UNIT-3 - SERVLETS

Introduction to Web Servers & Servlets:

Servlets are server side components that provide a powerful mechanism for developing server side programs. Servlets provide component-based, platform-independent methods for building Web-based applications. Using servlets web developers can create fast and efficient server side application which can run on any servlet enabled web server. Servlets can access the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls; receive all the benefits of the mature java language including portability, performance, reusability, and crash protection. Today servlets are the popular choice for building interactive web applications. Servlet containers are usually the components of web and application servers, such as BEA Weblogic Application Server, IBM Web Sphere, Sun Java System Web Server, Sun Java System Application Server and others.

Servlets are not designed for a specific protocol. It is different thing that they are most commonly used with the HTTP protocols Servlets uses the classes in the java packages javax.servlet and javax.servlet.http. Servlets provides a way of creating the sophisticated server side extensions in a server as they follow the standard framework and use the highly portable java language.

HTTP Servlet typically used to:

- Provide dynamic content like getting the results of a database query and returning to the client.
- Process and/or store the data submitted by the HTML.
- Manage information about the state of a stateless HTTP. e.g. an online shopping car manages request for multiple concurrent customers.

In order to understand the advantages of servlets, you must have a basic understanding of how Web browsers and servers cooperate to provide content to a 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 database to 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 in order 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 included C, 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 for each 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 of a 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. A number of Web servers from different vendors offer the Servlet API. Programs developed for this API can be moved to any of these environments without recompilation. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. You will see that some servlets are trusted and others are untrusted. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Life Cycle Methods of Servlets

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 scenario to understand when these methods are called. First, assume that a user enters a Uniform Resource Locator (URL) to a Web browser. 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 maps this 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 pass initialization parameters to the servlet so it may configure itself. Fourth, the server invokes the **service( )** method of the servlet. This method is 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.

A Generic Servlet contains the following five methods:

init ()

public void init(ServletConfig config) throws ServletException

The init() method is called only once by the servlet container throughout the life of a servlet. By this init() method the servlet get to know that it has been placed into service.

The servlet cannot be put into the service if

- The init () method does not return within a fix time set by the web server.
- It throws a ServletException

Parameters - The init () method takes a ServletConfig object that contains the initialization parameters and servlet's configuration and throws a ServletException if an exception has occurred.

service()

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException

Once the servlet starts getting the requests, the service() method is called by the servlet container to respond. The servlet services the client's request with the help of two objects. These two objects javax.servlet.ServletRequest and javax.servlet. Servlet Response are passed by the servlet container.

The status code of the response always should be set for a servlet that throws or sends an error. Parameters - The service () method takes the ServletRequest object that contains the client's request and the object ServletResponse contains the servlet's response. The service() method throws ServletException and IOException exception.

getServletConfig ()

public ServletConfig getServletConfig()

This method contains parameters for initialization and startup of the servlet and returns a ServletConfig object. This object is then passed to the init method. When this interface is implemented then it stores the ServletConfig object in order to return it. It is done by the generic class which implements this interface.

Returns - the ServletConfig object

getServletInfo ()

public String getServletInfo ()

The information about the servlet is returned by this method like version, author etc. This method returns a string which should be in the form of plain text and not any kind of markup.

Returns - a string that contains the information about the servlet

destroy()

public void destroy()

This method is called when we need to close the servlet. That is before removing a servlet instance from service, the servlet container calls the destroy() method. Once the servlet container calls the destroy() method, no service methods will be then called . That is after the exit of all the threads running in the servlet, the destroy() method is called. Hence, the servlet gets a chance to clean up all the resources like memory, threads etc which are being held.

Life Cycle of Servlet

The life cycle of a servlet can be categorized into four parts:

1. **Loading and Instantiation:** The servlet container loads the servlet during startup or when the first request is made. The loading of the servlet depends on the attribute <load-on-startup> of web.xml file. If the attribute <load-on-startup> has a positive value then the servlet is load with loading of the container otherwise it load when the first request comes for service. After loading of the servlet, the container creates the instances of the servlet.
2. **Initialization:** After creating the instances, the servlet container calls the init() method and passes the servlet initialization parameters to the init() method. The init() must be called by the servlet container before the servlet can service any request. The initialization parameters persist until the servlet is destroyed. The init() method is called only once throughout the life cycle of the servlet.
   The servlet will be available for service if it is loaded successfully otherwise the servlet container unloads the servlet.
3. **Servicing the Request:** After successfully completing the initialization process, the servlet will be available for service. Servlet creates separate threads for each request. The servlet container calls the service() method for servicing any request. The service() method determines the kind of request and calls the appropriate method (doGet () or doPost ()) for handling the request and sends response to the client using the methods of the response object.
4. **Destroying the Servlet:** If the servlet is no longer needed for servicing any request, the servlet container calls the destroy() method . Like the init() method this method is also called only once throughout the life cycle of the servlet. Calling the destroy() method indicates to the servlet container not to sent the any request for service and the servlet releases all the

resources associated with it. Java Virtual Machine claims for the memory associated with the resources for garbage collection.

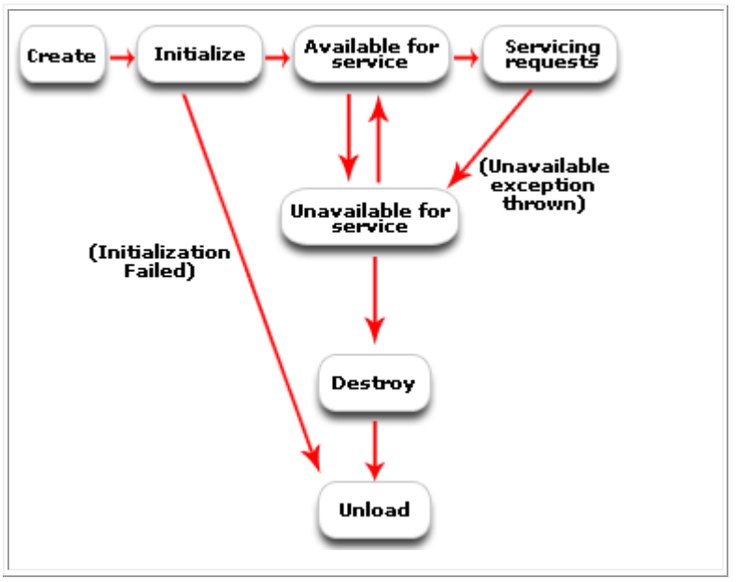


Figure: Life Cycle of a Servlet

The Advantages of Servlets

1. Portability
2. Powerful
3. Efficiency
4. Safety
5. Integration
6. Extensibility
7. Inexpensive

Each of the points are defined below:

Portability

As we know that the servlets are written in java and follow well known standardized APIs so they are highly portable across operating systems and server implementations. We can develop a servlet on Windows machine running the tomcat server or any other server and later we can deploy that servlet effortlessly on any other operating system like Unix server running on the iPlanet/Netscape Application server. So servlets are write once, run anywhere (WORA) program.

Powerful

We can do several things with the servlets which were difficult or even impossible to do with CGI, for example the servlets can talk directly to the web server while the CGI programs can't do. Servlets can share data among each other, they even make the database connection pools easy to implement. They can maintain the session by using the session tracking mechanism which helps them to maintain information from request to request. It can do many other things which are difficult to implement in the CGI programs.

Efficiency

As compared to CGI the servlets invocation is highly efficient. When the servlet get loaded in the server, it remains in the server's memory as a single object instance. However with servlets there are N threads but only a single copy of the servlet class. Multiple concurrent requests are handled by separate threads so we can say that the servlets are highly scalable.

Safety

As servlets are written in java, servlets inherit the strong type safety of java language. Java's automatic garbage collection and a lack of pointers means that servlets are generally safe from memory management problems. In servlets we can easily handle the errors due to Java's exception handling mechanism. If any exception occurs then it will throw an exception.

Integration

Servlets are tightly integrated with the server. Servlet can use the server to translate the file paths, perform logging, check authorization, and MIME type mapping etc.

Extensibility

The servlet API is designed in such a way that it can be easily extensible. As it stands today, the servlet API support Http Servlets, but in later date it can be extended for another type of servlets.

Inexpensive

There are number of free web servers available for personal use or for commercial purpose. Web servers are relatively expensive. So by using the free available web servers you can add servlet support to it.

Tomcat Web Server Installation & Configuration

In this section, we will see as how to install a WebServer, configure it and finally run servlets using this server. We will be using Apache's Tomcat server as the WebServer. Tomcat is not only an open and free server, but also the most preferred WebServer across the world. A few reasons we can attribute for its popularity is – Easy to install and configure, very less memory footprint, fast, powerful and portable. It is the ideal server for learning purpose.

1. Installation of Tomcat Server and JDK

As mentioned earlier, Apache's Tomcat Server is free software available for download @ www.apache.org. The current version of Tomcat Server is 6.0 (as of November 2007). This Server supports Java Servlets 2.5 and Java Server Pages (JSPs) 2.1 specifications. Important software required for running this server is Sun's JDK (Java Development Kit) and JRE (Java Runtime Environment). The current version of JDK is 6.0. Like Tomcat, JDK is also free and is available for download at www.java.sun.com.

2. Configuring Tomcat Server

- Set JAVA_HOME variable - You have to set this variable which points to the base installation directory of JDK installation. (e.g. c:\program file\java\jdk1.6.0). You can either set this from the command prompt or from My Computer -> Properties -> Advanced -> Environment Variables.
- Specify the Server Port – You can change the server port from 8080 to 80 (if you wish to) by editing the server.xml file in the conf folder. The path would be something like this – c:\program files\apache software foundation\tomcat6\conf\server.xml

3. Run Tomcat Server

Once the above pre-requisites are taken care, you can test as whether the server is successfully installed as follows:

Step 3.1

• Go to C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin and double click on tomcat6

OR

• Go to Start->Programs->Apache Tomcat 6.0 -> Monitor Tomcat. You will notice an icon appear on the right side of your Status Bar. Right click on this icon and click on Start service.

Step 3.2

• Open your Browser (e.g. MS Internet Explorer) and type the following URL:

http://localhost/ (If you have changed to port # to 80)

OR

• Open your Browser (e.g. MS Internet Explorer) and type the following URL: http://localhost:8080/ (If you have NOT changed the default port #)

In either case, you should get a page similar to the one in Figure-8 which signifies that the Tomcat Server is successfully running on your machine.

4. Compile and Execute your Servlet

This section through a step by step (and illustration) approach explains as how to compile and then run a servlet using Tomcat Server. Though this explanation is specific to Tomcat, the procedure explained holds true for other Web servers too (e.g. JRun, Caucho's Resin).

Step 4.1 – Compile your Servlet program

The first step is to compile your servlet program. The procedure is not different from that of writing and compiling a java program. But, the point to be noted is that neither the javax.servlet.* nor the javax.servlet.http.* is part of the standard JDK. It has to be exclusively added in the CLASSPATH. The set of classes required for writing servlets is available in a jar file called servlet-api.jar. This jar file can be downloaded from several sources. However, the easiest one is to use this jar file available with the Tomcat server (C:\Program Files\Apache Software Foundation\Tomcat 6.0\lib\servlet-api.jar).

You need to include this path in CLASSPATH. Once you have done this, you will be able to successfully compile your servlet program. Ensure that the class file is created successfully.

Step 4.2 – Create your Web application folder

The next step is to create your web application folder. The name of the folder can be any valid and logical name that represents your application (e.g. bank_apps, airline_tickets_booking, shopping_cart,etc). But the most important criterion is that this folder should be created under webapps folder. The path would be similar or close to this - C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps. For demo purpose, let us create a folder called demo-examples under the webapps folder.

Step 4.3 – Create the WEB-INF folder

The third step is to create the WEB-INF folder. This folder should be created under your web application folder that you created in the previous step. folder.

Step 4.4 – Create the web.xml file and the classes folder

The fourth step is to create the web.xml file and the classes folder. Ensure that the web.xml and classes folder are created under the WEB-INF folder.

Note – Instead of creating the web.xml file an easy way would be to copy an existing web.xml file (e.g. C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\examples\WEB-INF) and paste it into this folder. You can later edit this file and add relevant information to your web application.

Step 4.5 – Copy the servlet class to the classes folder

We need to copy the servlet class file to the classes folder in order to run the servlet that we created. All you need to do is copy the servlet class file (the file we obtained from Step 1) to this folder.

Step 4.6 – Edit web.xml to include servlet's name and url pattern

This step involves two actions viz. including the servlet's name and then mentioning the url pattern. Let us first see as how to include the servlet's name in the web.xml file.

Note – The servlet-name need not be the same as that of the class name. You can give a different name (or alias) to the actual servlet. This is one of the main reasons as why this tag is used for. Next, include the url pattern using the <servlet-mapping> </servlet-mapping> tag. The url pattern defines as how a user can access the servlet from the browser.

Step 4.7 – Run Tomcat server and then execute your Servlet

This step again involves two actions viz. running the Web Server and then executing the servlet. After ensuring that the web server is running successfully, you can run your servlet. To do this, open your web browser and enter the url as specified in the web.xml file. The complete url that needs to be entered in the browser is: Eureka! Here's the output of our first servlet. After a long and pain staking effort, we finally got an output! You can keep refreshing the browser window and see for yourself as how i value is incremented (a proof that the doGet is called every time you re-invoke a servlet).

How to run a Servlet?

To run a servlet one should follow the steps illustrated below:

- Download and Install the tomcat server: Install the tomcat server in a directory in which you want to install and set the classpath.for the variable JAVA_HOME in the environment variable. To get details about the installation process and setting the classpath click the link Tomcat installation.
- Set the class for the jar file: Set the classpath of the servlet-api.jar file in the variable CLASSPATH inside the environment variable by using the following steps.

For Windows XP,

Go to Start->Control Panel->System->Advanced->Environment Variables->New button and Set the values as
Variable Name: CLASSPATH
Variable Value: C:\Program Files\Java\Tomcat 6.0\lib\servlet-api.jar

For Windows 2000 and NT

Go to Start->Settings->Control Panel->System->Environment Variables->New button and Set the values as
Variable Name: CLASSPATH
Variable Value: C:\Program Files\Java\Tomcat 6.0\lib\servlet-api.jar

- Create a java source file and a web.xml file in a directory structure.
- Compile the java source file, put the compiled file (.class file) in the classes folder of your application and deploy the directory of your application in the webapps folder inside the tomcat directory.
- Start the tomcat server, open a browser window and type the URL http://localhost:8080/directory (folder name of your application) name/servlet name and press enter.
  If everything is correct your servlet will run.

The Servlet API

Two packages contain the classes and interfaces that are required to build servlets. These are **javax.servlet** and **javax.servlet.http**. They constitute the Servlet API. Keep in mind that these packages are not part of the Java core packages. Instead, they are standard extensions. Therefore, they are not included in the Java Software Development Kit. You must download Tomcat to obtain their functionality.

The Servlet API has been in a process of ongoing development and enhancement. The current servlet specification is version is 2.3 and that is the one used in this book. However, because changes happen fast in the world of Java, you will want to check for any additions or alterations. This chapter discusses the core of the Servlet API, which will be available to most readers. The Servlet API is supported by most Web servers, such as those from Sun, Microsoft, and others. Check at **http://java.sun.com** for the latest information.

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 the core interfaces that are provided in this package. The most significant 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

1. **Servlet** Declares life cycle methods for a servlet.
2. **ServletConfig** Allows servlets to get initialization parameters.
3. **ServletContext** Enables servlets to log events and access information about their environment.
4. **ServletRequest** Used to read data from a client request.
5. **ServletResponse** Used to write data to a client response.
6. **SingleThreadModel** Indicates that the servlet is thread safe.

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

Class Description

1. **GenericServlet** Implements the **Servlet** and **ServletConfig** interfaces.
2. **ServletInputStream** Provides an input stream for reading requests from a client.
3. **ServletOutputStream** Provides an output stream for writing responses to a client.
4. **ServletException** Indicates a servlet error occurred.
5. **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 as follows.

Method Description

1. **void destroy( )** - Called when the servlet is unloaded.
2. **ServletConfig getServletConfig( )** - Returns a **ServletConfig** object that contains any initialization parameters.
3. **String getServletInfo( )** - Returns a string describing the servlet.
4. **void init(ServletConfig *sc*) throws ServletException** - Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. An **UnavailableException** should be thrown if the servlet cannot be initialized.
5. **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 is called by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo( )** method to provide a string with useful information (for example, author, version, date, copyright). This method is also invoked by the server.

The ServletConfig Interface

The **ServletConfig** interface is implemented by the server. It allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

Method Description

1. **ServletContext getServletContext( )** - Returns the context for this servlet.
2. **String getInitParameter(String *param*)** -Returns the value of the initialization parameter named *param*
3. **Enumeration getInitParameterNames( )** - Returns an enumeration of all initialization parameter names.
4. **String getServletName( )** - Returns the name of the invoking servlet.

The ServletContext Interface

The **ServletContext** interface is implemented by the server. It enables servlets to obtain information about their environment. Several of its methods are summarized in Table 27.2.

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 path that corresponds to the virtual 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*)	Write *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*.

Table 27-2. *Various Methods Defined by* ServletContext

The ServletRequest Interface

The **ServletRequest** interface is implemented by the server. It enables a servlet to obtain information about a client request. Several of its methods are summarized in Table 27.3.

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 already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration 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 already been invoked for this request.

Table 27-3. *Various Methods Defined by* ServletRequest

The ServletResponse Interface

The **ServletResponse** interface is implemented by the server. It enables a servlet to formulate a response for a client. Several of its methods are summarized in Table 27-4.

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 already been invoked for this request.
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 already been invoked for this request.
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*.

Table 27-4. *Various Methods Defined by* ServletResponse

The SingleThreadModel Interface

This interface is used to indicate that only a single thread will execute the **service( )** method of a servlet at a given time. It defines no constants and declares no methods. If a servlet implements this interface, the server has two options. First, it can create several instances of the servlet. When a client request arrives, it is sent to an available instance of the servlet. Second, it can synchronize access to the servlet.

The GenericServlet Class

The **GenericServlet** class provides implementations of the basic life cycle methods for a servlet and is typically subclassed by servlet developers. **GenericServlet** implements the **Servlet** and **ServletConfig** interfaces. In addition, a method to append a string to the server log 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 that occurred.

The ServletInputStream Class

The **ServletInputStream** class extends **InputStream**. It is implemented by the server and provides an input stream that a servlet developer can use to read the data from a client request. It defines the default constructor. In addition, a method is provided to read bytes from the stream. Its signature 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*. The method 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 server and provides an output stream that a servlet developer can use to write data to a client response. A default constructor is defined. 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** , which indicates that a servlet problem has occurred. The second is **UnavailableException** , which extends **ServletException** . It indicates that a servlet is unavailable.

Reading Servlet Parameters

The **ServletRequest** class includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A Web page is defined in **PostParameters.htm** and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.htm** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that 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/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 and value 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 e = request.getParameterNames();
// Display parameter names and values.
while(e.hasMoreElements()) {
String pname = (String)e.nextElement();
pw.print(pname + " = ");
String pvalue = request.getParameter(pname);
pw.println(pvalue);
}
pw.close();
}
}
```

Compile the servlet and perform these steps to test this example:

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 is dynamically generated by the Servlet.

The javax.servlet.http Package

The **javax.servlet.http** package contains a number of interfaces and classes that are commonly used by servlet developers. You will see that its functionality makes it easy to build servlets that work with HTTP requests and responses. The following table summarizes the core interfaces that are provided in this package:

Interface Description

1. **HttpServletRequest** Enables servlets to read data from an HTTP request.
2. **HttpServletResponse** Enables servlets to write data to an HTTP response.
3. **HttpSession** Allows session data to be read and written.
4. **HttpSessionBindingListener** Informs an object that it is bound to or unbound from a session.

The following table summarizes the core classes that are provided in this package. 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.

1. **HttpServlet** Provides methods to handle HTTP requests and responses.
2. **HttpSessionEvent** Encapsulates a session-changed event.
3. **HttpSessionBindingEvent** Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

The HttpServletRequest Interface

The **HttpServletRequest** interface is implemented by the server. It enables a servlet to obtain information about a client request. Several of its methods are shown in Table 27-5.

Method Description

1. String getAuthType( ) Returns authentication scheme.
2. Cookie[ ] getCookies( ) Returns an array of the cookies in this request.
3. long getDateHeader(String *field*) Returns the value of the date header field named *field* .
4. String getHeader(String *field*) Returns the value of the header field named *field* .
5. Enumeration getHeaderNames( ) Returns an enumeration of the header names.
6. int getIntHeader(String *field*) Returns the **int** equivalent of the header field named *field* .
7. String getRequestedSessionId( ) Returns the ID of the session.
8. String getRequestURI( ) Returns the URI.
9. StringBuffer getRequestURL( ) Returns the URL.
10. HttpSession getSession( ) Returns the session for this request. If a session does not exist, one is created and then returned.
11. 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.

The HttpServletResponse Interface

The **HttpServletResponse** interface is implemented by the server. It enables a servlet to formulate an HTTP response 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 27-6.

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 is implemented by the server. It enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 27-7. All of these methods throw an **IllegalStateException** if the session has already been invalidated.

Method Description

- Object getAttribute(String *attr*) - Returns the value associated with the name passed in *attr* . Returns **null** if *attr* is not found.
- Enumeration getAttributeNames( ) Returns an enumeration of the attribute names associated with the session.
- long getCreationTime( ) Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
- 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 for this 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 HttpSessionBindingListener Interface

The **HttpSessionBindingListener** interface is implemented by objects that need to be notified when they are bound to or unbound from an HTTP session. The methods that are invoked when an object is bound or unbound are

void valueBound(HttpSessionBindingEvent *e*)
void valueUnbound(HttpSessionBindingEvent *e*)

Here, *e* is the event object that describes the binding.

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 the user's name, address, and other information. The user does not need to enter this 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 then included 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 of the information that is 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 is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user's machine. 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, it is 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 27-8.

Method Description

- String getDomain( ) Returns the domain.
- int getMaxAge( ) Returns the age (in seconds).
- String getName( ) Returns the name.
- boolean getSecure( ) Returns **true** if the cookie must be sent using only a secure protocol. Otherwise, returns **false**.
- String getValue( ) Returns the value.
- int getVersion( ) Returns the cookie protocol version. (Will be 0 or 1.)
- 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. Passing –1 causes the cookie to be removed when the browser is terminated.
- void setValue(String *v*) Sets the value to *v*.
- void setVersion(int v) Sets the cookie protocol version to *v* , which will be 0 or 1.

The HttpServlet Class

The **HttpServlet** class extends **GenericServlet**. It is commonly used when

developing servlets that receive and process HTTP requests. The methods of the **HttpServlet** class are summarized in Table 27-9.

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP DELETE.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP GET.

Table 27-9. *The Methods Defined by* HttpServlet

Method	Description
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP HEAD.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP OPTIONS.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP POST.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP PUT.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP TRACE.
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.

Table 27-9. *The Methods Defined by* HttpServlet (continued)

The HttpSessionEvent Class

HttpSessionEvent encapsulates session events. It extents **EventObject** and is generated when a change occurs to the session. It defines this constructor:

HttpSessionEvent(HttpSession *session*)

Here, *session* is the source of the event.

HttpSessionEvent defines one method, **getSession( )**, which is shown here:

HttpSession getSession( )

It returns the session in which the event occurred.

The HttpSessionBindingEvent Class

The **HttpSessionBindingEvent** class extends **HttpSessionEvent** . It is generated when a listener is bound to or unbound from a value in an **HttpSession** object. It is also generated when an attribute is bound or unbound. Here are its constructors:

HttpSessionBindingEvent(HttpSession *session* , String *name*)
HttpSessionBindingEvent(HttpSession *session* , String *name* , Object *val*)

Here, *session* is the source of the event and *name* is the name associated with the object that is being bound or unbound. If an attribute is being bound or unbound, its value is passed in *val* .

The **getName( )** method obtains the name that is being bound or unbound. Its is shown here:

String getName( )

The **getSession( )** method, shown next, obtains the session to which the listener is being bound or unbound:

HttpSession getSession( )

The **getValue( )** method obtains the value of the attribute that is being bound or unbound. It is shown here:

Object getValue( )

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 two files. A Web page is defined in **ColorGet.htm** and a servlet is defined in **ColorGetServlet.java** . The HTML source code for **ColorGet.htm** is shown in the 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/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 and perform these steps to test this example:

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 part of the URL that is sent to the Web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

http://localhost:8080/examples/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 servlet is invoked when a form on a Web page is submitted. The example contains two files. A Web page is defined in **ColorPost.htm** and a servlet is defined in **ColorPostServlet.java** . The HTML source code for **ColorPost.htm** is shown in the following listing. It is identical to **ColorGet.htm** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html> <body> <center>
<form name="Form1"
method="post" action="http://localhost:8080/examples/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 section to test it.

Note: Parameters for an HTTP POST request are not included as part of the URL that is sent to the Web server. In this example, the URL sent from the browser to the server is:

http://localhost:8080/examples/servlet/ColorGetServlet

The parameter names and values are sent in the body of the HTTP request.

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 three files as summarized here:

File Description

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

The HTML source code for **AddCookie.htm** is shown in the following listing. This page contains a text field in which a value can be entered. There is also 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/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 servlet and perform these steps:

1. Start Tomcat, if it is not already running.
2. Display **AddCookie.htm** 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 is displayed by the browser. Next, request the following URL via the browser:

http://localhost:8080/examples/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 cookie via the **setMaxAge( )** method of **Cookie** . Therefore, the cookie expires when the browser session ends. You can experiment by using **setMaxAge( )** and observe that the cookie is then saved to the disk 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 a server. Sessions provide such a mechanism.

As the HTTP is a stateless protocol, it can't persist the information. It always treats each request as a new request. In Http client makes a connection to the server, sends the request, gets the response, and closes the connection.

In session management client first make a request for any servlet or any page, the container receives the request and generates a unique session ID and gives it back to the client along with the response. This ID gets stores on the client machine. Thereafter when the client request again sends a request to the server then it also sends the session Id with the request. There the container sees the Id and sends back the request.

Session Tracking can be done in four ways:

1. **Hidden Form Fields:** This is one of the way to support the session tracking. As we know by the name, that in this fields are added to an HTML form which are not displayed in the client's request? The hidden form field is sent back to the server when the form is submitted. In hidden form fields the html entry will be like this : <input type ="hidden" name = "name" value="">. This means that when you submit the form, the specified name and value will be get included in get or post method. In this session ID information would be embedded within the form as a hidden field and submitted with the Http POST command.
2. **URL Rewriting:** This is another way to support the session tracking. URLRewriting can be used in place where we don't want to use cookies. It is used to maintain the session. Whenever the browser sends a request then it is always interpreted as a new request because http protocol is a stateless protocol as it is not persistent. Whenever we want that out request object to stay alive till we decide to end the request object then, there we use the concept of session tracking. In session tracking firstly a session object is created when the first request goes to the server. Then server creates a token which will be used to maintain the session. The token is transmitted to the client by the response object and gets stored on the client machine. By default the server creates a cookie and the cookie get stored on the client machine.

3. **Cookies:** When cookie based session management is used, a token is generated which contains user's information, is sent to the browser by the server. The cookie is sent back to the server when the user sends a new request. By this cookie, the server is able to identify the user. In this way the session is maintained. Cookie is nothing but a name- value pair, which is stored on the client machine. By default the cookie is implemented in most of the browsers. If we want then we can also disable the cookie. For security reasons, cookie based session management uses two types of cookies.
4. **Sessions:**

The servlet API has a built-in support for session tracking.

- **Session objects live on the server.**

– Each user has associated an HttpSession object--one user/session.

– It operates like a hashtable.

- **To get a user's existing or new session object:**

– HttpSession session = request.getSession(true);

– "true" means the server should create a new session object if necessary.

- **To store or retrieve an object in the session:**

– stores values: setAttribute("cartItem", cart);

– retrieves values: getAttribute("cartItem");

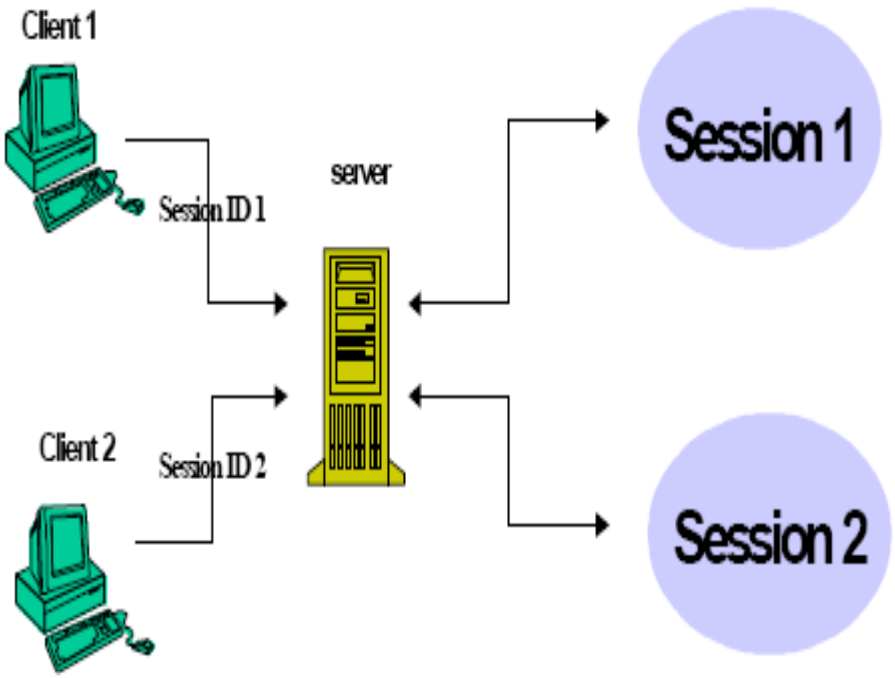
Session Tracking Usage

• When clients at an on- line store add an item to their shopping cart, how does the server know what's already in the cart?

• When clients decide to proceed to checkout, how can the server determine which previously created shopping cart is theirs?

A session can be created via the **getSession( )** method of **HttpServletRequest**. An **HttpSession** object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute( )**, **getAttribute( )**, **getAttributeNames( )**, and **removeAttribute( )** methods of **HttpSession** manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular 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 the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

Security Issues

In earlier chapters of this book, you learned that untrusted applets are constrained to operate in a “sandbox”. They cannot perform operations that are potentially dangerous to a user’s machine. This includes reading and writing files, opening sockets to arbitrary machines, calling native methods, and creating new processes. Other restrictions also apply. Similar constraints also exist for untrusted servlets. Code that is loaded from a remote machine is untrusted. However, trusted servlets are not limited in this manner. Trusted servlets are those which are loaded from the local machine.