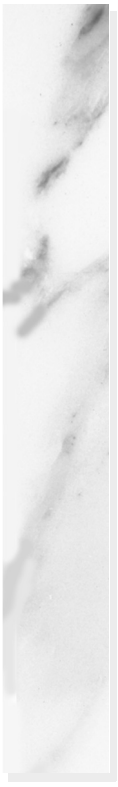
SERVLET AND JSP QUICK REFERENCE



Appendix A

A.1 Overview of Servlets and JavaServer Pages

Advantages of Servlets

- **Efficient**: threads instead of OS processes, one servlet copy, persistence
- **Convenient**: lots of high-level utilities
- **Powerful**: talking to server, sharing data, pooling, persistence
- **Portable**: run on virtually all operating systems and servers
- **Secure**: no shell escapes, no buffer overflows
- **Inexpensive**: inexpensive plug-ins if servlet support not bundled

Advantages of JSP

- **Versus ASP**: better language for dynamic part, portable
- **Versus PHP**: better language for dynamic part
- **Versus pure servlets**: more convenient to create HTML
- **Versus SSI**: much more flexible and powerful
- **Versus JavaScript**: server-side, richer language
- **Versus static HTML**: dynamic features

Free Servlet and JSP Software

- **Tomcat**: http://jakarta.apache.org/

- **JSWDK**: http://java.sun.com/products/servlet/download.html
- **JRun**: http://www.allaire.com/products/jrun/
- **ServletExec**: http://newatlanta.com/
- **LiteWebServer**: http://www.gefionsoftware.com/
- **Java Web Server**: http://www.sun.com/software/jwebserver/try/

Documentation

- http://java.sun.com/products/jsp/download.html
- http://java.sun.com/products/servlet/2.2/javadoc/
- http://www.java.sun.com/j2ee/j2sdkee/techdocs/api/

Servlet Compilation: CLASSPATH Entries

- The servlet classes (usually in install_dir/lib/servlet.jar)
- The JSP classes (usually in install_dir/lib/jsp.jar, ...jspengine.jar, or ...jasper.jar)
- The top-level directory of servlet installation directory (e.g., install_dir/webpages/WEB-INF/classes)

Tomcat 3.0 Standard Directories

- **install_dir/webpages/WEB-INF/classes**
  Standard location for servlet classes.
- **install_dir/classes**
  Alternate location for servlet classes.
- **install_dir/lib**
  Location for JAR files containing classes.

Tomcat 3.1 Standard Directories

- **install_dir/webapps/ROOT/WEB-INF/classes**
  Standard location for servlet classes.
- **install_dir/classes**
  Alternate location for servlet classes.
- **install_dir/lib**
  Location for JAR files containing classes.

JSWDK 1.0.1 Standard Directories

- **install_dir/webpages/WEB-INF/servlet**s
  Standard location for servlet classes.
- **install_dir/classes**
  Alternate location for servlet classes.
- **install_dir/lib**
  Location for JAR files containing classes.

Java Web Server 2.0 Standard Directories

- **install_dir/servlets**
  Location for frequently changing servlet classes. Auto-reloading.
- **install_dir/classes**
  Location for infrequently changing servlet classes.
- **install_dir/lib**
  Location for JAR files containing classes.

A.2 First Servlets

Simple Servlet

HelloWWW.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Installing Servlets

- Put in servlet directories shown in Section A.1.
- Put in subdirectories corresponding to their package.

Invoking Servlets

- http://host/servlet/ServletName
- http://host/servlet/package.ServletName
- Arbitrary location defined by server-specific customization.

Servlet Life Cycle

- **public void init() throws ServletException,**
  public void init(ServletConfig config) throws ServletException
  Executed once when the servlet is first loaded. *Not* called for each request. Use `getInitParameter` to read initialization parameters.
- **public void service(HttpServletRequest request,**
  HttpServletResponse response)
  throws ServletException, IOException
  Called in a new thread by server for each request. Dispatches to `doGet`, `doPost`, etc. Do not override this method!
- **public void doGet(HttpServletRequest request,**
  HttpServletResponse response)
  throws ServletException, IOException
  Handles `GET` requests. Override to provide your behavior.
- **public void doPost(HttpServletRequest request,**
  HttpServletResponse response)
  throws ServletException, IOException
  Handles `POST` requests. Override to provide your behavior. If you want `GET` and `POST` to act identically, call `doGet` here.
- **doPut**, **doTrace**, **doDelete**, etc.
  Handles the uncommon HTTP requests of `PUT`, `TRACE`, etc.
- **public void destroy()**
  Called when server deletes servlet instance. *Not* called after each request.
- **public long getLastModified(HttpServletRequest request)**
  Called by server when client sends conditional `GET` due to cached copy. See Section 2.8.
- **SingleThreadModel**
  If this interface implemented, causes server to avoid concurrent invocations.

A.3 Handling the Client Request: Form Data

Reading Parameters

- **request.getParameter**: returns first value
- **request.getParameterValues**: returns array of all values

Example Servlet

ThreeParams.java

```
package coreservlets;

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

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading Three Request Parameters";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI><B>param1</B>: "
                + request.getParameter("param1") + "\n" +
                "  <LI><B>param2</B>: "
                + request.getParameter("param2") + "\n" +
                "  <LI><B>param3</B>: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Example Form

ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Collecting Three Parameters</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
  First Parameter:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Second Parameter: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Third Parameter:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

Filtering HTML-Specific Characters

- Must replace <, >, ", & with `<`, `>`, `&quot;`, and `&`. Use `ServletUtilities.filter(htmlString)` for this substitution. See Section 3.6.

A.4 Handling the Client Request: HTTP Request Headers

Methods That Read Request Headers

These are all methods in `HttpServletRequest`.

- **public String getHeader(String headerName)**
  Returns value of an arbitrary header. Returns `null` if header not in request.
- **public Enumeration getHeaders(String headerName)**
  Returns values of all occurrences of header in request. 2.2 only.
- **public Enumeration getHeaderNames()**
  Returns names of all headers in current request.
- **public long getDateHeader(String headerName)**
  Reads header that represents a date and converts it to Java date format (milliseconds since 1970).

- **public int getIntHeader(String headerName)**
  Reads header that represents an integer and converts it to an `int`. Returns -1 if header not in request. Throws `NumberFormatException` for non-ints.
- **public Cookie[] getCookies()**
  Returns array of `Cookie` objects. Array is zero-length if no cookies. See Chapter 8.
- **public int getContentLength()**
  Returns value of `Content-Length` header as `int`. Returns -1 if unknown.
- **public String getContentType()**
  Returns value of `Content-Type` header if it exists in request (i.e., for attached files) or `null` if not.
- **public String getAuthType()**
  Returns `"BASIC"`, `"DIGEST"`, `"SSL"`, or `null`.
- **public String getRemoteUser()**
  Returns username if authentication used; `null` otherwise.

Other Request Information

- **public String getMethod()**
  Returns HTTP request method (`"GET"`, `"POST"`, `"HEAD"`, etc.)
- **public String getRequestURI()**
  Returns part of the URL that came after host and port.
- **public String getProtocol()**
  Returns HTTP version (`"HTTP/1.0"` or `"HTTP/1.1"`, usually).

Common HTTP 1.1 Request Headers

See RFC 2616. Get RFCs on-line starting at http://www.rfc-editor.org/.

- **Accept**: MIME types browser can handle.
- **Accept-Encoding**: encodings (e.g., gzip or compress) browser can handle. See compression example in Section 4.4.
- **Authorization**: user identification for password-protected pages. See example in Section 4.5. Normal approach is to not use HTTP authorization but instead use HTML forms to send username/password and then for servlet to store info in session object.
- **Connection**: In HTTP 1.0, `keep-alive` means browser can handle persistent connection. In HTTP 1.1, persistent connection is default. Servlets should set `Content-Length` with `setContentLength` (using `ByteArrayOutputStream` to determine length of output) to support persistent connections. See example in Section 7.4.

- **Cookie**: cookies sent to client by server sometime earlier. Use `getCookies`, not `getHeader`. See Chapter 8.
- **Host**: host given in original URL. This is a required header in HTTP 1.1.
- **If-Modified-Since**: indicates client wants page only if it has been changed after specified date. Don't handle this situation directly; implement `getLastModified` instead. See example in Section 2.8.
- **Referer**: URL of referring Web page.
- **User-Agent**: string identifying the browser making the request.

A.5 Accessing the Standard CGI Variables

You should usually think in terms of request info, response info, and server info, not CGI variables.

Capabilities Not Discussed Elsewhere

- **getServletContext().getRealPath("uri")**: maps URI to real path
- **request.getRemoteHost()**: name of host making request
- **request.getRemoteAddress()**: IP address of host making request

Servlet Equivalent of CGI Variables

- AUTH_TYPE: `request.getAuthType()`
- CONTENT_LENGTH: `request.getContentLength()`
- CONTENT_TYPE: `request.getContentType()`
- DOCUMENT_ROOT: `getServletContext().getRealPath("/")`
- HTTP_XXX_YYY: `request.getHeader("Xxx-Yyy")`
- PATH_INFO: `request.getPathInfo()`
- PATH_TRANSLATED: `request.getPathTranslated()`
- QUERY_STRING: `request.getQueryString()`
- REMOTE_ADDR: `request.getRemoteAddr()`
- REMOTE_HOST: `request.getRemoteHost()`
- REMOTE_USER: `request.getRemoteUser()`
- REQUEST_METHOD: `request.getMethod()`
- SCRIPT_NAME: `request.getServletPath()`
- SERVER_NAME: `request.getServerName()`
- SERVER_PORT: `request.getServerPort()`
- SERVER_PROTOCOL: `request.getProtocol()`
- SERVER_SOFTWARE: `getServletContext().getServerInfo()`

A.6 Generating the Server Response: HTTP Status Codes

Format of an HTTP Response

Status line (HTTP version, status code, message), response headers, blank line, document, in that order. For example:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World
```

Methods That Set Status Codes

These are methods in `HttpServletResponse`. Set status codes before you send any document content to browser.

- **public void setStatus(int statusCode)**
  Use a constant for the code, not an explicit `int`.
- **public void sendError(int code, String message)**
  Wraps message inside small HTML document.
- **public void sendRedirect(String url)**
  Relative URLs permitted in 2.2.

Status Code Categories

- **100-199**: informational; client should respond with some other action.
- **200-299**: request was successful.
- **300-399**: file has moved. Usually includes a Location header indicating new address.
- **400-499**: error by client.
- **500-599**: error by server.

Common HTTP 1.1 Status Codes

- **200 (OK)**: Everything is fine; document follows. Default for servlets.
- **204 (No Content)**: Browser should keep displaying previous document.
- **301 (Moved Permanently)**: Requested document permanently moved elsewhere (indicated in `Location` header). Browsers go to new location automatically.
- **302 (Found)**: Requested document temporarily moved elsewhere (indicated in Location header). Browsers go to new location automatically. Servlets should use `sendRedirect`, not `setStatus`, when setting this header. See example in Section 6.3.

- **401 (Unauthorized)**: Browser tried to access password-protected page without proper `Authorization` header. See example in Section 4.5.
- **404 (Not Found)**: No such page. Servlets should use `sendError` to set this header. See example in Section 6.3.

A.7 Generating the Server Response: HTTP Response Headers

Setting Arbitrary Headers

These are methods in `HttpServletResponse`. Set response headers before you send any document content to browser.

- **public void setHeader(String headerName, String headerValue)**
  Sets an arbitrary header.
- **public void setDateHeader(String headerName, long milliseconds)**
  Converts milliseconds since 1970 to a date string in GMT format.
- **public void setIntHeader(String headerName, int headerValue)**
  Prevents need to convert `int` to `String` before calling `setHeader`.
- **addHeader**, **addDateHeader**, **addIntHeader**
  Adds new occurrence of header instead of replacing. 2.2 only.

Setting Common Headers

- **setContentType**: Sets the `Content-Type` header. Servlets almost always use this. See Table 7.1 for the most common MIME types.
- **setContentLength**: Sets the `Content-Length` header. Used for persistent HTTP connections. Use `ByteArrayOutputStream` to buffer document before sending it, to determine size. See Section 7.4 for an example.
- **addCookie**: Adds a value to the `Set-Cookie` header. See Chapter 8.
- **sendRedirect**: Sets the `Location` header (plus changes status code). See example in Section 6.3.

Common HTTP 1.1 Response Headers

- **Allow**: the request methods server supports. Automatically set by the default `service` method when servlet receives `OPTIONS` requests.
- **Cache-Control**: A `no-cache` value prevents browsers from caching results. Send `Pragma` header with same value in case browser only understands HTTP 1.0.
- **Content-Encoding**: the way document is encoded. Browser reverses this encoding before handling document. Servlets must confirm that

browser supports a given encoding (by checking the `Accept-Encoding` request header) before using it. See compression example in Section 4.4.

- **Content-Length**: the number of bytes in the response. See `setContentLength` above.
- **Content-Type**: the MIME type of the document being returned. See `setContentType` above.
- **Expires**: the time at which document should be considered out-of-date and thus should no longer be cached. Use `setDateHeader` to set this header.
- **Last-Modified**: the time document was last changed. Don't set this header explicitly; provide a `getLastModified` method instead. See example in Section 2.8.
- **Location**: the URL to which browser should reconnect. Use `sendRedirect` instead of setting this header directly. For an example, see Section 6.3.
- **Pragma**: a value of `no-cache` instructs HTTP 1.0 clients not to cache document. See the `Cache-Control` response header (Section 7.2).
- **Refresh**: the number of seconds until browser should reload page. Can also include URL to connect to. For an example, see Section 7.3.
- **Set-Cookie**: the cookies that browser should remember. Don't set this header directly; use `addCookie` instead. See Chapter 8 for details.
- **WWW-Authenticate**: the authorization type and realm client should supply in its `Authorization` header in the next request. For an example, see Section 4.5.

Generating GIF Images from Servlets

- **Create an Image.**
  Use the `createImage` method of `Component`.
- **Draw into the Image.**
  Call `getGraphics` on the `Image`, then do normal drawing operations.
- **Set the Content-Type response header.**
  Use `response.setContentType("image/gif")`.
- **Get an output stream.**
  Use `response.getOutputStream()`.
- **Send the Image down output stream in GIF format.**
  Use Jef Poskanzer's `GifEncoder`. See http://www.acme.com/java/.

A.8 Handling Cookies

Typical Uses of Cookies

- Identifying a user during an e-commerce session
- Avoiding username and password
- Customizing a site
- Focusing advertising

Problems with Cookies

- It's a privacy problem, not a security problem.
- Privacy problems include: servers can remember what you did in previous sessions; if you give out personal information, servers can link that information to your previous actions; servers can share cookie information through use of a cooperating third party like doubleclick.net (by each loading image off the third-party site); poorly designed sites could store sensitive information like credit card numbers directly in the cookie.

General Usage

- **Sending cookie to browser (standard approach):**

```
Cookie c = new Cookie("name", "value");
c.setMaxAge(...);
// Set other attributes.
response.addCookie(c);
```

- **Sending cookie to browser (simplified approach):**
  Use `LongLivedCookie` class (Section 8.5).
- **Reading cookies from browser (standard approach):**

```
Cookie[] cookies = response.getCookies();
for(int i=0; i<cookies.length; i++) {
  Cookie c = cookies[i];
  if (c.getName().equals("someName")) {
    doSomethingWith(c);
    break;
  }
}
```

- **Reading cookies from browser (simplified approach):**
  Extract cookie or cookie value from cookie array by using `ServletUtilities.getCookie` or `ServletUtilities.getCookieValue`.

Cookie Methods

- **getComment/setComment**: gets/sets comment. Not supported in version 0 cookies (which is what most browsers now support).

- **getDomain/setDomain**: lets you specify domain to which cookie applies. Current host must be part of domain specified.
- **getMaxAge/setMaxAge**: gets/sets the cookie expiration time (in seconds). If you fail to set this, cookie applies to current browsing session only. See `LongLivedCookie` helper class (Section 8.5).
- **getName/setName**: gets/sets the cookie name. For new cookies, you supply name to constructor, not to `setName`. For incoming cookie array, you use `getName` to find the cookie of interest.
- **getPath/setPath**: gets/sets the path to which cookie applies. If unspecified, cookie applies to URLs that are within or below directory containing current page.
- **getSecure/setSecure**: gets/sets flag indicating whether cookie should apply only to SSL connections or to all connections.
- **getValue/setValue**: gets/sets value associated with cookie. For new cookies, you supply value to constructor, not to `setValue`. For incoming cookie array, you use `getName` to find the cookie of interest, then call `getValue` on the result.
- **getVersion/setVersion**: gets/sets the cookie protocol version. Version 0 is the default; stick with that until browsers start supporting version 1.

A.9 Session Tracking

Looking Up Session Information: getValue

```
HttpSession session = request.getSession(true);
ShoppingCart cart =
  (ShoppingCart)session.getValue("shoppingCart");
if (cart == null) { // No cart already in session
  cart = new ShoppingCart();
  session.putValue("shoppingCart", cart);
}
doSomethingWith(cart);
```

Associating Information with a Session: putValue

```
HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart cart =
  (ShoppingCart)session.getValue("previousItems");
if (cart == null) { // No cart already in session
  cart = new ShoppingCart();
  session.putValue("previousItems", cart);
}
String itemID = request.getParameter("itemID");
if (itemID != null) {
  cart.addItem(Catalog.getItem(itemID));
}
```

HttpSession Methods

- **public Object getValue(String name) [2.1]**
  public Object getAttribute(String name) [2.2]
  Extracts a previously stored value from a session object. Returns `null` if no value is associated with given name.
- **public void putValue(String name, Object value) [2.1]**
  public void setAttribute(String name, Object value) [2.2]
  Associates a value with a name. If value implements `HttpSessionBindingListener`, its `valueBound` method is called. If previous value implements `HttpSessionBindingListener`, its `valueUnbound` method is called.
- **public void removeValue(String name) [2.1]**
  public void removeAttribute(String name) [2.2]
  Removes any values associated with designated name. If value being removed implements `HttpSessionBindingListener`, its `valueUnbound` method is called.
- **public String[] getValueNames() [2.1]**
  public Enumeration getAttributeNames() [2.2]
  Returns the names of all attributes in the session.
- **public String getId()**
  Returns the unique identifier generated for each session.
- **public boolean isNew()**
  Returns `true` if the client (browser) has never seen the session; `false` otherwise.
- **public long getCreationTime()**
  Returns time at which session was first created (in milliseconds since 1970). To get a value useful for printing, pass value to `Date` constructor or the `setTimeInMillis` method of `GregorianCalendar`.
- **public long getLastAccessedTime()**
  Returns time at which the session was last sent from the client.
- **public int getMaxInactiveInterval()**
  public void setMaxInactiveInterval(int seconds)
  Gets or sets the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that session should never time out. *Not* the same as cookie expiration date.
- **public void invalidate()**
  Invalidates the session and unbinds all objects associated with it.

Encoding URLs

In case servlet is using URL rewriting to implement session tracking, you should give the system a chance to encode the URLs.

- **Regular URLs**

```
String originalURL = someRelativeOrAbsoluteURL;
String encodedURL = response.encodeURL(originalURL);
out.println("<A HREF=\"" + encodedURL + "\">...</A>");
```

- **Redirect URLs**

```
String originalURL = someURL; // Relative URL OK in 2.2
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

A.10 JSP Scripting Elements

Types of Scripting Elements

- **Expressions: <%= expression %>**
  Evaluated and inserted into servlet's output. You can also use

```
<jsp:expression>
expression
</jsp:expression>
```

- **Scriptlets: <% code %>**
  Inserted into servlet's `_jspService` method (called by `service`). You can also use

```
<jsp:scriptlet>
code
</jsp:scriptlet>
```

- **Declarations: <%! code %>**
  Inserted into body of servlet class, outside of any existing methods. You can also use

```
<jsp:declaration>
code
</jsp:declaration>
```

Template Text

- Use `<\%` to get `<%` in output.
- `<%-- JSP Comment --%>`
- `<!-- HTML Comment -->`
- All other non-JSP-specific text passed through to output page.

Predefined Variables

Implicit objects automatically available in expressions and scriptlets (not declarations).

- **request**: the `HttpServletRequest` associated with request.

- **response**: the `HttpServletResponse` associated with response to client.
- **out**: the `JspWriter` (`PrintWriter` subclass) used to send output to the client.
- **session**: the `HttpSession` object associated with request. See Chapter 9.
- **application**: the `ServletContext` as obtained by `getServletConfig().getContext()`. Shared by all servlets and JSP pages on server or in Web application. See Section 15.1.
- **config**: the `ServletConfig` object for this page.
- **pageContext**: the `PageContext` object associated with current page. See Section 13.4 for a discussion of its use.
- **page**: synonym for `this` (current servlet instance); not very useful now. Placeholder for future.

A.11 The JSP page Directive: Structuring Generated Servlets

The import Attribute

- `<%@ page import="package.class" %>`
- `<%@ page import="package.class1,...,package.classN" %>`

The contentType Attribute

- `<%@ page contentType="MIME-Type" %>`
- `<%@ page contentType="MIME-Type; charset=Character-Set" %>`
- Cannot be invoked conditionally. Use `<% response.setContentType("..."); %>` for that.

Example of Using contentType

Excel.jsp

```
<%@ page contentType="application/vnd.ms-excel" %>
<%-- Note that there are tabs, not spaces, between columns. --%>
1997   1998   1999   2000   2001 (Anticipated)
12.3   13.4   14.5   15.6   16.7
```

Example of Using setContentType

ApplesAndOranges.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- HEAD part removed. -->
<BODY><CENTER><H2>Comparing Apples and Oranges</H2>

<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
  response.setContentType("application/vnd.ms-excel");
}
%>
<TABLE BORDER=1>
  <TR><TH></TH><TH>Apples<TH>Oranges
  <TR><TH>First Quarter<TD>2307<TD>4706
  <TR><TH>Second Quarter<TD>2982<TD>5104
  <TR><TH>Third Quarter<TD>3011<TD>5220
  <TR><TH>Fourth Quarter<TD>3055<TD>5287
</TABLE>

</CENTER></BODY></HTML>
```

The isThreadSafe Attribute

- `<%@ page isThreadSafe="true" %> <%!-- Default --%>`
- `<%@ page isThreadSafe="false" %>`
- A value of `true` means that *you* have made your code threadsafe and that the system can send multiple concurrent requests. A value of `false` means that the servlet resulting from JSP document will implement `SingleThreadModel` (see Section 2.6).
- Non-threadsafe code:

```
<%! private int idNum = 0; %>
<% String userID = "userID" + idNum;
   out.println("Your ID is " + userID + ".");
   idNum = idNum + 1; %>
```

- Threadsafe code:

```
<%! private int idNum = 0; %>
<% synchronized(this) {
     String userID = "userID" + idNum;
     out.println("Your ID is " + userID + ".");
     idNum = idNum + 1;
   } %>
```

The session Attribute

- `<%@ page session="true" %> <%!-- Default --%>`
- `<%@ page session="false" %>`

The buffer Attribute

- `<%@ page buffer="sizekb" %>`
- `<%@ page buffer="none" %>`
- Servers can use a larger buffer than you specify, but not a smaller one. For example, `<%@ page buffer="32kb" %>` means the document content should be buffered and not sent to the client until at least 32 kilobytes have been accumulated or the page is completed.

The autoflush Attribute

- `<%@ page autoflush="true" %> <%!-- Default --%>`
- `<%@ page autoflush="false" %>`
- A value of `false` is illegal when `buffer="none"` is also used.

The extends Attribute

- `<%@ page extends="package.class" %>`

The info Attribute

- `<%@ page info="Some Message" %>`

The errorPage Attribute

- `<%@ page errorPage="Relative URL" %>`
- The exception thrown will be automatically available to the designated error page by means of the `exception` variable. See Listings 11.5 and 11.6 for examples.

The isErrorPage Attribute

- `<%@ page isErrorPage="true" %>`
- `<%@ page isErrorPage="false" %> <%!-- Default --%>`
- See Listings 11.5 and 11.6 for examples.

The language Attribute

- `<%@ page language="cobol" %>`
- For now, don't bother with this attribute since `java` is both the default and the only legal choice.

XML Syntax

- **Usual syntax:**

```
<%@ page attribute="value" %>
<%@ page import="java.util.*" %>
```

- **XML equivalent:**

```
<jsp:directive.page attribute="value" />
<jsp:directive.page import="java.util.*" />
```

A.12 Including Files and Applets in JSP Documents

Including Files at Page Translation Time

- `<%@ include file="Relative URL" %>`
- Changing included file does not necessarily cause retranslation of JSP document. You have to manually change JSP document or update its modification date. Convenient way:

```
<%-- Navbar.jsp modified 3/1/00 --%>
<%@ include file="Navbar.jsp" %>
```

Including Files at Request Time

- `<jsp:include page="Relative URL" flush="true" />`
- Servlets can use `include` method of `RequestDispatcher` to accomplish similar result. See Section 15.3.
- Because of a bug, you must use .html or .htm extensions for included files used with the Java Web Server.

Applets for the Java Plug-In: Simple Case

- **Regular form:**

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
</APPLET>
```

- **JSP form for Java Plug-in:**

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
</jsp:plugin>
```

Attributes of jsp:plugin

All attribute names are case sensitive; all attribute values require single or double quotes.

- **type**: for applets, this attribute should have a value of `applet`.

- **code**: used identically to the CODE attribute of APPLET.
- **width**: used identically to the WIDTH attribute of APPLET.
- **height**: used identically to the HEIGHT attribute of APPLET.
- **codebase**: used identically to the CODEBASE attribute of APPLET.
- **align**: used identically to the ALIGN attribute of APPLET and IMG.
- **hspace**: used identically to the HSPACE attribute of APPLET.
- **vspace**: used identically to the VSPACE attribute of APPLET.
- **archive**: used identically to the ARCHIVE attribute of APPLET.
- **name**: used identically to the NAME attribute of APPLET.
- **title**: used identically to the rare TITLE attribute of APPLET (and virtually all other HTML elements in HTML 4.0), specifying a title that could be used for a tool-tip or for indexing.
- **jreversion**: identifies the version of the Java Runtime Environment (JRE) that is required. The default is 1.1.
- **iepluginurl**: designates a URL from which the plug-in for Internet Explorer can be downloaded.
- **nspluginurl**: designates a URL from which the plug-in for Netscape can be downloaded.

Parameters in HTML: jsp:param

- **Regular form:**

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
  <PARAM NAME="PARAM1" VALUE="VALUE1">
  <PARAM NAME="PARAM2" VALUE="VALUE2">
</APPLET>
```

- **JSP form for Java Plug-In:**

```
<jsp:plugin type="applet"
            code="MyApplet.class"
            width="475" height="350">
  <jsp:params>
    <jsp:param name="PARAM1" value="VALUE1" />
    <jsp:param name="PARAM2" value="VALUE2" />
  </jsp:params>
</jsp:plugin>
```

Alternative Text

- **Regular form:**

```
<APPLET CODE="MyApplet.class"
        WIDTH=475 HEIGHT=350>
  <B>Error: this example requires Java.</B>
</APPLET>
```

- **JSP form for Java Plug-In:**

```
<jsp:plugin type="applet"
```

```
            code="MyApplet.class"
            width="475" height="350">
  <jsp:fallback>
    <B>Error: this example requires Java.</B>
  </jsp:fallback>
</jsp:plugin>
```

- The Java Web Server does not properly handle `jsp:fallback`.

A.13 Using JavaBeans with JSP

Basic Requirements for Class to be a Bean

1. Have a zero-argument (empty) constructor.
2. Have no public instance variables (fields).
3. Access persistent values through methods called `get`*`Xxx`* (or `is`*`Xxx`*) and `set`*`Xxx`*.

Basic Bean Use

- `<jsp:useBean id="name" class="package.Class" />`
- `<jsp:getProperty name="name" property="property" />`
- `<jsp:setProperty name="name" property="property"`
  `value="value" />`

  The `value` attribute can take a JSP expression.

Associating Properties with Request Parameters

- **Individual properties:**

```
<jsp:setProperty
    name="entry"
    property="numItems"
    param="numItems" />
```

- **Automatic type conversion:** for primitive types, performed according to `valueOf` method of wrapper class.
- **All properties:**

```
<jsp:setProperty name="entry" property="*" />
```

Sharing Beans: The scope Attribute of jsp:useBean

Examples of sharing beans given in Chapter 15.

- **page**
  Default value. Indicates that, in addition to being bound to a local variable, bean object should be placed in `PageContext` object for duration of current request.

- **application**
  Means that, in addition to being bound to a local variable, bean will be stored in shared `ServletContext` available through predefined `application` variable or by a call to `getServletContext()`.
- **session**
  Means that, in addition to being bound to a local variable, bean will be stored in `HttpSession` object associated with current request, where it can be retrieved with `getValue`.
- **request**
  Signifies that, in addition to being bound to a local variable, bean object should be placed in `ServletRequest` object for duration of current request, where it is available by means of the `getAttribute` method.

Conditional Bean Creation

- A `jsp:useBean` element results in a new bean being instantiated only if no bean with the same `id` and `scope` can be found. If a bean with the same `id` and `scope` *is* found, the preexisting bean is simply bound to the variable referenced by `id`.
- You can make `jsp:setProperty` statements conditional on new bean creation:

```
<jsp:useBean ...>
  statements
</jsp:useBean>
```

A.14 Creating Custom JSP Tag Libraries

The Tag Handler Class

- Implement `Tag` interface by extending `TagSupport` (no tag body or tag body included verbatim) or `BodyTagSupport` (tag body is manipulated).
- `doStartTag`: code to run at beginning of tag
- `doEndTag`: code to run at end of tag
- `doAfterBody`: code to process tag body

The Tag Library Descriptor File

- Within `taglib` element, contains `tag` element for each tag handler. E.g.:

```
<tag>
  <name>prime</name>
  <tagclass>coreservlets.tags.PrimeTag</tagclass>
  <info>Outputs a random N-digit prime.</info>
  <bodycontent>EMPTY</bodycontent>
  <attribute>
    <name>length</name>
    <required>false</required>
  </attribute>
</tag>
```

The JSP File

- `<%@ taglib uri="some-taglib.tld" prefix="prefix" %>`
- `<prefix:tagname />`
- `<prefix:tagname>body</prefix:tagname>`

Assigning Attributes to Tags

- **Tag handler**:
  Implements `setXxx` for each attribute `xxx`.
- **Tag Library Descriptor:**

```
<tag>
  ...
  <attribute>
    <name>length</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue> <%-- sometimes --%>
  </attribute>
</tag>
```

Including the Tag Body

- **Tag handler**:
  You should return `EVAL_BODY_INCLUDE` instead of `SKIP_BODY` from `doStartTag`.
- **Tag Library Descriptor:**

```
<tag>
  ...
  <bodycontent>JSP</bodycontent>
</tag>
```

Optionally Including the Tag Body

- **Tag handler**:
  Return `EVAL_BODY_INCLUDE` or `SKIP_BODY` at different times, depending on value of request time parameter.

Manipulating the Tag Body

- **Tag handler**:
  You should extend `BodyTagSupport`, implement `doAfterBody`. Call `getBodyContent` to get `BodyContent` object describing tag body. `BodyContent` has three key methods: `getEnclosingWriter`, `getReader`, and `getString`. Return `SKIP_BODY` from `doAfterBody`.

Including or Manipulating the Tag Body Multiple Times

- **Tag handler**:
  To process body again, return `EVAL_BODY_TAG` from `doAfterBody`. To finish, return `SKIP_BODY`.

Using Nested Tags

- **Tag handler**:
  Nested tags can use `findAncestorWithClass` to find the tag in which they are nested. Place data in field of enclosing tag.
- **Tag Library Descriptor:**
  Declare all tags separately, regardless of nesting structure in real page.

A.15 Integrating Servlets and JSP

Big Picture

- Servlet handles initial request, reads parameters, cookies, session information, etc.
- Servlet then does whatever computations and database lookups are needed.
- Servlet then stores data in beans.
- Servlet forwards request to one of many possible JSP pages to present final result.
- JSP page extracts needed values from beans.

Request Forwarding Syntax

```
String url = "/path/presentation1.jsp";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
dispatcher.forward();
```

Forwarding to Regular HTML Pages

- If initial servlet handles `GET` requests only, no change is necessary.
- If initial servlet handles `POST`, then change destination page from SomePage.html to SomePage.jsp so that it, too, can handle `POST`.

Setting Up Globally Shared Beans

- **Initial servlet:**

```
Type1 value1 = computeValueFromRequest(request);
getServletContext().setAttribute("key1", value1);
```

- **Final JSP document:**

```
<jsp:useBean id="key1" class="Type1" scope="application" />
```

Setting Up Session Beans

- **Initial servlet:**

```
Type1 value1 = computeValueFromRequest(request);
HttpSession session = request.getSession(true);
session.putValue("key1", value1);
```

- **Final JSP document:**

```
<jsp:useBean id="key1" class="Type1" scope="session" />
```

Interpreting Relative URLs in the Destination Page

- URL of original servlet is used for forwarded requests. Browser does not know real URL, so it will resolve relative URLs with respect to original servlet's URL.

Getting a RequestDispatcher by Alternative Means (2.2 Only)

- **By name:** use `getNamedDispatcher` method of `ServletContext`.
- **By path relative to initial servlet's location:** use the `getRequestDispatcher` method of `HttpServletRequest` rather than the one from `ServletContext`.

Including Static or Dynamic Content

- **Basic usage:**

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("...");
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/path/resource");
dispatcher.include(request, response);
out.println("...");
```

- JSP equivalent is `jsp:include`, not the JSP `include` directive.

Forwarding Requests from JSP Pages

- `<jsp:forward page="Relative URL" />`

A.16 Using HTML Forms

The FORM Element

- **Usual form:**
  `<FORM ACTION="URL" ...> ... </FORM>`
- **Attributes:** `ACTION` (required), `METHOD`, `ENCTYPE`, `TARGET`, `ONSUBMIT`, `ONRESET`, `ACCEPT`, `ACCEPT-CHARSET`

Textfields

- **Usual form:**
  `<INPUT TYPE="TEXT" NAME="..." ...>` (no end tag)
- **Attributes:** `NAME` (required), `VALUE`, `SIZE`, `MAXLENGTH`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`
- Different browsers have different rules regarding the situations where pressing Enter in a textfield submits the form. So, include a button or image map that submits the form explicitly.

Password Fields

- **Usual form:**
  `<INPUT TYPE="PASSWORD" NAME="..." ...>` (no end tag)
- **Attributes:** `NAME` (required), `VALUE`, `SIZE`, `MAXLENGTH`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`
- Always use `POST` with forms that contain password fields.

Text Areas

- **Usual form:**

```
<TEXTAREA NAME="..." ROWS=xxx COLS=yyy> ...
  Some text
</TEXTAREA>
```

- **Attributes:** NAME (required), ROWS (required), COLS (required), WRAP (nonstandard), ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP
- White space in initial text is maintained and HTML markup between start and end tags is taken literally, except for character entities such as <, ©, and so forth.

Submit Buttons

- **Usual form:**
  <INPUT TYPE="SUBMIT" ...> (no end tag)
- **Attributes:** NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR
- When a submit button is clicked, the form is sent to the servlet or other server-side program designated by the ACTION parameter of the FORM.

Alternative Push Buttons

- **Usual form:**

```
<BUTTON TYPE="SUBMIT" ...>
  HTML Markup
</BUTTON>
```

- **Attributes:** NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR
- Internet Explorer only.

Reset Buttons

- **Usual form:**
  <INPUT TYPE="RESET" ...> (no end tag)
- **Attributes:** VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR
  Except for VALUE, attributes are only for use with JavaScript.

Alternative Reset Buttons

- **Usual form:**

```
<BUTTON TYPE="RESET" ...>
  HTML Markup
</BUTTON>
```

- **Attributes:** VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR
- Internet Explorer only.

JavaScript Buttons

- **Usual form:**
  `<INPUT TYPE="BUTTON" ...>` (no end tag)
- **Attributes:** `NAME`, `VALUE`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`

Alternative JavaScript Buttons

- **Usual form:**

```
<BUTTON TYPE="BUTTON" ...>
  HTML Markup
</BUTTON>
```

- **Attributes:** `NAME`, `VALUE`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`
- Internet Explorer only.

Check Boxes

- **Usual form:**
  `<INPUT TYPE="CHECKBOX" NAME="..." ...>` (no end tag)
- **Attributes:** `NAME` (required), `VALUE`, `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`
- Name/value transmitted only if check box is checked.

Radio Buttons

- **Usual form:**
  `<INPUT TYPE="RADIO" NAME="..." VALUE="..." ...>`
  (no end tag)
- **Attributes:** `NAME` (required), `VALUE` (required), `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`
- You indicate a group of radio buttons by providing all of them with the same `NAME`.

Combo Boxes

- **Usual form:**

```
<SELECT NAME="Name" ...>
  <OPTION VALUE="Value1">Choice 1 Text
  <OPTION VALUE="Value2">Choice 2 Text
  ...
  <OPTION VALUE="ValueN">Choice N Text
</SELECT>
```

- **`SELECT` Attributes:** `NAME` (required), `SIZE`, `MULTIPLE`, `ONCLICK`, `ONFOCUS`, `ONBLUR`, `ONCHANGE`
- **`OPTION` Attributes:** `SELECTED`, `VALUE`

File Upload Controls

- **Usual form:**
  `<INPUT TYPE="FILE" ...>` (no end tag)
- **Attributes:** `NAME` (required), `VALUE` (ignored), `SIZE`, `MAXLENGTH`, `ACCEPT`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR` (nonstandard)
- Use an `ENCTYPE` of `multipart/form-data` in the `FORM` declaration.

Server-Side Image Maps

- **Usual form:**
  `<INPUT TYPE="IMAGE" ...>` (no end tag)
- **Attributes:** `NAME` (required), `SRC`, `ALIGN`
- You can also provide an `ISMAP` attribute to a standard `IMG` element that is inside an `<A HREF...>` element.

Hidden Fields

- **Usual form:**
  `<INPUT TYPE="HIDDEN" NAME="..." VALUE="...">` (no end tag)
- **Attributes:** `NAME` (required), `VALUE`

Internet Explorer Features

- **FIELDSET (with LEGEND):** groups controls
- **TABINDEX**: controls tabbing order
- Both capabilities are part of HTML 4.0 spec; neither is supported by Netscape 4.

A.17 Using Applets As Servlet Front Ends

Sending Data with GET and Displaying the Resultant Page

```
String someData =
  name1 + "=" + URLEncoder.encode(val1) + "&" +
  name2 + "=" + URLEncoder.encode(val2) + "&" +
  ...
  nameN + "=" + URLEncoder.encode(valN);
try {
  URL programURL = new URL(baseURL + "?" + someData);
  getAppletContext().showDocument(programURL);
} catch(MalformedURLException mue) { ... }
```

Sending Data with GET and Processing the Results Directly (HTTP Tunneling)

1. **Create a `URL` object referring to applet's home host.** You usually build a URL based upon the hostname from which the applet was loaded.
```
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();
String host = currentPage.getHost();
int port = currentPage.getPort();
String urlSuffix = "/servlet/SomeServlet";
URL dataURL = new URL(protocol, host, port, urlSuffix);
```
2. **Create a `URLConnection` object.** The `openConnection` method of `URL` returns a `URLConnection` object. This object will be used to obtain streams with which to communicate.
```
URLConnection connection = dataURL.openConnection();
```
3. **Instruct the browser not to cache the URL data.**
```
connection.setUseCaches(false);
```
4. **Set any desired HTTP headers.** If you want to set HTTP request headers (see Chapter 4), you can use `setRequestProperty` to do so.
```
connection.setRequestProperty("header", "value");
```
5. **Create an input stream.** There are several appropriate streams, but a common one is `BufferedReader`. It is at the point where you create the input stream that the connection to the Web server is actually established behind the scenes.
```
BufferedReader in =
  new BufferedReader(new InputStreamReader(
                         connection.getInputStream()));
```
6. **Read each line of the document.** Simply read until you get `null`.
```
String line;
while ((line = in.readLine()) != null) {
  doSomethingWith(line);
}
```
7. **Close the input stream.**
```
in.close();
```

Sending Serialized Data: The Applet Code

1. **Create a `URL` object referring to the applet's home host.** It is best to specify a URL suffix and construct the rest of the URL automatically.
```
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();
String host = currentPage.getHost();
int port = currentPage.getPort();
String urlSuffix = "/servlet/SomeServlet";
URL dataURL = new URL(protocol, host, port, urlSuffix);
```
2. **Create a `URLConnection` object.** The `openConnection` method of `URL` returns a `URLConnection` object. This object will be used to obtain streams with which to communicate.
```
URLConnection connection = dataURL.openConnection();
```
3. **Instruct the browser not to cache the URL data.**
```
connection.setUseCaches(false);
```
4. **Set any desired HTTP headers.**
```
connection.setRequestProperty("header", "value");
```
5. **Create an `ObjectInputStream`.** The constructor for this class simply takes the raw input stream from the `URLConnection`.
```
ObjectInputStream in =
  new ObjectInputStream(connection.getInputStream());
```
6. **Read the data structure with `readObject`.** The return type of `readObject` is `Object`, so you need to make a typecast to whatever more specific type the server actually sent.
```
SomeClass value = (SomeClass)in.readObject();
doSomethingWith(value);
```
7. **Close the input stream.**
```
in.close();
```

Sending Serialized Data: The Servlet Code

1. **Specify that binary content is being sent.** To do so, designate
```
application/x-java-serialized-object
```
as the MIME type of the response. This is the standard MIME type for objects encoded with an `ObjectOutputStream`, although in practice, since the applet (not the browser) is reading the result, the MIME type is not very important. See the discussion of `Content-Type` in Section 7.2 for more information on MIME types.
```
String contentType =
  "application/x-java-serialized-object";
response.setContentType(contentType);
```

2. **Create an ObjectOutputStream.**
```
ObjectOutputStream out =
  new ObjectOutputStream(response.getOutputStream());
```
3. **Write the data structure by using `writeObject`.** Most built-in data structures can be sent this way. Classes *you* write, however, must implement the `Serializable` interface.
```
SomeClass value = new SomeClass(...);
out.writeObject(value);
```
4. **Flush the stream to be sure all content has been sent to the client.**
```
out.flush();
```

Sending Data by POST and Processing the Results Directly (HTTP Tunneling)

1. **Create a `URL` object referring to the applet's home host.** It is best to specify a URL suffix and construct the rest of the URL automatically.
```
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();
String host = currentPage.getHost();
int port = currentPage.getPort();
String urlSuffix = "/servlet/SomeServlet";
URL dataURL =
  new URL(protocol, host, port, urlSuffix);
```
2. **Create a `URLConnection` object.**
```
URLConnection connection = dataURL.openConnection();
```
3. **Instruct the browser not to cache the results.**
```
connection.setUseCaches(false);
```
4. **Tell the system to permit you to send data, not just read it.**
```
connection.setDoOutput(true);
```
5. **Create a `ByteArrayOutputStream` to buffer the data that will be sent to the server.** The purpose of the `ByteArrayOutputStream` here is the same as it is with the persistent (keep-alive) HTTP connections shown in Section 7.4 — to determine the size of the output so that the `Content-Length` header can be set.
```
ByteArrayOutputStream byteStream =
  new ByteArrayOutputStream(512);
```
6. **Attach an output stream to the `ByteArrayOutputStream`.** Use a `PrintWriter` to send normal form data. To send serialized data structures, use an `ObjectOutputStream` instead.
```
PrintWriter out = new PrintWriter(byteStream, true);
```

7. **Put the data into the buffer.** For form data, use `print`. For high-level serialized objects, use `writeObject`.
```
String val1 = URLEncoder.encode(someVal1);
String val2 = URLEncoder.encode(someVal2);
String data = "param1=" + val1 +
              "&param2=" + val2; // Note '&'
out.print(data);  // Note print, not println
out.flush(); // Necessary since no println used
```
8. **Set the `Content-Length` header.** This header is required for `POST` data, even though it is unused with `GET` requests.
```
connection.setRequestProperty
  ("Content-Length", String.valueOf(byteStream.size()));
```
9. **Set the `Content-Type` header.** Netscape uses `multipart/form-data` by default, but regular form data requires a setting of `application/x-www-form-urlencoded`, which is the default with Internet Explorer. So, for portability you should set this value explicitly when sending regular form data. The value is irrelevant when you are sending serialized data.
```
connection.setRequestProperty
  ("Content-Type", "application/x-www-form-urlencoded");
```
10. **Send the real data.**
```
byteStream.writeTo(connection.getOutputStream());
```
11. **Open an input stream.** You typically use a `BufferedReader` for ASCII or binary data and an `ObjectInputStream` for serialized Java objects.
```
BufferedReader in =
  new BufferedReader(new InputStreamReader
                       (connection.getInputStream()));
```
12. **Read the result.**
    The specific details will depend on what type of data the server sends. Here is an example that does something with each line sent by the server:
```
String line;
while((line = in.readLine()) != null) {
  doSomethingWith(line);
}
```

Bypassing the HTTP Server

Applets can talk directly to servers on their home host, using any of:

- Raw sockets
- Sockets with object streams
- JDBC
- RMI
- Other network protocols

A.18 JDBC and Database Connection Pooling

Basic Steps in Using JDBC

1. **Load the JDBC driver.** See http://java.sun.com/products/jdbc/drivers.html for available drivers. Example:
```
Class.forName("package.DriverClass");
Class.forName("oracle.jdbc.driver.OracleDriver");
```
2. **Define the connection URL.** The exact format will be defined in the documentation that comes with the particular driver.
```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
                   ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host  +
                   ":" + port + ":" + "?SERVICENAME=" +
                   dbName;
```
3. **Establish the connection.**
```
String username = "jay_debesee";
String password = "secret";
Connection connection =
  DriverManager.getConnection(oracleURL, username, password)
```
An optional part of this step is to look up information about the database by using the `getMetaData` method of `Connection`. This method returns a `DatabaseMetaData` object which has methods to let you discover the name and version of the database itself (`getDatabaseProductName`, `getDatabaseProductVersion`) or of the JDBC driver (`getDriverName`, `getDriverVersion`).
4. **Create a statement object.**
```
Statement statement = connection.createStatement();
```
5. **Execute a query or update.**
```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```
6. **Process the results.** Use `next` to get a new row. Use `getXxx(index)` or `getXxx(columnName)` to extract values from a row. First column has index 1, not 0.
```
while(resultSet.next()) {
  System.out.println(results.getString(1) + " " +
                     results.getString(2) + " " +
                     results.getString(3));
}
```
7. **Close the connection.**
```
connection.close();
```

Database Utilities

These are static methods in the `DatabaseUtilities` class (Listing 18.6).

- **getQueryResults**
  Connects to a database, executes a query, retrieves all the rows as arrays of strings, and puts them inside a `DBResults` object (see Listing 18.7). Also places the database product name, database version, the names of all the columns and the `Connection` object into the `DBResults` object. There are two versions of `getQueryResults`: one makes a new connection, the other uses an existing connection. `DBResults` has a simple `toHTMLTable` method that outputs result in HTML, which can be used as either a real HTML table or as an Excel spreadsheet (see Section 11.2).
- **createTable**
  Given a table name, a string denoting the column formats, and an array of strings denoting the row values, this method connects to a database, removes any existing versions of the designated table, issues a `CREATE TABLE` command with the designated format, then sends a series of `INSERT INTO` commands for each of the rows. Again, there are two versions: one makes a new connection, and the other uses an existing connection.
- **printTable**
  Given a table name, this method connects to the specified database, retrieves all the rows, and prints them on the standard output. It retrieves the results by turning the table name into a query of the form "`SELECT * FROM tableName`" and passing it to `getQueryResults`.
- **printTableData**
  Given a `DBResults` object from a previous query, this method prints it on the standard output. This is the underlying method used by `printTable`, but it is also useful for debugging arbitrary database results.

Prepared Statements (Precompiled Queries)

- **Use `connection.prepareStatement` to make precompiled form. Mark parameters with question marks.**

```
String template =
  "UPDATE employees SET salary = ? WHERE id = ?";
PreparedStatement statement =
  connection.prepareStatement(template);
```

- **Use `statement.setXxx` to specify parameters to query.**

```
statement.setFloat(1, 1.234);
statement.setInt(2, 5);
```

- **Use `execute` to perform operation.**

```
statement.execute();
```

Steps in Implementing Connection Pooling

If you don't care about implementation details, just use the `ConnectionPool` class developed in Chapter 18. Otherwise, follow these steps.

1. **Preallocate the connections.**
   Perform this task in the class constructor. Call the constructor from servlet's `init` method. The following code uses vectors to store available idle connections and unavailable, busy connections.

```
availableConnections = new Vector(initialConnections);
busyConnections = new Vector();
for(int i=0; i<initialConnections; i++) {
  availableConnections.addElement(makeNewConnection());
}
```

2. **Manage available connections.**
   If a connection is required and an idle connection is available, put it in the list of busy connections and then return it. The busy list is used to check limits on the total number of connections as well as when the pool is instructed to explicitly close all connections. Discarding a connection opens up a slot that can be used by processes that needed a connection when the connection limit had been reached, so use `notifyAll` to tell all waiting threads to wake up and see if they can proceed.

```
public synchronized Connection getConnection()
    throws SQLException {
  if (!availableConnections.isEmpty()) {
    Connection existingConnection =
      (Connection)availableConnections.lastElement();
    int lastIndex = availableConnections.size() - 1;
    availableConnections.removeElementAt(lastIndex);
    if (existingConnection.isClosed()) {
      notifyAll(); // Freed up a spot for anybody waiting.
      return(getConnection()); // Repeat process.
    } else {
      busyConnections.addElement(existingConnection);
      return(existingConnection);
    }
  }
}
```

3. **Allocate new connections.**
   If a connection is required, there is no idle connection available, and the connection limit has not been reached, then start a

background thread to allocate a new connection. Then, wait for the first available connection, whether or not it is the newly allocated one.

```
if ((totalConnections() < maxConnections) &&
    !connectionPending) { // Pending = connecting in bg
  makeBackgroundConnection();
  try {
    wait(); // Give up lock and suspend self.
  } catch(InterruptedException ie) {}
  return(getConnection()); // Try again.
```

4. **Wait for a connection to become available.**
   This situation occurs when there is no idle connection and you've reached the limit on the number of connections. This waiting should be accomplished without continual polling. It is best to use the `wait` method, which gives up the thread synchronization lock and suspends the thread until `notify` or `notifyAll` is called.

```
try {
  wait();
} catch(InterruptedException ie) {}
return(getConnection());
```

5. **Close connections when required.**
   Note that connections are closed when they are garbage collected, so you don't always have to close them explicitly. But, you sometimes want more explicit control over the process.

```
public synchronized void closeAllConnections() {
  // The closeConnections method loops down Vector, calling
  // close and ignoring any exceptions thrown.
  closeConnections(availableConnections);
  availableConnections = new Vector();
  closeConnections(busyConnections);
  busyConnections = new Vector();
}
```

Connection pool timing results	
Condition	***Average Time***
Slow modem connection to database, 10 initial connections, 50 max connections (`ConnectionPoolServlet`)	11 seconds
Slow modem connection to database, recycling a single connection (`ConnectionPoolServlet2`)	22 seconds

Connection pool timing results	
Condition	*Average Time*
Slow modem connection to database, no connection pooling (`ConnectionPoolServlet3`)	82 seconds
Fast LAN connection to database, 10 initial connections, 50 max connections (`ConnectionPoolServlet`)	1.8 seconds
Fast LAN connection to database, recycling a single connection (`ConnectionPoolServlet2`)	2.0 seconds
Fast LAN connection to database, no connection pooling (`ConnectionPoolServlet3`)	2.8 seconds