogic.html.AppletElement

- weblogic.html.AlignType
- weblogic.html.ParamElement
- weblogic.html.TextFlowElement

AppletElement

Use an AppletElement to set attributes for the applet and to describe where it should be placed on the htmlKona page. The three parameters required in an applet element, the CODE, WIDTH, and HEIGHT attributes, are used in the constructors. There are also methods in the AppletElement class for setting alignment of the applet, the code base (URI), name, HSPACE, VSPACE, and alternate text for the applet.

In addition, htmlKona supports the ARCHIVE browser extension that lets you specify a .zip file to be downloaded to the user's disk to speed applet loading. Note that the .zip file is looked for relative to the CODEBASE for the applet, and that the .zip file **must not be compressed**. Classes needed by the applet that are not included in the .zip file will be searched for by traditional methods.

htmlKona also supports the HTML3.2 TEXTFLOW element within the applet tag, which is used to set alternate text that will be displayed if the applet cannot load or run. Note that the HTML specifications 2.0 and later require that you set alternate text for your applet. You may use the AppletElement.setAlt() method if desired, or you may add a TextFlowElement object with AppletElement.addElement(). The latter is recommended.

To set parameters for the applet, add ParamElements to the AppletElement with the AppletElement.addElement() method. A ParamElement is a name/value pair. ParamElements added to an AppletElement are accessible by index position with the getElementAt() method. Note that parameter names are case sensitive.

Use an AlignType object as an argument for the AppletElement object to set the applet's alignment on the page. AlignType objects are also used to set alignment for other objects.

Example

This example shows a Java Server-style servlet that loads the BlinkingText element from JavaSoft. This class file can be found in the distribution in weblogic/examples/htmlkona/. The showException method is part of another class in the distribution (defaults).

```
package examples.htmlkona;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import weblogic.html.*;

public class appletexample extends HttpServlet {

  public synchronized void service(ServletRequest req,
                                   ServletResponse res)
    throws IOException
  {
    try {
      // Housekeeping for Java Server servlets
      res.setStatus(ServletResponse.SC_OK);
      res.setContentType("text/html");
      res.writeHeaders();

      // Construct the AppletElement
      AppletElement applet =
        new AppletElement("Blink.class",
                          300,
                  100,
  AlignType.center);
      applet.setCodeBase("http://www.weblogic.com/classes/")
            .addElement(new ParamElement("lbl ",
 "This is the next best thing to sliced bread! "
 + "Toast, toast, toast, butter, jam, toast, "
          + "marmite, toast."))
            .addElement(new ParamElement("speed", "4"));

      // Create a Servletpage and set its attributes
      ServletPage hp = new ServletPage("Blinking Text Applet");
      hp.getBodyElement()
.setAttribute(BodyElement.bgColor, HtmlColor.white);

      // Add the AppletElement to the page
      hp.getBody()
        .addElement(new HeadingElement("This is the Blinking " +
                                 "Text Applet from Sun"))
        .addElement(applet)
.addElement(MarkupElement.HorizontalRule);

     hp.output(res.getOutputStream());
    }
    catch (Exception e) {
      defaults.showException(e, res.getOutputStream());
    }
```

```
    }
}
```

Embedding a file on a page

Classes for embedding objects:

- weblogic.html.EmbedElement

htmlKona supports the EMBED HTML element, which allows you to embed an arbitrary object directly into a page. Embedded objects are supported in some browsers by application-specific plug-ins. htmlKona also supports the HIDDEN, AUTOSTART, and NAME attributes for EmbedElements, as well as standard attributes (SRC, HEIGHT, and WIDTH).

Different browsers treat embedded objects in different ways. You should understand the behavior of your target browser when using this extension.

Using htmlKona to display dynamic data

One of the most powerful uses of htmlKona is to build pages that can display dynamic data. With htmlKona and WebLogic's database connectivity products like dbKona, the WebLogic JDBC Server, and the two-tier jdbcKona native JDBC drivers, you can build a web application that can handle any kind of input and output information and that can present information dynamically.

- dbKona
- WebLogic JDBC Options
- WebLogic JDBC Server

Here is a simple Netscape example that displays information from records retrieved from an employee database into the dbKona QueryDataSet "qs". This example uses WebLogic JDBC, a Java-only implementation of JDBC, to connect to an Oracle database through WebLogic. Information for the connections between the WebLogic

JDBC client, the WebLogic JDBC Server, and the Oracle database are handled by methods in the defaults class. To run the examples, you need to edit this file and recompile the directory. Here are the methods used from the defaults class:

```
import netscape.server.applet.HttpApplet;

import java.io.*;
import java.util.*;
import xjava.sql.*;
import weblogic.db.xjdbc.*;
import weblogic.common.*;
import weblogic.html.*;

public class defaults {

  public static Connection login(T3Client t3)
       throws Exception
    {

    // Sets parameters for the connection between
    // the WebLogic JDBC Server and the DBMS
    // (the two-tier connection).
    Properties dbprops = new Properties();
    dbprops.put("user",              "scott");
    dbprops.put("password",          "tiger");
    dbprops.put("server",            defaults.server());

    // Sets parameters for the connection between
    // the T3Client, the WebLogic JDBC Server, and
    // the DBMS (the multitier connection).
    Properties t3props = new Properties();
    t3props.put("weblogic.t3",       t3);
    t3props.put("weblogic.t3.dbprops",dbprops);

    // Sets the class name and URL of the two-tier
    // JDBC driver for the connection between the
    // WebLogic Server and the DBMS.
    t3props.put("weblogic.t3.driver","weblogic.jdbc.oci.Driver");
    t3props.put("weblogic.t3.url",   "jdbc:weblogic:oracle");

    // Sets the class name and URL of the multitier
    // (Java-only) driver for the connection between
    // the T3Client, WebLogic Server and the DBMS.
    Class.forName("weblogic.jdbc.t3client.Driver")
      .newInstance();
    Connection conn =
      DriverManager.getConnection("jdbc:weblogic:t3client",
                                  t3props);
```

```
    return conn;
  }

  // Specifies the URL for the WebLogic Server that
  // will be listening for T3Client requests.
  public static final String t3clienturl()  {
    return (String)System.getProperty("t3url",
                                      "t3://localhost:7001");
  }

  // Provides connection information to your database.
  public static final String server()  {
    return (String)System.getProperty("server", "DEMO");
  }

  // Outputs a stack trace onto an HTML page when
  // there is an exception.
  public static void showException(Exception e,
                                   OutputStream out)
     throws IOException
    {
    ServletPage hp = new ServletPage("An Exception occurred");
    ByteArrayOutputStream ostr = new ByteArrayOutputStream();
    e.printStackTrace(new PrintStream(ostr));
    hp.getBody()
     .addElement(new HeadingElement("Exception occurred:", 2))
     .addElement(new LiteralElement(ostr.toString()));
    hp.output(out);
  }

  // Sets defaults for htmlKona page generation.
  public static void setPageDefaults() {
    TableCaptionElement.defaultAlign= AlignType.top;
    TableElement.defaultCaption     = new TableCaptionElement("");
    TableElement.defaultBorder      = 4;
    TableElement.defaultCellspacing = 2;
    TableElement.defaultCellpadding = 2;
    TableElement.defaultWidth       = "100%";
  }
}
```

The example uses methods in the defaults class to get information for DBMS connections, page defaults, and to show exceptions. This example, uses a WebLogic JDBC Client to connect to the DBMS through WebLogic's JDBC Server. WebLogic JDBC requires no client-side libraries, and, consequently, is appropriate for use in applets, although it is used in this example for serverside Java. Because dbKona is a higher-level abstraction that doesn't depend on vendor-specific constructs of data, you can use dbKona to write programs that can retrieve data from any SQL database in a

very consistent way. You could adapt this example for use with Sybase or Microsoft SQL Server by doing nothing more than changing the connection information in the defaults class.

This example, uses serverside Java to connect to a database and display information from it. Several records are inserted, modified, and deleted from a database, using dbKona DataSets, which provide automatic generation of SQL and Query-by-example (QBE) functionality.

```
import netscape.server.applet.HttpApplet;

import java.io.*;
import java.util.*;
import weblogic.db.xjdbc.*;
import weblogic.html.*;
import xjava.sql.*;

public class example5 extends HttpApplet {

  public synchronized void run() throws IOException  {
    T3Client   t3   = null;
    Connection conn = null;

    try {
      // Housekeeping for Netscape serverside Java
      this.returnNormalResponse("text/html");

      // Connection is handled by methods in the default class
      t3 = new T3Client(defaults.t3clienturl());
      t3.connect();
      defaults.setPageDefaults();
      conn = defaults.login(t3);

      // Create 3 SQL statements
      Statement stmt = conn.createStatement();
     String insert = "insert into emp(empno, ename, job, deptno) " +
                     "values (8000, 'MURPHY', 'SALESMAN', 10)";
      String update = "update emp set ename = 'SMITH', " +
                      "job = 'MANAGER' where empno = 8000";
      String delete = "delete from emp where empno = 8000";

      // Execute the 1st statement and verify results
      stmt.execute(insert);
      TableDataSet ds1 = new TableDataSet(conn, "emp");
      ds1.where("empno = 8000")
         .fetchRecords();

      // Execute the 2nd statement and verify results
```

```
            stmt.execute(update);
            TableDataSet ds2 = new TableDataSet(conn, "emp");
            ds2.where("empno = 8000")
               .fetchRecords();

            // Execute the 3rd statement and verify results
            stmt.execute(delete);
            TableDataSet ds3 = new TableDataSet(conn, "emp");
            ds2.where("empno = 8000")
               .fetchRecords();

            // Construct the page and set its attributes
            ServletPage hp = new ServletPage("Example 5");
            hp.getBodyElement()
              .setAttribute(BodyElement.bgColor, HtmlColor.white);
            hp.getBody()
              .addElement(MarkupElement.HorizontalLine)
              .addElement(new HeadingElement("INSERT results", 2))
              .addElement(new HeadingElement("Using SQL:", 3))
              .addElement(new LiteralElement(insert))
              .addElement(new LiteralElement(ds1))
              .addElement(MarkupElement.HorizontalLine)
              .addElement(new HeadingElement("UPDATE results", 2))
              .addElement(new HeadingElement("Using SQL:", 3))
              .addElement(new LiteralElement(update))
              .addElement(new LiteralElement(ds2))
              .addElement(MarkupElement.HorizontalLine)
              .addElement(new HeadingElement("DELETE results", 2))
              .addElement(new HeadingElement("Using SQL:", 3))
              .addElement(new LiteralElement(delete))
              .addElement(new LiteralElement(ds3))
              .addElement(MarkupElement.HorizontalRule)
        .addElement("Copyright 1996-98, BEA Systems Inc.");

            hp.output(getOutputStream());

            // Close the DataSets
            ds1.close();
            ds2.close();
            ds3.close();
          }
          catch (Exception e) {
            defaults.showException(e, getOutputStream());
          }

          // Close the connections and disconnect
          finally {
            try {conn.close();}    catch (Exception e) {;}
            try {t3.disconnect();} catch (Exception e) {;}
```

```
    }
  }
}
```

Using htmlKona with Java-enabled servers

There are currently a few HTTP servers that support serverside Java; that is, the server can run a Java class file. JavaSoft's Java WebServer, the Netscape Enterprise and Fast Track servers, Oracle's WebServer, and the WebLogic Server are among HTTP servers that support serverside Java.

WebLogic can execute a Java class file that subclasses javax.servlet.http.HttpServlet (or weblogic.html.FormServlet) as a response to an HTTP request; WebLogic, when a dochome is configured, can serve arbitrary files like HTML pages, images, applets, and archives, and can also *proxy* requests to another HTTP server.

htmlKona classes are most commonly executed as server-side Java. This is covered here in terms of using htmlKona with WebLogic. This means that you have compiled your htmlKona .java file into a class file which you register in your weblogic.properties file and place in your WebLogic Server host's CLASSPATH. When a user requests that class file (with a URL), the class file is executed and its results are returned as a response to the HTTP request.

Writing a class with htmlKona that is destined to be used as server-side Java means that you must structure the class file in a way that is appropriate to the HTTP server that will respond to a request for the class file. WebLogic supports the Java standard servlet API, and htmlKona is optimized for such a use. This means importing the proper classes and writing classes that match the request/response pattern that the HTTP server expects.

Standard Java Servlet API example

Here is a general outline of how you implement for server-side Java using the Java Servlet API. WebLogic serves servlets that conform to this API, specifically subclasses of javax.servlet.http.HttpServlet. All of the forms and interactive pages on WebLogic's website use servlets like these, running on WebLogic.

A little history: WebLogic has supported the Java Servlet API since its inception as Jeeves (versions A1.2 and A2). In version 1.0 of the Java Servlet API, JavaSoft changed the servlet package names to begin **javax**. If you are upgrading your servlets for use with WebLogic version 2.5 or later, you will need to change the package names from java.servlet to javax.servlet.

You may also be subclassing FormServlet. Long before web servers supported servlets, htmlKona supported a class called FormServlet. When Jeeves version A2 was released, Jeeves had support for a subclass of HttpServlet called FormServlet, and at that time, WebLogic dropped its own FormServlet to support the Javasoft standard. But version 1.0 of the Servlet API dropped support for FormServlet. So, in order to continue to provide support for those who might have existing classes that use FormServlet functionality, FormServlet was resurrected and reinstated it in the htmlKona API. In release 3.0, FormServlet was deprecated in favor of form-related functionality that was available in HttpServlet.

If you aren't using FormServlet already in your classes, there is no need to start. All of the functionality is readily available in the more general HttpServlet. (Specifically you can implement the doGet() and doPost() method to process the query, or you can override the service() method, which dispatches the request to either doGet() or doPost(), depending on the request. The doGet() and doPost() methods are implemented as no-ops in HttpServlet.) For that reason, the FormServlet class has been deprecated, although WebLogic will continue to support it for backwards compatibility.

- Import javax.servlet.*, javax.servlet.http.*, and weblogic.html.*.
- Your class should extend javax.servlet.http.HttpServlet.
- Your class should implement the HttpServlet.service() method (which takes two arguments: an HttpServletRequest object and an HttpServletResponse object), or—if you will be using form data that you want to post back to the server—your class should implement the HttpServlet.doGet() or HttpServlet.doPost()method to process the query.
- Call the setStatus() method on the HttpServletResponse object with one of the final static ints in this class to indicate request status.
- Call the setContentType() method on the ServletResponse object with the content-type of the page, usually "text/html".

- Use a weblogic.html.ServletPage as your htmlKona canvas. The default codeset is “8859_1". You can set an alternate codeset by using another ServletPage constructor, which takes a Codeset object as an argument, as shown here:

  ```
  ServletPage sp = new ServletPage(new Codeset("SJIS"));
  ```

- Use the results of the HttpServletResponse.getOutputStream() method as an argument for the output() method you call on the ServletPage.

```
package examples.htmlkona;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import weblogic.html.*;

public class example1 extends HttpServlet {

  public synchronized void service(HttpServletRequest req,
                                   HttpServletResponse res)
    throws IOException
  {
    // Set the status and content type on the response object
    res.setStatus(HttpServletResponse.SC_OK);
    res.setContentType("text/html");

    // Create a servlet page and add one string to it
    ServletPage sp = new ServletPage("Example 1");
    sp.getBody().addElement("Hello world!");

    // Call the page's output method with the response
    // object's output stream
    sp.output(res.getOutputStream());
  }
}
```

Shortcuts for testing output

htmlKona is designed to be used in with a Java-enabled HTTP server, like the WebLogic Server.

One of the drawbacks of developing programs for serverside Java is that you must restart the server each time you compile a class, since each class is loaded only once (the first time it is requested). When you are testing and tweaking, you may prefer to output your htmlKona class into an HTML file that you can view for testing without a server.

You can use the Java compiler to generate the HTML that your code will produce, direct the output into a file, and then view that file in a browser to check layout, colors, etc. (Of course, you cannot test user input within forms without a server.) To output your htmlKona class into a file:

1. Create a .java file with your htmlKona code. Your class should contain public static void main(String[] argv), and the HtmlPage.output() method should take no arguments.
2. Compile it with javac from the command line. If you are using packages, you should provide the full package name, as in calculator.adder or any of the examples/htmlkona classes. Make sure you have set the proper CLASSPATH in the shell in which you run javac.
3. Copy the resulting .class file into a directory in your Java CLASSPATH. Or direct the output of the javac command into the appropriate directory with the -d option.
4. Invoke the .class file with java from the command line. If you have invoked the output() method without an argument, the HTML will be output to stout. You can direct the output into a file, as in java examples.htmlkona.HelloWorld > test.html.
5. View the HTML file in a browser. There will be an extra line at the top of the file for “Content-type” which is normally passed to the HTTP server; you can ignore this line in your testing, since it will not appear in a page served by HTTP.