RV Educational Institutions®

RV Institute of Technology and Management

(Affiliated to VTU, Belagavi)

JP Nagar 8th Phase, Bengaluru - 560076

Department of Information Science and Engineering

Course Name: Advanced Java

Course Code: BIS402

Module 4 – Servlets & JSP

IV Semester

2022 Scheme

Prepared By :

Dr. Vinoth Kumar M & Dr. Kirankumar K,

Associate & Assistant Professor,

Department of Information

Science and Engineering RVITM, Bengaluru -560076

Introducing Servlets

Servlets are small programs that execute on the server side of a web connection.

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content toa user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a databaseto store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These includedC, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client

request. It was also expensive to open and close database connections foreach client request. In addition, the CGI programs were not platform- independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space ofa web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set ofrestrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with other software via the sockets and RMI mechanisms that you have seen already.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init( )**, **service()**, and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenarioto understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a webbrowser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server mapsthis request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init( )** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to passinitialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service( )** method of the servlet. This methodis called to process the HTTP request. You will see that it is possible for the servlet to

read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service( )** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the **destroy( )** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To experiment with servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Apache Tomcat. Glassfish is an open source project sponsored by Oracle and is provided with the Java EE SDK. It is supported by NetBeans. Apache Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support thatyou will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 8.5.31. Thisversion of Tomcat is used here because it will work for a very wide range of readers. You can download Tomcat from **tomcat.apache.org**. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 8.5.31 was unpacked from the rootdirectly, the default location is

C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\

This is the location assumed by the examples in this notes. If you load Tomcatin a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-8.5.31** directory. To stop Tomcat, execute**shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib

To make **servlet-api.jar** accessible, update your **CLASSPATH** environmentvariable so that it includes

C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib\servlet-api.jar

Alternatively, you can specify this file when you compile the servlets. Forexample, the following command compiles the first servlet example:

javac HelloServlet.java -classpath "C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\lib\servlet-api.jar"

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, theexamples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\webapps\examples\WEB-INF\classes

Next, add the servlet's name and mapping to the **web.xml** file in the followingdirectory:

C:\apache-tomcat-8.5.31-windows-x64\apache-tomcat-8.5.31\webapps\examples\WEB-INF

For instance, assuming the first example, called **HelloServlet**, you will add thefollowing lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servletmappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlets/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's classfile to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.
2. Start Tomcat.
3. Start a web browser and request the servlet.Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the followingprogram:

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

public class HelloServlet extends GenericServlet {

  public void service(ServletRequest request,
    ServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>Hello!");
    pw.close();
  }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init( )** and **destroy( )**, which may be used as is. You need supply only the **service( )** method.

Inside **HelloServlet**, the **service( )** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType( )** establishes the MIME type of the HTTP response. In this program, the MIME type is text/html. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter( )** method obtains a **PrintWriter**. Anything written to this stream

is sent to the client as part of the HTTP response. Then **println( )** isused to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try toexecute a servlet.

Start a Web Browser and Request the Servlet

Start a web browser and enter the URL shown here:

http://localhost:8080/examples/servlets/servlet/HelloServlet

Alternatively, you may enter the URL shown here:

http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet

This can be done because 127.0.0.1 is defined as the IP address of the localmachine.

You will observe the output of the servlet in the browser display area. It willcontain the string **Hello!** in bold type.

The Servlet API

Two packages contain the classes and interfaces that are required to build the servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they

are provided by Tomcat. They are alsoprovided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The servlet specification supported by Tomcat 8.5.31 is version 3.1 (As a point of interest, Tomcat 9 supports servlet specification 4.).

The javax.servlet Package

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes several key interfaces that are provided in this package. The mostsignificant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the **javax.servlet** package:

Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the **Servlet** interface. It declares the **init( )**, **service( )**, and **destroy( )** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by **Servlet** are shown in Table 4-1.

Table 4-1 The Methods Defined by **Servlet**

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. A **ServletException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

The **init( )**, **service( )**, and **destroy( )** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig( )** method iscalled by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo( )** method to provide a string with useful information (for example, the version number). This method is also invoked bythe server.

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data whenit is loaded. The methods declared by this interface are summarized here:

Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration<String> getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface enables servlets to obtain information about their environment. Several of its methods are summarized here Table 4-2.

Table 4-2 Various Methods Defined by **ServletContext**

Method	Description
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real (i.e., absolute) path that corresponds to the relative path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Writes *s* and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

The ServletRequest Interface

The **ServletRequest** interface enables a servlet to obtain information about aclient request. Several of its methods are summarized in Table 4-3.

Table 4-3 Various Methods Defined by **ServletRequest**

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has been previously invoked on this object.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration<String> getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has been previously invoked on this object.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

The ServletResponse Interface

The **ServletResponse** interface enables a servlet to formulate a response for aclient. Several of its methods are summarized in Table 4-4.

Table 4-4 Various Methods Defined by **ServletResponse**

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has been previously invoked on this object.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has been previously invoked on this object.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

The GenericServlet Class

The **GenericServlet** class provides implementations of the basic life cycle methods for a servlet. **GenericServlet** implements the **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string to the serverlog file is available. The signatures of this method are shown here:

void log(String *s*)
void log(String *s*, Throwable *e*)

Here, *s* is the string to be appended to the log, and *e* is an exception thatoccurred.

The ServletInputStream Class

The **ServletInputStream** class extends **InputStream**. It is implemented by theservlet container and provides an input stream that a servlet developer can use to read the data from a client request. In addition to the input methods inherited from **InputStream**, a method is provided to read bytes from the stream. It is shown here:

int readLine(byte[ ] *buffer*, int *offset*, int *size*) throws IOException

Here, *buffer* is the array into which *size* bytes are placed starting at *offset.* Themethod returns the actual number of bytes read or –1 if an end-of-stream condition is encountered.

The ServletOutputStream Class

The **ServletOutputStream** class extends **OutputStream**. It is implemented by the servlet container and provides an output stream that a servlet developer canuse to write data to a client response. In addition to the output methods provided by **OutputStream**, it also defines the **print( )** and **println( )** methods,which output data to the stream.

The Servlet Exception Classes

javax.servlet defines two exceptions. The first is **ServletException**, whichindicates that a servlet problem has occurred. The second is **UnavailableException**, which extends **ServletException**. It indicates that aservlet is unavailable.

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We willdevelop a servlet that illustrates their use. The example contains two files. Aweb page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of thelabels is Employee and the other is Phone. There is also a submit button. Noticethat the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/
          servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service( )** method is overridden to process client requests. The **getParameterNames( )** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name andvalue are output to the client. The parameter value is obtained via the **getParameter( )** method.

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

public class PostParametersServlet
extends GenericServlet {

  public void service(ServletRequest request,
    ServletResponse response)
  throws ServletException, IOException {

    // Get print writer.
    PrintWriter pw = response.getWriter();

    // Get enumeration of parameter names.
    Enumeration<String> e = request.getParameterNames();

    // Display parameter names and values.
    while(e.hasMoreElements()) {
      String pname = e.nextElement();
      pw.print(pname + " = ");
      String pvalue = request.getParameter(pname);
      pw.println(pvalue);
    }
    pw.close();
  }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test thisexample:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.

3. Enter an employee name and phone number in the text fields.
4. Submit the web page.

After following these steps, the browser will display a response that isdynamically generated by the servlet.

The javax.servlet.http Package

The preceding examples have used the classes and interfaces defined in **javax.servlet**, such as **ServletRequest**, **ServletResponse**, and **GenericServlet**, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in **javax.servlet.http**.
As you will see, its functionality makes it easy to build servlets that work withHTTP requests and responses.

The following table summarizes the interfaces used in this chapter:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.

The following table summarizes the classes used in this chapter. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.

The HttpServletRequest Interface

The **HttpServletRequest** interface enables a servlet to obtain informationabout a client request. Several of its methods are shown in Table 4-5.

Table 4-5 Various Methods Defined by **HttpServletRequest**

Method	Description
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration<String> getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

The HttpServletResponse Interface

The **HttpServletResponse** interface enables a servlet to formulate an HTTPresponse

to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example,**SC_OK** indicates that the HTTP request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in Table 4-6.

Table 4-6 Various Methods Defined by **HttpServletResponse**

Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods aresummarized in Table 4-7. All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Table 4-7 Various Methods Defined by **HttpSession**

Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration<String> getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save theuser's name, address, and other information. The user does not need to enterthis data each time he or she

visits the store.

A servlet can write a cookie to a user's machine via the **addCookie( )** method of the **HttpServletResponse** interface. The data for that cookie is thenincluded in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some ofthe information that can be saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it isdeleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, itis not.

There is one constructor for **Cookie**. It has the signature shown here:Cookie(String *name*, String *value*)

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in Table 4-8.

Table 4-8 The Methods Defined by **Cookie**

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
boolean isHttpOnly( )	Returns **true** if the cookie has the **HttpOnly** attribute.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setHttpOnly(boolean *httpOnly*)	If *httpOnly* is **true**, then the **HttpOnly** attribute is added to the cookie. If *httpOnly* is **false**, the **HttpOnly** attribute is removed.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

The HttpServlet Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used whendeveloping servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in Table 4-9.

Table 4-9 The Methods Defined by **HttpServlet**

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP OPTIONS request.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**,**doPost( )**, **doPut( )**, and **doTrace( )**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains twofiles. A web page is defined in **ColorGet.html**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.html** is shown inthe following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
  action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing.The **doGet( )** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

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

public class ColorGetServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");
    pw.println(color);
    pw.close();
  }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test thisexample:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as partof the URL that is sent to the web server. Assume that the user selects the redoption and submits the form. The URL sent from the browser to the server is

http://localhost:8080/examples/servlets/servlet/ColorGetServlet? color=Red

The characters to the right of the question mark are known as the *query string*.

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servletis invoked when a form on a web page is submitted. The example contains twofiles. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and theaction parameter for the form tag specifies a different servlet.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost( )** method is overridden to process any HTTP POST requests that are sent to

this servlet. It uses the **getParameter( )** method of**HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

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

public class ColorPostServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is: ");
    pw.println(color);
    pw.close();
  }
}
```

Compile the servlet and perform the same steps as described in the previous

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains threefiles as summarized here:

File	Description
AddCookie.html	Allows a user to specify a value for the cookie named **MyCookie**.
AddCookieServlet.java	Processes the submission of **AddCookie.html**.
GetCookiesServlet.java	Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following listing. This page contains a text field in which a value can be entered. There isalso a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie**object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the**addCookie( )** method. A feedback message is then written to the browser.

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

public class AddCookieServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {
    // Get parameter from HTTP request.
    String data = request.getParameter("data");

    // Create cookie.
    Cookie cookie = new Cookie("MyCookie", data);

    // Add cookie to HTTP response.
    response.addCookie(cookie);

    // Write output to browser.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>MyCookie has been set to");
    pw.println(data);
    pw.close();
  }
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies( )** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

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

public class GetCookiesServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
  throws ServletException, IOException {

    // Get cookies from header of HTTP request.
    Cookie[] cookies = request.getCookies();

    // Display these cookies.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>");
    for(int i = 0; i < cookies.length; i++) {
      String name = cookies[i].getName();
      String value = cookies[i].getValue();
      pw.println("name = " + name +
        "; value = " + value);
    }
    pw.close();
  }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps totest this example:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.html** in a browser.
3. Enter a value for **MyCookie**.
4. Submit the web page.

After completing these steps, you will observe that a feedback message isdisplayed by the browser.

Next, request the following URL via the browser:

http://localhost:8080/examples/servlets/servlet/GetCookiesServlet

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookievia the **setMaxAge( )** method of **Cookie**. Therefore, the cookie expires whenthe browser session ends. You can experiment by using **setMaxAge( )** and observe that the cookie is then saved on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and aserver. Sessions provide such a mechanism.

A session can be created via the **getSession( )** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can storea set of bindings that associate names with objects. The **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )**, and

removeAttribute( ) methods of **HttpSession** manage these bindings. Session state is shared by all servlets thatare associated with a client.

The following servlet illustrates how to use session state. The **getSession( )** method gets the current session. A new session is created if one does not already exist. The **getAttribute( )** method is called to obtain the object that is bound to the name "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute( )** method is called to bind the name "date" to this object.

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

public class DateServlet extends HttpServlet {
```

```
public void doGet(HttpServletRequest request,
   HttpServletResponse response)
throws ServletException, IOException {

   // Get the HttpSession object.
   HttpSession hs = request.getSession(true);

   // Get writer.
   response.setContentType("text/html");
   PrintWriter pw = response.getWriter();
   pw.print("<B>");

   // Display date/time of last access.
   Date date = (Date)hs.getAttribute("date");
   if(date != null) {
     pw.print("Last access: " + date + "<br>");
   }

   // Display current date/time.
   date = new Date();
   hs.setAttribute("date", date);
   pw.println("Current date: " + date);
  }
}
```

When you first request this servlet, the browser displays one line with thecurrent date and time information. On subsequent invocations, two lines aredisplayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

JSP

Java Server Page (JSP) is a server-side program that is similar in design and functionality to a Java servlet. A JSP is called by a client to provide a web service, the nature of which depends on the J2EE application. A JSP processes the request by using logic built into the JSP or by calling other web components built using Java servlet technology or Enterprise JavaBean technology, or created using other technologies. Once the request is processed, the JSP responds by sending the results to the client.

There are three methods that are automatically called when a JSP is requested and when the JSP terminates normally.

These are the jsplnt() method, the jspDestroy() method, and the service() method.

These methods can be overridden, although the jsplnt() method and jspDestroy() methods are commonly overridden in a JSP to provide customized functionality when the JSP is called and terminates. The jsplnt() method is identical the init() method in a Java servlet and in an applet.

The jsplnt() method is called first when the JSP is requested and is used to initialize objects and variables that are used throughout the life of the JSP. The jspDestroy() method is identical to the destroy() method in a Java servlet. The destroy() method is automatically called when the JSP terminates normally. It isn't called if the JSP abruptly terminates such as when the server crashes. The destroy() method is used for cleanup where resources. used during the execution of the JSP are released, such as disconnecting from a database.

The service() method is automatically called and retrieves a connection to HTTP.

JSP Tags:

A JSP program consists of a combination of HTML tags and JSP tags. JSP tags define Java code that is to be executed before the output of the JSP program is sent to the browser. AJSP tag begins with a <% which is followed by java code and ends with %>. There is also an Extendable Markup Language (XML) version of JSP tags, which are formatted as <jsp:TagID> </jsp:TagID>.

JSP tags are embedded into the HTML component of a JSP program and are processed by a JSP virtual engine such as Tomcat, which is discussed later in this

chapter. Tomcat reads the JSP program whenever the program is called by a browser and resolves JSP tags, then sends the HTML tags and related information to the browser.

There are five types of JSP tags:

Comment tag: A comment tag opens with<%-- and closes with--%>, and is followed by a comment that usually describes the functionality of statements that follow the comment tag.

Declaration statement tags: A declaration statement tag opens with <%! and is followed by a Java declaration statement(s) that define variables, objects, and methods that are available to other components of the JSP program .

Directive tags: A directive tag opens with<%@ and commands the JSP virtual engine to perform a specific task, such as importing a Java package required by objects and methods used in a declaration statement. The directive tag closes with %>. There are three commonly used directives. TI1ese are import, include, and taglib. The import tag is used to import Java packages into the JSP program.

The include tag inserts a specified file into the JSP program replacing the include tag. The taglib tag specifies a file that contains a tag library. Here are examples of each tag. The first tag imports the java.sql package. The next tag includes the books.html file located in the keogh directory. And the last tag loads the myTags.tld library.

<%@ page import=" import java . sql.* "; %>

<%@ include file="keogh\books.html" %>

<%@ taglib uri= "rnyTags.tld" %>

Expression tags: An expression tag opens with <%= and is used for an expression statement whose result replaces the expression tag when the JSP virtual engine resolves JSP tags. An expression tags closes with%>.

Scriptlet tags: A scriptlet tag opens with <% and contains commonly used Java control statements and loops. A scriptlet tag closes with%>.

Variables and Objects:

You can declare Java variables and objects that are used in a JSP program by using the same coding technique as used to declare them in Java. However, the declaration statement must appear as a JSP tag within the JSP program before the variable or object is used in the program.

Listing program shows a simple JSP program that declares and uses a variable. In this example, the program declares an int called age and initializes the variable with the number 29. The declaration statement is placed within JSP tag.

You'll notice that this JSP tag begins with<%!. This tells the JSP virtual engine to make statements contained in the tag available to other JSP tags in the program. You'll need to do this nearly every time you declare variables or objects in your program unless they are only to be used within the JSP tag where they are declared.

The variable age is used in an expression tag that is embedded within the HTML paragraph tag <P>. A JSP expression tag begins with <%=, which is followed by the expression.

The JSP virtual engine resolves the JSP expression before sending the output of the JSP program to the browser. That is, the JSP tag <%=age%> is replaced with the number 29; afterwards, the HTML paragraph tag and related information is sent to the browser.

```
<HTML>
<HEAD>
<TITLE> JSP Programming </TITLE>
</HEAD>
<BODY>
<%! int age=29; %>
<P> Your age is: <%=age%> </P>
</BODY>
< / HTML>
```

Any Java declaration statement can be used in a JSP tag similar to how those statements are used in a Java program. You are able to place multiple statements within a JSP tag by extending the close JSP tag to another line in the JSP program.

```
<HTML>
<HEAD>
<TI'I'LE> JSP Programming < / TI'rLE>
</HEAD>
<BODY>
<% ! int age=29;
float salary;
int empnurnber;
%>
</ BODY>
</ HTML>
```

Methods:

JSP offers the same versatility that you have with Java programs, such as defining methods that are local to the JSP program. A method is defined similar to how a method is defined in a Java program except the method definition is placed within a JSP tag. You can call the method from within the JSP tag once the method is defined. In the programming example, the method is passed a student's grade and then applies a curve before returning the curved grade. The method is called from within an HTML paragraph tag in this program, although any appropriate tag can be used to call the method. Technically, the method is called from within the JSP tag that is enclosed within the HTML paragraph tag.

```
<HTML>
<HEAD>
<TITLE> JSP Programming < / 'I'ITLE>
< / HEAD>
<BODY>
<%! boolean curve (int grade)
return 10 + grade;
```

```
%>
<P> Your curved grade is : <%=curve( 8 0)%> < / P>
</BODY>
< /HTML>
```

A JSP program is capable of handling practically any kind of method that you normally use in a Java program. For example, Listing program shows how to define and use an overloaded method. Both methods are defined in the same JSP tag, although each follows Java syntax structure for defining a method. One method uses a default value for the curve, while the overload method enables the statement that calls the method to provide the value of the curve. Once again, these methods are called from an embedded JSP tag placed inside two HTML paragraph tags.

```
<HTML>
<HEAD>
<TITLE> JSP Programming < / TI TLE>
</HEAD>
<BODY>
<%! boolean curve (int grade )
{
return 10 + grade;
}
boolean curve ( int grade , int curveValue )
{
return curveValue + grade;
%>
<P> Your curved grade is: <% =curve( 80 , 10)%> < / P>
<P> Your curved grade is: <%=curve(70)%> < / P>
< / BODY>
</ HTML>
```

Control Statements:

One of the most powerful features available in JSP is the ability to change the flow of the program to truly create dynamic content for a web page based on conditions

received from the browser.

There are two control statements used to change the flow of a JSP program. These are the if statement and the switch statement, both of which are also used to direct the flow of a Java program.

The if statement evaluates a condition statement to determine if one or more lines of code are to be executed or skipped (as you probably remember from when you learned Java). Similarly, a switch statement compares a value with one or more other values associated with a case statement. The code segment that is associated with the matching case statement is executed. Code segments associated with other case statements are ignored.

```
<HTML>
<HEAD>
<TITLE> JSP Programming </TITLE>
</HEAD>
<BODY>
<% ! int grade=70;%>
<% if (grade > 69) { %>
<P> You passed! </ P>
<%} else { %>
<P> Better luck n ext time. </P>
<%} %>
<% switch (grade) {
case 90 : %>
<P> Your final grade is a A </P>
<% break; %>
case 80 : %>
<P> Your final grade is a B </P>
<% break;
case 70 : %>
<P> Your final grade is a C </P>
<% break;
case 60 : %>
```

```
<P> Your final grade is an F </P>
<% break ;
}
%>
</ BODY>
</HTML>
```

Loops

JSP loops are nearly identical to loops that you use in your Java program except you can repeat HTML tags and related information multiple times within your JSP program without having to enter the additional HTML tags.

There are three kinds of loops commonly used in a JSP program. These are the for loop, the while loop, and the do ... while loop. The for loop repeats, usually a specified number of times, although you can create an endless for loop, which you no doubt learned when you were introduced to Java .

The while loop execute continually as long as a specified condition remains true. However, the While loop may not execute because the condition may never be true. In contrast, the do ... while loop executes at least once; afterwards, the conditional expression in the do ... while loop is evaluated to determine if the loop should be executed another time.

Listing Program shows a similar routine used to populate three HTML tables with values assigned to an array.

All the tables appear the same, although a different loop is used to create each table. The JSP program initially declares and initializes an array and an integer, and then begins to create the first table.

There are two rows in each table. The first row contains three column headings that are hard coded into the program. The second row also contains three columns each of which is a value of an element of the array.

The first table is created using the for loop. The opening table row tag <TR> is entered into the program before the for loop begins. This is because the for loop is only populating columns and not rows.

A pair of HTML table data cell tags <TD> are placed inside the for loop along with a JSP tag that contains an element of the array. The JSP tag resolves to the value of the array element by the JSP virtual program.

The close table row </TR> tag and the close </TABLE> tag are inserted into the program following the French brace that closes the for loop block These tags terminate the construction of the table.

A similar process is used to create the other two tables, except the while loop and the do ... while loop are used in place of the for loop.

```
<HTML>
<HEAD>
<TITLE> JSP Programming </TITLE>
</HEAD>
<BODY>
```

```
<%! int[ ]  Grade = {100,82,93};
    int x=0;
%>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% for (int 1; i<3; i++) { %>
        <TD><%=Grade[i]%> </TD>
    <%   } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% while (x<3){ %>
        <TD><%=Grade[x]%> </TD>
        <%    x++;
      } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% x=0;
       do{ %>
           <TD><%=Grade[x]%></TD>
         <%x++;
         } while (x<3) %>
```

Request String:

The browser generates a user request string whenever the Submit button is selected. The user request string consists of the URL and the query string.

```
http://www.jimkeogh.com/jsp/myprogram.jsp?fname="Bob"&lname="Smith"
```

Your program needs to parse the query string to extract values of fields that are to be processed by your program. You can parse the query string by using methods of the JSP request object.

The getParameter(Name) is the method used to parse a value of a specific field. The getParameter() method requires an argument, which is the name of the field whose value you want to retrieve.

Let's say that you want to retrieve the value of the fname field and the value of the lname field in the previous request string. Here are the statements that you'll need in your JSP program:

```
<%! String Firstname = request.getParameter(fname);
    String Lastname = request.getParameter(lname);
%>
```

In the previous example, the first statement used the getParameter() method to copy the value of the fname from the request string and assign that value to the Firstname object. Likewise, the second statement performs a similar function, but using the value of the lname from the request string. You can use request string values throughout your program once the values are assigned to variables in your JSP program.

There are four predefined implicit objects that are in every JSP program. These are request, response, session, and out. The previous example used the request object's getParameter() method to retrieve elements of the request string. The request object is an instance of the HttpServletRequest (see Chapter 9). The response object is an instance of HttpServletResponse, and the session object is an instance of HttpSession. The out object is an instance of the JspWriter that is used to send a response to the client.

Copying a value from a multivalued field such as a selection list field can be tricky since there are multiple instances of the field name, each with a different value. However, you can easily handle multivalued fields by using the getParameterValues() method.

The getParameterValues() method is designed to return multiple values from the field specified as the argument to the getParameterValues(). Here is how the getParameterValues() is implemented in a JSP program.

In this example, we're retrieving the selection list field shown in the below code. The name of the selection list field is EMAILADDRESS, the values of which are copied into

```
<%! String [ ] EMAIL = request.getParameterValues("EMAILADDRESS ") ; %>
<P> <%= EMAIL [0]%> </P>
<P> <%= EMAIL [1]%> </P>
```

an array of String objects called EMAIL. Elements of the array are then displayed in JSP expression tags

You can parse field names by using the getParameterNames() method. This method returns an enumeration of String objects that contains the field names in the request string. You can use the enumeration extracting methods that you learned in Java to copy field names to variables within your program.

Parsing Other Information

The request string sent to the JSP by the browser is divided into two general components that are separated by the question mark. The URL component appears to the left of the question mark and the query string is to the right of the question mark. In the previous section you learned how to parse components of the query string, which are field names and values using request object methods. These are similar to the method used to parse URL information.

The URL is divided into four parts, beginning with the protocol. The protocol defines the rules that are used to transfer the request string from the browser to the JSP program.

Three of the more commonly used protocols are HTTP, HTTPS (the secured version of HTTP), and FTP, which is a file transfer protocol.

Next is the host and port combination. The host is the Internet Protocol(IP) address or name of the server that contains the JSP program:.: The port number is the port that the host monitors. Usually the port is excluded from the request string whenever HTTP is used because the assumption is the host is monitoring port 80. Following the host and port is the virtual path of the JSP program. The server maps the virtual path to the physical path.

Here's a typical URL. The http is the protocol. The host is ww_w.jimkeogh.com·. There isn't a port because the browser assumes that the server is monitoring port 80. The virtual path is / jsp/myprogram:jsp.

http:/ /www.jimkeogh.com/jsp/myprogram.jsp

User Sessions:

A JSP program must be able to track a session as a client moves between HTML pages and JSP programs as discussed in Chapter 9. There are three commonly used methods to track a session. These are by using a hidden field, by using a cookie, or by using a JavaBean.

A hidden field is a field in an HTML form whose value isn't displayed on the HTML page, as you learned in Chapter 8. You can assign a value to a hidden field in a JSP program before the program sends the dynamic HTML page to the browser.

Let's say that your JSP database system displays a dynamic login screen. The browser sends the user ID and password. to the JSP program when the Submit button is selected where these parameters are parsed and stored into two memory variables.

The JSP program then validates the login information and generates another dynamic HTML page once the user ID and password are approved. The new dynamically built HTML page contains a form that contains a hidden field, 1mong other fields. And the user ID is assigned as the value to the hidden field.

'When the person selects the Submit button on the new HTML page, the user ID stored in the hidden field and information in other fields on the form are sent by the browser to another JSP program for processing.

This cycle continues where the JSP program processing the request string receives the user ID as a parameter and then passes the user ID to the next dynamically built HTML page as a hidden field. In this way, each HTML page and subsequent JSP program has access to the user ID and therefore can track the session.

Cookies:

A cookie is a small piece of information created by a JSP program that is stored on the client's hard disk by the browser. Cookies are used to store various kinds of information, such as user preferences and an ID that tracks a session with a JSP database system.

You can create and read a cookie by using methods of the Cookie class and the response object and creates and writes a cookie called user ID that has a value of JK1234.

The program begins by initializing the cookie name and cookie value and then passes these String objects as arguments to the constructor of a new cookie. This cookie is then passed to the addCookie() method, which causes the cookie to be written to the client's hard disk. The below program retrieves a cookie and sends the cookie name and cookie value to the browser, which displays these on the screen. This program begins by initializing the MyCookieName String object to the name of the cookie that needs to be retrieved front the client's hard disk. I call the cookie userID.

Two other String objects are created to hold the name and value of the cookie read from the client. Also I created an int called found and initialized it to zero. This variable is used as a flag to indicate whether or not the userID cookie is read.

Next an array of Cookie objects called cookies is created and•c1ssigned the results of the request.getCookies() method, which reads all the cookies from the client's hard disk and assigns them to the array of Cookie objects.

The program proceeds to use the getname() and getValue() methods to retrieve the name and value from each object of the array of Cookie objects. Each time a Cookie

object is read, the program compares the name of the cookie to the value of the MyCookie Name String object, ·which is userID.

When a match is found, the program assigns the value of the current Cookie object to the MyCookieValue String object and changes the value of the found variable from 0 to 1.

After the program reads all the Cookie objects, the program evaluates the value of the found variable. If the value is 1, the program sends the value of the MyCookieName and MyCookieValue to the browser, which displays these values on the screen.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue = "JK1234";
        response.addCookie(new Cookie(MyCookieName, MyCookieValue));
    %>
  </BODY>
</HTML>
```

How to create a cookie

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue;
        String CName, CValue;
```

```
        int found=0;
        Cookie[] cookies = request.getCookies();
        for(int i=0; i<cookies.length; i++) {
          CName  = cookies[i].getName();
          CValue = cookies[i].getValue();
          if(MyCookieName.equals(cookieNames[i])) {
            found = 1;
            MyCookieValue = cookieValue;
          }
        }
        if (found ==1) { %>
          <P> Cookie name = <%= MyCookieName %> </P>
          <P> Cookie value = <%= MyCookieValue %> </P>
      <%}%>
  </BODY>
</HTML>
```

How to read a cookie

Session Objects:

AJSP database system is able to share information among JSP programs within a session by using a session object. Each time a session is created, a unique ID is assigned to the session and stored as a cookie.

The unique ID enables JSP programs to track multiple sessions simultaneously while maintaining data integrity of each session. The session ID is used to prevent the intermingling of information from clients.

In addition to the session ID, a session object is also used to store other types of information, called attributes. An attribute can be login information, preferences, or even purchases placed in an electronic shopping cart.

Let's say that you built a Java database system that enables customers to purchase goods online. A JSP program dynamically generates catalogue pages of available merchandise. A new catalogue page is generated each time the JSP program executes.

The customer selects merchandise from a catalogue page, then jumps to another catalogue page where additional merchandise is available for purchase. Your JSP database system must be able to temporarily store purchases made from each

catalogue page; otherwise, the system is unable to execute the checkout process. This means that purchases must be accessible each time the JSP program executes.

There are several ways in which you can share purchases. You might store merchandise temporally in a table, but then you'll need to access the DBMS several times during the session, which might cause performance degradation.

A better approach is to use a session object and store information about purchases as session attributes. Session attributes can be retrieved and modified each time the JSP program runs.

The below code illustrates how to assign information to a session attribute, In this example, the program creates and initializes two String objects. One String object is assigned the name of the attribute and ·the other String object is assigned a value for the attribute. Next, the program calls the setAttribute() method and passes this method the name and value of the attribute.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String AtName = "Product";
        String AtValue = "1234";
        session.setAttribute(AtName, AtValue);
    %>
  </BODY>
</HTML>
```

The below code reads attributes. The program begins by calling the getAttributeNames() method that returns names of all the attributes as an Enumeration.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! Enumeration purchases = session.getAttributeNames();
        while(purchases.hasMoreElements()){
          String AtName = (String)attributeNames.nextElement();
          String AtValue = (String)session.getAttribute(AtName); %>
          <P> Attribute Name <%= AtName %> </P>
          <P> Attribute Value <%= AtValue %> </P>
     <% } %>
  </BODY>
</HTML>
```

Next, the program tests whether or not the getAttributeNames() method returned any attributes. If so, statements within the while loop execute, which assigns the attribute name of the current element to the AtName String object. The AtName String object is then passed as an argument to the getAttribute() method, which returns the value of the attribute. The value is assigned to the AtValue String object. The program then sends the attribute name and value to the browser.