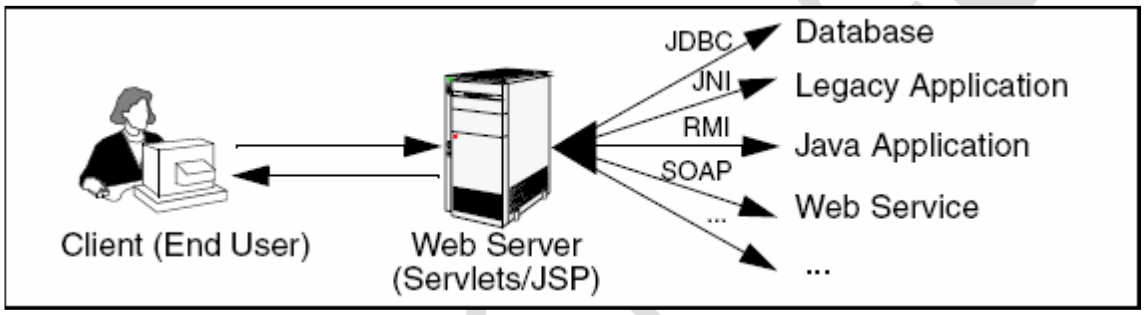
UNIT-V - Servlets

1. What is Servlet? Explain about Servlet Container?

Servlet : A servlet is a Java technology based web component, managed by a container, which generates dynamic content. Like other Java-based components, servlets are platform independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java enabled web server. Containers, sometimes called servlet engines, are web server extensions that provide servlet functionality. Servlets interact with web clients via a request/response paradigm implemented by the servlet container.

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 1.



- **Read the explicit data sent by the client** - The end user normally enters the data in an HTML form on a Web page.However, the data could also come from an applet or a custom HTTP client program.

- **Read the implicit HTTP request data sent by the browser** - Figure 1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really two varieties of data: the explicit data that the end user enters in a form and the behind-the-scenes HTTP information. Both varieties are critical. The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth.

- **Generate the results** - This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly. Your real data may be in a relational database. Fine. But your database probably doesn't speak HTTP or return results in HTML, so the Web browser can't talk directly to the database. Even if it could, for security reasons, you probably would not want it to. The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

- **Send the explicit data (i.e., the document) to the client** – This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format. But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.

- **Send the implicit HTTP response data** - Figure 1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client. But, there are really two varieties of data sent: the document itself and the behind-the-scenes HTTP information. Again, both varieties are critical to effective development. Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlet Container : The servlet container is a part of a web server or application server that provides the network services over which requests and responses are sent, decodes MIME based requests, and formats MIME based responses. A servlet container also contains and manages servlets through their lifecycle.

A servlet container can be built into a host web server, or installed as an add on component to a Web Server via that server's native extension API. Servlet containers can also be built into or possibly installed into web-enabled application servers.

All servlet containers must support HTTP as a protocol for requests and responses, but additional request/response based protocols such as HTTPS (HTTP over SSL) may be supported. The minimum required version of the HTTP specification that a container must implement is HTTP/1.0. It is strongly suggested that containers implement the HTTP/1.1 specification as well.

An Example

The following is a typical sequence of events:

1. A client (e.g., a web browser) accesses a web server and makes an HTTP request.
2. The request is received by the web server and handed off to the servlet container. The servlet container can be running in the same process as the host web server, in a different process on the same host, or on a different host from the web server for which it processes requests.
3. The servlet container determines which servlet to invoke based on the configuration of its servlets, and calls it with objects representing the request and response.
4. The servlet uses the request object to find out that the remote user is, what HTTP POST parameters may have been sent as part of this request, and other relevant data. The servlet performs whatever logic it was programmed with, and generates data to send back to the client. It sends this data back to the client via the response object.
5. Once the servlet has finished processing the request, the servlet container ensures that the response is properly flushed, and returns control back to the host web server.

Comparing Servlets with Other Technologies

In functionality, servlets lie somewhere between Common Gateway Interface (CGI) programs and proprietary server extensions such as the Netscape Server API (NSAPI) or Apache Modules.

Servlets have the following advantages over other server extension mechanisms:

1. They are generally much faster than CGI scripts because a different process model is used.
2. They use a standard API that is supported by many web servers.
3. They have all the advantages of the Java programming language, including ease of development and platform independence.
4. They can access the large set of APIs available for the Java platform.

5.1.1 Advantage of Servlets Over "Traditional" CGI?

Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies. (More importantly, servlet developers get paid more than Perl programmers :-).

1. Efficient. With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time.With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class. Servlets also have more alternatives than do regular CGI programs for optimizations such as caching previous computations, keeping database connections open, and the like.

2. Convenient. Hey, you already know Java. Why learn Perl too? Besides the convenience of being able to use a familiar language, servlets have an extensive infrastructure for utomatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

3. Powerful. Java servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement. They can also maintain information from request to request, simplifying things like session tracking and caching of previous computations.

4. Portable. Servlets are written in Java and follow a well-standardized API. Consequently, servlets
written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or

WebStar. Servlets are supported directly or via a plugin on almost every major Web server.

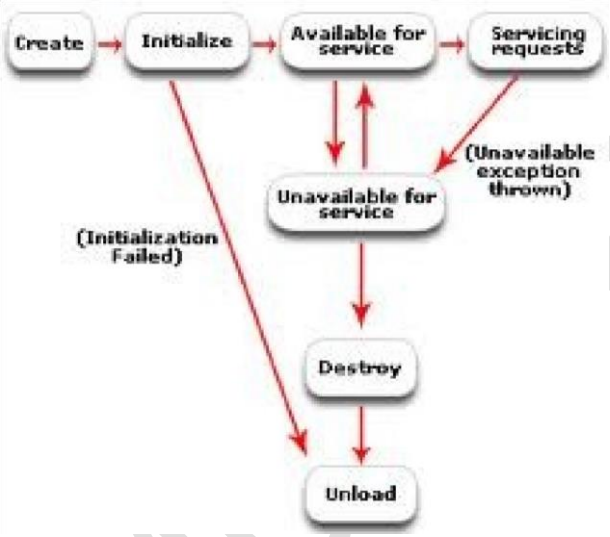
5. Inexpensive. There are a number of free or very inexpensive Web servers available that are good for "personal" use or low-volume Web sites. However, with the major exception of Apache, which is free, most commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter the cost of that server, adding servlet support to it (if it doesn't come preconfigured to support servlets) is generally free or cheap.

2. **Explain Servlet Life Cycle Methods.**

Servlets run on the web server platform as part of the same process as the web server itself. The web server is responsible for initializing, invoking, and destroying each servlet instance.

A web server communicates with a servlet through a simple interface, javax.servlet.Servlet. This interface consists of three main methods:

- init()
- service()
- destroy()



Servlet Life Cycle Architecture

The init() Method

When a servlet is first loaded, its init () method is invoked. This allows the servlet to per form any setup processing such as opening files or establishing connections to their servers. If a servlet has been permanently installed in a server, it loads when the server starts to run. Otherwise, the server activates a servlet when it receives the first client request for the services provided by the servlet.

The init() method is guaranteed to finish before any other calls are made to the servlet-such as a call to the service() method. Note that init() will only be called once; it will not be called again unless the servlet has been unloaded and then reloaded by the server.

The init() method takes one argument, a reference to a ServletConfig object which provides initialization arguments for the servlet. This object has a method getServletContext() that returns a ServletContext object containing information about the servlet's environment (see the discussion on Servlet Initialization Context below).

The service() Method

The service() method is the heart of the servlet. Each request message from a client results in a single call to the servlet's service () method. The service() method reads the request and produces the response message from its two parameters:

- A ServletRequest object with data from the client. The data consists of name/value pairs of parameters and an InputStream. Several methods are provided that return the client's parameter information. The InputStream from the client can be obtained via the getInputStream() method. This method returns a ServletInputStream, which can be used to get additional data from the client. If you are interested in processing character-level data instead of byte-level data, you can get a BufferedReader instead with getReader().
- A ServletResponse represents the servlet's reply back to the client. When preparing a response, the method setContentType() is called first to set the MIME type of the reply. Next, the method getOutputStream() or getWriter() can be used to obtain a ServletOutputStream or PrintWriter, respectively, to send data back to the client.

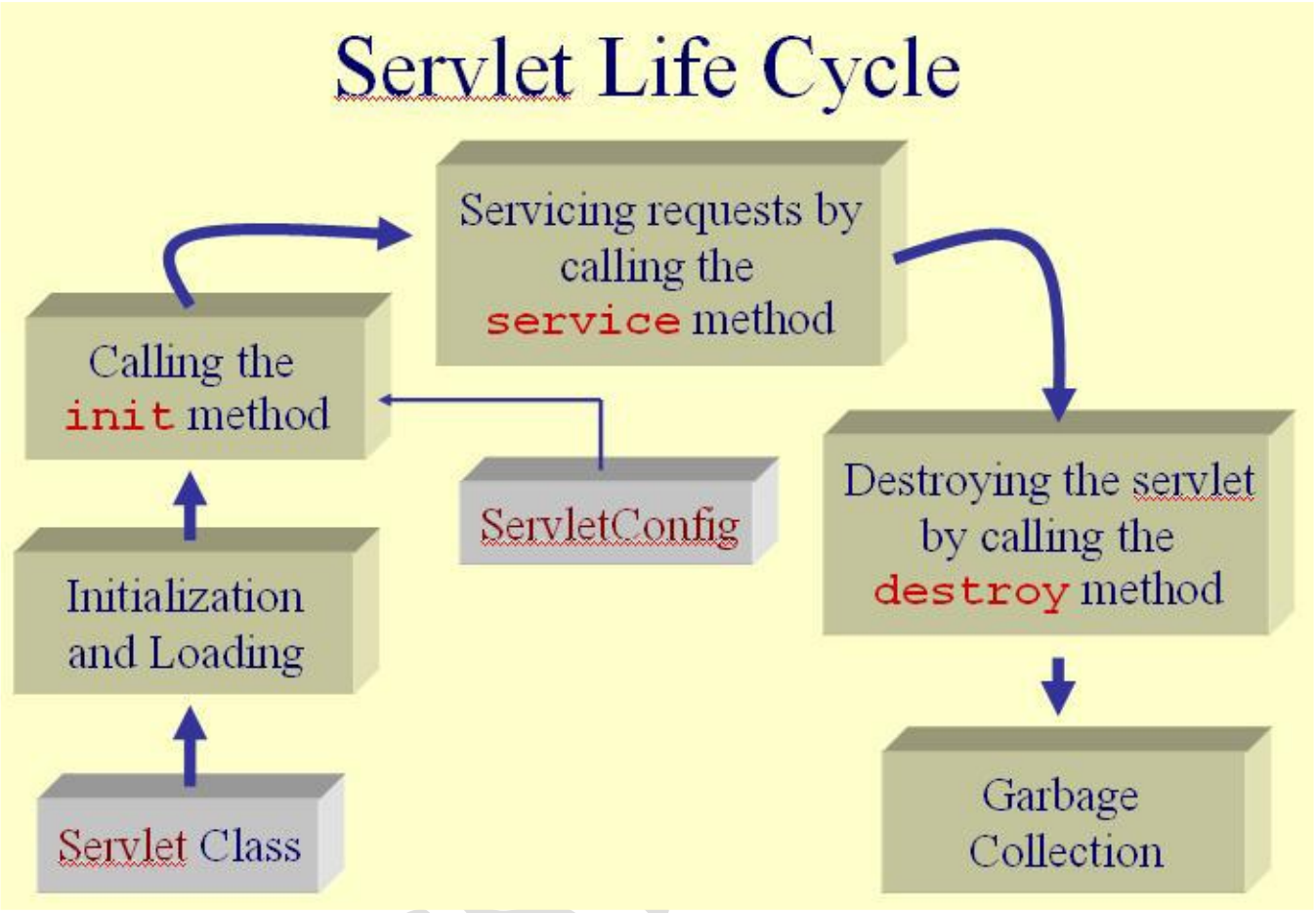
As you can see, there are two ways for a client to send information to a servlet. The first is to send parameter values and the second is to send information via the InputStream (or Reader). Parameter values can be embedded into a URL. How this is done is discussed below. How the parameter values are read by the servlet is discussed later.

The service() method's job is conceptually simple--it creates a response for each client request sent to it from the host server. However, it is important to realize that there can be multiple service requests being processed at once. If your service method requires any outside resources, such as files, databases, or some external data, you must ensure that resource access is thread-safe. Making your servlets thread-safe is discussed in a later section of this course.

The destroy() Method

The destroy() method is called to allow your servlet to clean up any resources (such as open files or database connections) before the servlet is unloaded. If you do not require any clean-up operations, this can be an empty method.

The server waits to call the destroy() method until either all service calls are complete, or a certain amount of time has passed. This means that the destroy() method *can* be called while some longer-running service() methods are still running. It is important that you write your destroy() method to avoid closing any necessary resources until all service() calls have completed.



Sample Servlet

The code below implements a simple servlet that returns a static HTML page to a browser. This example fully implements the Servlet interface.

```
import java.io.*; import
javax.servlet.*;
public class SampleServlet implements Servlet { private
ServletConfig config;

  public void init (ServletConfig config)
    throws ServletException {
this.config = config; }

  public void destroy() {} // do nothing

  public void service (ServletRequest req,
    ServletResponse res
```

```
 ) throws ServletException, IOException {
res.setContentType( "text/html" ); PrintWriter out =
res.getWriter(); out.println( "<html>" );
out.println( "<head>" );
   out.println( "<title>A Sample Servlet</title>" );
   out.println( "</head>" ); out.println( "<body>"
);
   out.println( "<h1>A Sample Servlet</h1>" );
out.println( "</body>" ); out.println( "</html>" );
out.close();
 }
}
```

5.3 THE JAVA SERVLET API

The Java Servlet API is a set of Java classes which define a standard interface between a Web client and a
Web servlet. Client requests are made to the Web server, which then invokes the servlet to service the request through this interface. The Java Servlet API is a Standard Java Extension API, meaning that it is not part of the core Java framework, but rather, is available as an add-on set of packages. The API is composed of two packages:

- *javax.servlet*
- *javax.servlet.http*

The *javax.servlet* package contains classes to support generic protocol-independent servlets. This means that servlets can be used for many protocols, for example, HTTP and FTP. The *javax.servlet.http* package extends the functionality of the base package to include specific support for the HTTP protocol.
Each time a client request is made, a new servlet thread is spawned which services the request. In this way, the server can handle multiple concurrent requests to the same servlet. For each request, usually the *service, doGet*, or *doPost* methods will be called. These methods are passed the *HttpServletRequest* and *HttpServletResponse* parameter objects.

***doPost*:** Invoked whenever an HTTP POST request is issued through an HTML form. The parameters associated with the POST request are communicated from the browser to the server as a separate HTTP request. The *doPost* method should be used whenever modifications on the server will take place.

***doGet*:** Invoked whenever an HTTP GET method from a URL request is issued, or an HTML form. An HTTP GET method is the default when a URL is specified in a Web browser. In contrast to the *doPost* method, *doGet* should be used when no modifications will be made on the server, or when the parameters are not sensitive data. The parameters associated with a GET request are appended to the end of the URL, and are passed into the QueryString property of the *HttpServletRequest*.

3. Explain Servlet Protocol Support?

The Servlet API provides a tight link between a server and servlets. This allows servlets to add new protocol support to a server. (You will see how HTTP support is provided for you in the API packages.) Essentially, any protocol that follows a request/response computing model can be implemented by a servlet. This could include:

- SMTP

- POP
- FTP

Servlet support is currently available in several web servers, and will probably start appearing in other types of application servers in the near future. You will use a web server to host the servlets in this class and only deal with the HTTP protocol.

Because HTTP is one of the most common protocols, and because HTML can provide such a rich presentation of information, servlets probably contribute the most to building HTTP based systems.

HTML Support

HTML can provide a rich presentation of information because of its flexibility and the range of content that it can support. Servlets can play a role in creating HTML content. In fact, servlet support for HTML is so common, the javax.servlet.http package is dedicated to supporting HTTP protocol and HTML generation.

Complex web sites often need to provide HTML pages that are tailored for each visitor, or even for each hit. Servlets can be written to process HTML pages and customize them as they are sent to a client. This can be as simple as on the fly substitutions or it can be as complex as compiling a grammar-based description of a page and generating custom HTML.

HTTP Support

Servlets that use the HTTP protocol are very common. It should not be a surprise that there is specific help for servlet developers who write them. Support for handling the HTTP protocol is provided in the package javax.servlet.http. Before looking at this package, take a look at the HTTP protocol itself.

HTTP stands for the HyperText Transfer Protocol. It defines a protocol used by web browsers and servers to communicate with each other. The protocol defines a set of text-based request messages called *HTTP methods*. (Note: The HTTP specification calls these *HTTP methods*; do not confuse this term with Java methods. Think of *HTTP methods* as messages requesting a certain type of response). The HTTP methods include:

- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- CONNECT
- OPTIONS

For this course, you will only need to look at only three of these methods: GET, HEAD, and POST.

The HTTP GET Method

The HTTP GET method requests information from a web server. This information could be a file, output from a device on the server, or output from a program (such as a servlet or CGI script).

The POST Method

An HTTP POST request allows a client to send data to the server. This can be used for several purposes, such as

- Posting information to a newsgroup
- Adding entries to a web site's guest book
- Passing more information than a GET request allows

Pay special attention to the third bullet above. The HTTP GET request passes all its arguments as part of the URL. Many web servers have a limit to how much data they can accept as part of the URL. The POST method passes all of its parameter data in an input stream, removing this limit.

4. **Explain Servlet Interface?**

The Servlet interface is the central abstraction of the servlet API. All servlets implement this interface either directly, or more commonly, by extending a class that implements the interface. The two classes in the servlet API that implement the Servlet interface are GenericServlet and HttpServlet. For most purposes, developers will extend HttpServlet to implement their servlets.

Request Handling Methods

The basic Servlet interface defines a service method for handling client requests. This method is called for each request that the servlet container routes to an instance of a servlet.

The handling of concurrent requests to a web application generally requires the web developer design servlets that can deal with multiple threads executing within the service method at a particular time.

Generally the web container handles concurrent requests to the same servlet by concurrent execution of the service method on different threads.

HTTP Specific Request Handling Methods

The HttpServlet abstract subclass adds additional methods beyond the basic Servlet interface which are automatically called by the service method in the HttpServlet class to aid in processing HTTP based requests. These methods are:

- doGet for handling HTTP GET requests
- doPost for handling HTTP POST requests
- doPut for handling HTTP PUT requests

- doDelete for handling HTTP DELETE requests
- doHead for handling HTTP HEAD requests
- doOptions for handling HTTP OPTIONS requests
- doTrace for handling HTTP TRACE requests

Typically when developing HTTP based servlets, a Servlet Developer will only concern himself with the doGet and doPost methods. The other methods are considered to be methods for use by programmers very familiar with HTTP programming.

Example:

```
import java.io.*; import
javax.servlet.*; import
javax.servlet.http.*;
/** Simple servlet for testing the use of packages. */

public class HelloServlet2 extends HttpServlet { public void doGet(HttpServletRequest
request,HttpServletResponse
        response) throws ServletException, IOException {
                response.setContentType("text/html"); PrintWriter out =
                response.getWriter();
                out.println("<HTML><HEAD><TITLE>");
                out.println("Hello (2)</TITLE></HEAD>");
                out.println("<BODY BGCOLOR=\"#FDF5E6\">");
                out.println("<H1>Hello (2)</H1>");
                out.println("</BODY></HTML>");
        }
}
```

Example: Write a Servlet program to display 'Hello World'.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet
{
public void doGet(
HttpServletRequest req,HttpServletResponse res)
throws IOException, ServletException
{
PrintWriter out = res.getWriter();
out.println("<html><head><title>First Servlet
</title></head>");
out.println("<b>HelloServlet</b></html>");
}
}
```

Example: Write a Servlet program to display the current date.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;
public class DateServlet extends HttpServlet
{
public void doGet(
HttpServletRequest req,HttpServletResponse res)
throws IOException,ServletException
{
res.setContentType("Text/Html");
PrintWriter out=res.getWriter();
Date d=new Date();
out.println("<Html><Head><Title>Today
</Title></Head>");
out.println("<Body><H1> Date: "+d+"</H1>");
out.flush();
out.println("</Body></Html>");
}
}
```

Servlet Context

A servlet lives and dies within the bounds of the server process. To understand its operating environment, a servlet can get information about its environment at different times. Servlet initialization information is available during servlet start-up; information about the hosting server is available at any time; and each service request can contain specific contextual information.

Server Context Information

Server context information is available at any time through the ServletContext object.

A servlet can obtain this object by calling the getServletContext() method on the ServletConfig object. Remember that this was passed to the servlet during the initialization phase. A well written init() method saves the reference in a private variable.

The ServletContext interface defines several methods. These are outlined below.

getAttribute()	An extensible way to get information about a server via attribute name/value pairs. This is server specific.
getMimeType()	Returns the MIME type of a given file. getRealPath() This method translates a relative or virtual path to a new path relative to the server's HTML documentation root location.

getServerInfo()	Returns the name and version of the network service under which the servlet is running.
getServlet()	Returns a Servlet object of a given name. Useful when you want to access the services of other servlets.
getServletNames()	Returns an enumeration of servlet names available in the current namespace.
log()	Writes information to a servlet log file. The log file name and format are server specific.

The Request :

The request object encapsulates all information from the client request. In the HTTP protocol, this information is transmitted from the client to the server in the HTTP headers and the message body of the request.

HTTP Protocol Parameters

Request parameters for the servlet are the strings sent by the client to a servlet container as part of its request. When the request is a HttpServletRequest object, and conditions set out below are met, the container populates the parameters from the URI query string and POST-ed data.

The parameters are stored as a set of name-value pairs. Multiple parameter values can exist for any given parameter name. The following methods of the ServletRequest interface are available to access parameters:

- getParameter
- getParameterNames
- getParameterValues

The getParameterValues method returns an array of String objects containing all the parameter values associated with a parameter name. The value returned from the getParameter method must be the first value in the array of String objects returned by getParameterValues.

Attributes

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via the RequestDispatcher). Attributes are accessed with the following methods of the ServletRequest interface:

- getAttribute
- getAttributeNames
- setAttribute

Only one attribute value may be associated with an attribute name.

Attribute names beginning with the prefixes of “java.” and “javax.” Are reserved for definition by this specification. Similarly attribute names beginning with the prefixes of “sun.”, and “com.sun.” are reserved for definition by Sun Microsystems. It is suggested that all attributes placed into the

attribute set be named in accordance with the reverse domain name convention suggested by the Java Programming Language Specification1 for package naming.

Example program for getting user requests HTML

File

```
< html >
    < head >
            < title> Sample Program </title >
    < /head >
    < body >
            < form name="loginForm" action="CheckValidUser"  >
            Enter Name : <input type=text name="user" > <p>
            Enter Password : <input type=password name="pwd" ><p>
            < input type=submit value=" Send"  >
            < input type=reset value="Clear"  >
            < /form >
    < /body >
< /html >
```

Servlet File

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

public class CheckValidUser extends GenericServlet
{ public void service(ServletRequest req, ServletResponse res) throws
    ServletException ,IOException
    {
            PrintWriter out = res.getWriter();
            String user = req.getParameter("user");
            String pwd = req.getParameter("pwd"); if (user.equals("bhagavan")
            && pwd.equals("ram")) out.println("<br><h1> Welcome to
            Trendz</h1> "); else out.println(" Invalid User ");
    }
}
```

Example program to get output stream from HttpServletResponse

```
import      java.io.ByteArrayOutputStream;
    import  java.io.DataInputStream;  import
    java.io.DataOutputStream;         import
    java.io.IOException;

    import      javax.servlet.ServletException;     import
    javax.servlet.ServletOutputStream;              import
    javax.servlet.http.HttpServlet;                 import
    javax.servlet.http.HttpServletRequest;          import
```

```
javax.servlet.http.HttpServletResponse;          import
javax.servlet.http.HttpSession;

public class CounterServer extends HttpServlet {    static final
String COUNTER_KEY = "Counter.txt";

  public void doPost(HttpServletRequest req, HttpServletResponse resp) th
rows ServletException,IOException {
   HttpSession session = req.getSession(true);
   int count = 1 ;
   Integer i = (Integer) session.getAttribute(COUNTER_KEY);
   if (i != null) {
    count = i.intValue() + 5 ;
   }
   session.setAttribute(COUNTER_KEY, new Integer(count));
   DataInputStream in = new DataInputStream(req.getInputStream());
resp.setContentType("application/octet-stream");
   ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
   DataOutputStream out = new DataOutputStream(byteOut);
   out.writeInt(count);    out.flush();
   byte[] buf = byteOut.toByteArray();    resp.setContentLength(buf.length);
   ServletOutputStream servletOut = resp.getOutputStream();
servletOut.write(buf);    servletOut.close();
  }
}
```

5. Explain Servlet RequestDispatcher Interface?

Dispatching Requests

When building a web application, it is often useful to forward processing of a request to another servlet, or to include the output of another servlet in the response.

An object implementing the RequestDispatcher interface may be obtained from the ServletContext via the following methods:

- getRequestDispatcher
- getNamedDispatcher

The getRequestDispatcher method takes a String argument describing a path within the scope of the ServletContext. This path must be relative to the root of the ServletContext and begin with a '/'. The method uses the path to look up a servlet, wraps it with a RequestDispatcher object, and returns the resulting object. If no servlet can be resolved based on the given path, a RequestDispatcher is provided that returns the content for that path.

The getNamedDispatcher method takes a String argument indicating the name of a servlet known to the ServletContext. If a servlet is found, it is wrapped with a RequestDispatcher object and the object returned. If no servlet is associated with the given name, the method must return null.

To allow RequestDispatcher objects to be obtained using relative paths that are relative to the path of the current request (not relative to the root of the ServletContext), the following method is provided in the ServletRequest interface:

• getRequestDispatcher

The behavior of this method is similar to the method of the same name in the ServletContext. The servlet container uses information in the request object to transform the given relative path against the current servlet to a complete path. For example, in a context rooted at '/' and a request to /garden/tools.html, a request dispatcher obtained via ServletRequest.getRequestDispatcher("header.html") will behave exactly like a call to ServletContext.getRequestDispatcher("/ garden/header.html").

Using a Request Dispatcher

To use a request dispatcher, a servlet calls either the include method or forward method of the RequestDispatcher interface. The parameters to these methods can be either the request and response arguments that were passed in via the service method of the Servlet interface, or instances of subclasses of the request and response wrapper classes that have been introduced for version 2.3 of the specification. In the latter case, the wrapper instances must wrap the request or response objects that the container passed into the service method.

The Container Provider must ensure that the dispatch of the request to a target servlet occurs in the same thread of the same VM as the original request.

The Include Method

The include method of the RequestDispatcher interface may be called at any time. The target servlet of the include method has access to all aspects of the request object, but its use of the response object is more limited:

It can only write information to the ServletOutputStream or Writer of the response object and commit a response by writing content past the end of the response buffer, or by explicitly calling the flushBuffer method of the ServletResponse interface. It cannot set headers or call any method that affects the headers of the response. Any attempt to do so must be ignored.

Example for Include Method:

```
import javax.servlet.RequestDispatcher; import
javax.servlet.ServletException; import
javax.servlet.http.HttpServlet; import
javax.servlet.http.HttpServletRequest; import
javax.servlet.http.HttpServletResponse; public class
MultipleInclude extends HttpServlet {

 protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException,     java.io.IOException {

  response.setContentType("text/html");     java.io.PrintWriter out =
response.getWriter();
```

```
   out.println("<html>");     out.println("<head>");
   out.println("<title>Multiple Includes</title>");
   out.println("</head>");     out.println("<body>");
   out.println("<h1>Hello from Level 1< /h1>");     out.println("This text is
displayed at Level 1 .");
   RequestDispatcher          dispatcher          =          request.getRequestDispatcher("/Level4");
dispatcher.include(request, response);
   out.println("</body>");
out.println("</html>");     out.close();

  }
}

//       here       is       another       servlet       import
javax.servlet.ServletException;                    import
javax.servlet.http.HttpServlet;                    import
javax.servlet.http.HttpServletRequest;             import
javax.servlet.http.HttpServletResponse;

public class Level4 extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response
)
   throws ServletException, java.io.IOException {

       java.io.PrintWriter out = response.getWriter();          out.println("<h4>Hello from
another doGet</h4>");     out.println("Hello from another doGet.");
  }
}
```

web.xml

```
  < servlet >
      < servlet-name>MultipleInclude</servlet-name >
      < servlet-class>MultipleInclude</servlet-class >
   </servlet>
   < servlet-mapping >
      < servlet-name>MultipleInclude</servlet-name >
      < url-pattern>/MultipleInclude</url-pattern >
   < /servlet-mapping >
   < servlet >
      < servlet-name>Level4</servlet-name >
      < servlet-class>Level4</servlet-class >
   < /servlet >
   < servlet-mapping >
      < servlet-name>Level4</servlet-name >
      < url-pattern>/Level4</url-pattern >
   < /servlet-mapping >
```

The Forward Method

The forward method of the RequestDispatcher interface may be called by the calling servlet only when no output has been committed to the client. If output data exists in the response buffer that has not been committed, the content must be cleared before the target servlet's service method is called. If the response has been committed, an IllegalStateException must be thrown.

The path elements of the request object exposed to the target servlet must reflect the path used to obtain the RequestDispatcher.

The only exception to this is if the RequestDispatcher was obtained via the getNamedDispatcher method. In this case, the path elements of the request object must reflect those of the original request.

Before the forward method of the RequestDispatcher interface returns, the response content must be sent and committed, and closed by the servlet container.

Example program for forward method and sendRedirect method:

loginSS.html file :

```
< html >
        < head >
                < title> Sample Forward Program </title >
        < /head >
        < body >
                < form name="loginForm" action="CheckUser"  >
                Enter Name : <input type=text name="user" > <p>
                Enter Password : <input type=password name="pwd" ><p>
                < input type=submit value=" Send"  >
                < input type=reset value="Clear"  >
                < /form >
        < /body >
< /html >
```

CheckUser.java servlet program:

```
import  javax.servlet.http.*;  import
javax.servlet.*; import java.io.*;
public class CheckUser extends HttpServlet
{  public  void  doGet(HttpServletRequest  req,  HttpServletResponse  res)  throws
ServletException ,IOException
        {
                RequestDispatcher disp;
                PrintWriter out = res.getWriter();
                String user = req.getParameter("user");
                String pwd = req.getParameter("pwd");
                if (user.equals("bhagavan") && pwd.equals("ram"))
                {
```

```
                    ServletContext context = getServletContext();  disp =
                    context.getRequestDispatcher("/Welcome");
                    disp.forward(req,res);
                } else
                { res.sendRedirect("loginSS.html");
                }
        }
        public void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException ,IOException
        { doGet(req,res);
        }
}
```

Welcome.java ServletPorgram:

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

public class Welcome extends HttpServlet
{ public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException ,IOException
        {
                PrintWriter out = res.getWriter();
                String user = req.getParameter("user"); out.println(" Welcome
                "+user);
        }
        public void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException ,IOException
        { doGet(req,res);
        }
}
```

Web.xml:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3 //EN" "http://java.sun.com/dtd/web-app_2_3.dtd" >
< web-app >
  < servlet >
    < servlet-name>CheckUser</servlet-name >
    < servlet-class>CheckUser</servlet-class >
  < /servlet >
  < servlet >
    < servlet-name>Welcome</servlet-name >
    < servlet-class>Welcome</servlet-class >
  < /servlet >
  < servlet-mapping >
    < servlet-name>CheckUser</servlet-name >
    < url-pattern>/CheckUser/*</url-pattern >
  < /servlet-mapping >
```

```
  < servlet-mapping >
    < servlet-name>Welcome</servlet-name >
    < url-pattern>/Welcome/*</url-pattern >
  < /servlet-mapping >
< /web-app >
```

6. **Explain Servlet SessionMangement Mechanisms ?**

A session is a collection of HTTP requests, over a period of time, b/w a client and a web server. Lifetime of a session specifies the period of time till the session is active. When a session expires, the session is destroyed. Its resources are returned back to the Servlet engine. Session tracking is a mechanism used tomaintain the state of a user within a series of requests across some period i.e. the lifetime of a session. Means to keep track of session data. Session data represents the data being transferred in a session. Session tracking is used when session data from one session may be required by a web server to complete specific operations in the same or different sessions. Different mechanism for session tracking:

· URL rewriting
· Hidden form fields
· Cookies
· HTTP session.

The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective web applications, it is imperative that requests from a particular client be associated with each other. Many strategies for session tracking have evolved over time, but all are difficult or troublesome for the programmer to use directly.

This specification defines a simple HttpSession interface that allows a servlet container to use any of several approaches to track a user's session without involving the Application Developer in the nuances of any one approach.

Session Tracking Mechanisms

The following sections describe approaches to tracking a user's sessions

1) Cookies

Cookies are small bits of textual information that a web server sends to a browser and that the browser later returns unchanged when visiting the same website or domain.

You can use HTTP cookies to store information about a shopping session, and each subsequent connection can look up the current session and then extract information about that session from some location on the server machine.

This is an excellent alternative, and is the most widely used approach. However, even though Servlets have a high-level and easy-to-use interface to cookies high-level and easy-to-use interface to cookies, there are still a number of relatively tedious details that need to be handled:

• Extracting the cookie that stores the session identifier from the other cookies (there may be many, after all),
• Setting an appropriate expiration time for the cookie (sessions interrupted by 24 hours probably should be reset), and
• Associating information on the server with the session identifier (there may be far too much information to actually store it in the cookie, plus sensitive data like credit card numbers should *never* go in cookies).

Session tracking through HTTP cookies is the most used session tracking mechanism and is required to be supported by all servlet containers.

The container sends a cookie to the client. The client will then return the cookie on each subsequent request to the server, unambiguously associating the request with a session. The name of the session tracking cookie must be JSESSIONID.

2) URL Rewriting

URL rewriting is the lowest common denominator of session tracking. When a client will not accept a cookie, URL rewriting may be used by the server as the basis for session tracking.

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies.

URL rewriting involves adding data, a session id, to the URL path that is interpreted by the container to associate the request with a session.

The session id must be encoded as a path parameter in the URL string. The name of the parameter must be jsessionid. Here is an example of a URL containing encoded path information: http://www.myserver.com/catalog/index.html;jsessionid=1234

3) **Hidden form fields.**

HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">. This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session. However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

4) Creating a Session

It provides a way to identify a user across more than one page request or visit to a Web site. The servlet engine uses this interface to create a session between an HTTP client and an HTTP server. The session

persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times.
The server can maintain a session either by using cookies or by rewriting URLs.

This interface allows servlets to

- View and manipulate information about a session, such as the ession identifier, creation time, or context.
- Bind objects to sessions, allowing you to use an online shopping cart to hold data that persists across multiple user connections

HttpSession defines methods that store these types of data:

- Standard session properties, such as a session identifier or session context.
- Data that the application provides, accessed using this interface and stored using a dictionary-like interface

An HTTP session represents the server's view of the session. The server considers a session new under any of these conditions:

- The client does not yet know about the session.
- The session has not yet begun.
- The client chooses not to join the session, for example, if the server supports only cookies and the client rejects the cookies the server sends

When the session is new, the isNew() method returns true.

Method Summary:

String getId() - Returns a string containing the unique identifier assigned to this session.
Object getValue(String name) - Returns the object bound with the specified name in this session or null if no object of that name exists.
String[]getValueNames()- Returns an array containing the names of all the objects bound to this session.
boolean isNew() - Returns true if the Web server has created a session but the client has not yet joined.
void setAttribute(String name, Object value) - This method binds an object to this session, using the name specified.

Object getAttribute(String name) - This method returns the object bound with the specified name in this session, or null if no object is bound under the name

A session is considered "new" when it is only a prospective session and has not been established. Because HTTP is a request-response based protocol, an HTTP session is considered to be new until a client "joins" it. A client joins a session when session tracking information has been returned to the server indicating that a session has been established. Until the client joins a session, it cannot be assumed that the next request from the client will be recognized as part of a session.

The session is considered "new" if either of the following is true: •

The client does not yet know about the session

• The client chooses not to join a session.

These conditions define the situation where the servlet container has no mechanism by which to associate a request with a previous request.

A Servlet Developer must design his application to handle a situation where a client has not, cannot, or will not join a session.

Session Timeouts

In the HTTP protocol, there is no explicit termination signal when a client is no longer active. This means that the only mechanism that can be used to indicate when a client is no longer active is a timeout period.

The default timeout period for sessions is defined by the servlet container and can be obtained via the getMaxInactiveInterval method of the HttpSession interface. This timeout can be changed by the Developer using the setMaxInactiveInterval method of the HttpSession interface. The timeout periods used by these methods are defined in seconds. By definition, if the timeout period for a session is set to -1, the session will never expire.

Distributed Environments

Within an application marked as distributable, all requests that are part of a session must handled by one virtual machine at a time. The container must be able to handle all objects placed into instances of the HttpSession class using the setAttribute or putValue methods appropriately. The following restrictions are imposed to meet these conditions:

- The container must accept objects that implement the Serializable interface
- The container may choose to support storage of other designated objects in the HttpSession, such as references to Enterprise JavaBean components and transactions.
- Migration of sessions will be handled by container-specific facilities.

The servlet container may throw an IllegalArgumentException if an object is placed into the session that is not Serializable or for which specific support has not been made available. The IllegalArgumentException must be thrown for objects where the container cannot support the mechanism necessary for migration of a session storing them.

These restrictions mean that the Developer is ensured that there are no additional concurrency issues beyond those encountered in a non-distributed container.

Example program for Session Class Object.

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

public class SessionExample extends HttpServlet {
  ResourceBundle rb = ResourceBundle.getBundle("LocalStrings");  public  void
doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {  response.setContentType("text/html");

   PrintWriter out = response.getWriter();
   out.println("<html>");
   out.println("<body bgcolor=\"white\">");  out.println("<head>");

   String title = rb.getString("sessions.title");  out.println("<title>" + title + "</title>");
   out.println("</head>");  out.println("<body>");

   out.println("<a href=\"../sessions.html\">");
   out.println("<img src=\"../images/code.gif\" height=24 "
     +  "width=24 align=right border=0 alt=\"view code\"></a>");
   out.println("<a href=\"../index.html\">");
   out.println("<img src=\"../images/return.gif\" height=24 "  +  "width=24 align=right
border=0 alt=\"return\"></a>");  out.println("<h3>" + title + "</h3>");


   HttpSession session = request.getSession(true);
   out.println(rb.getString("sessions.id") + " " + session.getId());
   out.println("<br>");
   out.println(rb.getString("sessions.created")  +  "  ");  out.println(new
Date(session.getCreationTime())  +  "<br>");
out.println(rb.getString("sessions.lastaccessed")  +  "  ");  out.println(new
Date(session.getLastAccessedTime()));

   String dataName = request.getParameter("dataname");
   String dataValue = request.getParameter("datavalue");  if (dataName
!= null && dataValue != null) {  session.setAttribute(dataName,
dataValue);
   }
   out.println("<P>");
   out.println(rb.getString("sessions.data") + "<br>");
   Enumeration names = session.getAttributeNames();
   while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
```

```
    String value = session.getAttribute(name).toString();
out.println(HTMLFilter.filter(name) + " = " + HTMLFilter.filter(value) + "<br>");
   }
   out.println("<P>"); out.print("<form
action=\"");
   out.print(response.encodeURL("SessionExample"));
   out.print("\" "); out.println("method=POST>");
   out.println(rb.getString("sessions.dataname")); out.println("<input type=text
size=20 name=dataname>");
   out.println("<br>");
   out.println(rb.getString("sessions.datavalue")); out.println("<input type=text size=20
name=datavalue>"); out.println("<br>");
   out.println("<input type=submit>"); out.println("</form>");
   out.println("<P>GET based form:<br>"); out.print("<form
action=\"");
   out.print(response.encodeURL("SessionExample"));
out.print("\" "); out.println("method=GET>");
   out.println(rb.getString("sessions.dataname")); out.println("<input type=text
size=20 name=dataname>"); out.println("<br>");
   out.println(rb.getString("sessions.datavalue")); out.println("<input type=text size=20
name=datavalue>");
   out.println("<br>");
   out.println("<input type=submit>");
   out.println("</form>"); out.print("<p><a
href=\""); out.print(response
      .encodeURL("SessionExample?dataname=foo&datavalue=bar"));
   out.println("\" >URL encoded </a>");
out.println("</body>"); out.println("</html>");
out.println("</body>"); out.println("</html>");
}

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

final class HTMLFilter {
 public static String filter(String message) { if (message
== null) return (null);
   char content[] = new char[message.length()]; message.getChars(0,
message.length(), content, 0);
   StringBuffer result = new StringBuffer(content.length + 50);
   for (int i = 0; i < content.length; i++) {
    switch (content[i]) {
    case '<': result.append("&lt;");
```

```
      break;                    case '>':
result.append("&gt;");          break;
case '&':      result.append("&amp;");
      break;                    case '"':
result.append("&quot;");          break;
default:       result.append(content[i]);
     }
   }
   return (result.toString());
  }
}
```

Use cookie to save session data

```
import java.io.IOException; import
java.io.PrintWriter;
import java.io.UnsupportedEncodingException; import
java.net.URLEncoder;

import javax.servlet.ServletException; import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet; import
javax.servlet.http.HttpServletRequest; import
javax.servlet.http.HttpServletResponse; public class
ShoppingCartViewerCookie extends HttpServlet {

 public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException,
    IOException {
   res.setContentType("text/html");      PrintWriter out =
res.getWriter();

   String sessionid = null;
   Cookie[] cookies = req.getCookies();    if (cookies
!= null) {
    for (int i = 0; i < cookies.length; i++) {          if
(cookies[i].getName().equals("sessionid")) {
      sessionid = cookies[i].getValue();          break;
     }
    }
   }

   // If the session ID wasn't sent, generate one.
   // Then be sure to send it to the client with the response.
   if (sessionid == null) {          sessionid =
generateSessionId();
```

```
    Cookie c = new Cookie("sessionid", sessionid);
res.addCookie(c);    }

   out.println("<HEAD><TITLE>Current Shopping Cart Items</TITLE></HEAD>");
out.println("<BODY>");

   // Cart items are associated with the session ID
   String[] items = getItemsFromCart(sessionid);

   // Print the current cart items.     out.println("You currently have the following items in your
cart:<BR>");
   if (items == null) {       out.println("<B>None</B>");
   } else {       out.println("<UL>");
     for (int i = 0; i < items.length; i++) {        out.println("<LI>" + items[i]);
     }
     out.println("</UL>");     }

   // Ask if they want to add more items or check out.          out.println("<FORM
ACTION=\"/servlet/ShoppingCart\" METHOD=POST>");
   out.println("Would you like to<BR>");
   out.println("<INPUT TYPE=SUBMIT VALUE=\" Add More Items \">");     out.println("<INPUT
TYPE=SUBMIT VALUE=\" Check Out \">");     out.println("</FORM>");

   // Offer a help page.          out.println("For help, click <A
HREF=\"/servlet/Help"                                                +
"?topic=ShoppingCartViewerCookie\">here</A>");

   out.println("</BODY></HTML>");  }

  private static String generateSessionId() throws
UnsupportedEncodingException {
   String uid = new java.rmi.server.UID().toString(); // guaranteed unique          return
URLEncoder.encode(uid,"UTF-8"); // encode any special chars   }

  private static String[] getItemsFromCart(String sessionid) {          return new
String[]{"a","b"};
  }
}
```

Use URL rewrite to save session data

```
import java.io.IOException;     import
java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import                javax.servlet.ServletException;                import
javax.servlet.http.HttpServlet;                                      import
javax.servlet.http.HttpServletRequest;                               import
```

```
javax.servlet.http.HttpServletResponse; public class ShoppingCartViewerRewrite extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException,
      IOException {
   res.setContentType("text/html");  PrintWriter out = res.getWriter();

   out.println("<HEAD><TITLE>Current Shopping Cart Items</TITLE></HEAD>"); out.println("<BODY>");

   // Get the current session ID, or generate one if necessary
   String sessionid = req.getPathInfo();
   if (sessionid == null) {  sessionid = generateSessionId();  }

   // Cart items are associated with the session ID
   String[] items = getItemsFromCart(sessionid);

   // Print the current cart items.  out.println("You currently have the following items in your cart:<BR>");  if (items == null) {  out.println("<B>None</B>");
   } else {  out.println("<UL>");
     for (int i = 0; i < items.length; i++) {  out.println("<LI>" + items[i]);
     }
     out.println("</UL>");  }

   // Ask if the user wants to add more items or check out.
   // Include the session ID in the action URL.
   out.println("<FORM ACTION=\"/servlet/ShoppingCart/" + sessionid + "\" METHOD=POST>");
   out.println("Would you like to<BR>");
   out.println("<INPUT TYPE=SUBMIT VALUE=\" Add More Items \">");  out.println("<INPUT TYPE=SUBMIT VALUE=\" Check Out \">");  out.println("</FORM>");

   // Offer a help page. Include the session ID in the URL.  out.println("For help, click <A HREF=\"/servlet/Help/" + sessionid + "?topic=ShoppingCartViewerRewrite\">here</A>");

   out.println("</BODY></HTML>");  }

  private static String generateSessionId() throws UnsupportedEncodingException {
   String uid = new java.rmi.server.UID().toString(); // guaranteed unique  return URLEncoder.encode(uid, "UTF-8"); // encode any special chars  }

  private static String[] getItemsFromCart(String sessionid) {
   return new String[] { "a", "b" };
  }
```

```
}
```

Use hidden fields to save session data

```
import java.io.IOException; import java.io.PrintWriter; import
    javax.servlet.ServletException; import
    javax.servlet.http.HttpServlet; import
    javax.servlet.http.HttpServletRequest; import
    javax.servlet.http.HttpServletResponse; public class
    ShoppingCartViewerHidden extends HttpServlet {

     public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {  res.setContentType("text/html");
    PrintWriter out = res.getWriter();

      out.println("<HEAD><TITLE>Current Shopping Cart Items</TITLE></HEAD>"
    ) ;
      out.println("<BODY>");

      // Cart items are passed in as the item parameter.
      String[] items = req.getParameterValues("item");

      // Print the current cart items.  out.println("You currently have the following items
    in your cart:<BR>
    ");
      if (items == null) {  out.println("<B>None</B>");
      }  else {
       out.println("<UL>");
       for (int i = 0; i < items.length; i++) {  out.println("<LI>" + items[i]);
       }
       out.println("</UL>");  }

      // Ask if the user wants to add more items or check out.
      // Include the current items as hidden fields so they'll be passed on
    .                  out.println("<FORM  ACTION=\"/servlet/ShoppingCart\"
    METHOD=POST>");  if (items != null) {
       for (int i = 0; i < items.length; i++) {
        out.println("<INPUT TYPE=HIDDEN NAME=\"item\" VALUE=\"" +
         items[i] + "\">");
       }
      }
      out.println("Would you like to<BR>");
      out.println("<INPUT  TYPE=SUBMIT  VALUE=\"  Add  More  Items  \">");
    out.println("<INPUT  TYPE=SUBMIT  VALUE=\"  Check  Out  \">");
    out.println("</FORM>");

      out.println("</BODY></HTML>");
     }
    }
```

Program To Implement Loginform Using Sevlets

PROGRAM

filename:Usevalidation.html

```
<html>
<head>
<title>Login Form</title>
</head>
<body leftmargin=75 topmargin=75 bgcolor=white>
<form method=Get action="http:127.0.0.1:8080\validation">
UserName  <input type=text name=user>
<br><br>
Password  <input type=password name=pass>
<br><br>
<input type=submit value=submit>   
<input type=reset name=cancel>
</form>
</body>
</html>
```

filename:UserValid.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class UserValid extends HttpServlet
{
public void doGet(HttpServletRequest req,HttpServletResponse res)throws
ServletException,IOException
{
res.setContentType("text/html");
PrintWriter out=res.getWriter();
String usr=res.getParameter("user");
String pwd=res.getParameter("pass");
if(usr equals("naveen")&&pwd equals("nav")
out.println("Successfully Logged in");
else
```

```
out.println("Unsuccessful");
}
}
```

filename:web.xml

```
<web-app>
<servlet>
<servlet-name>UserValidation</servlet-name>
<servlet-class>UserValid</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>UserValidation</servlet-name>
<url-pattern>/validation/*</url-pattern>
</servlet-mapping>
</web-app>
```

5: SERVELETS & COOKIES

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: **PostParameters.htm** defines a Web page, and **PostParametersServlet.java** defines a servlet. 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. The form also includes 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/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();
}
}
```

The example contains two files: **ColorGet.htm** defines a Web page, and **ColorGetServlet.java** defines a servlet. 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.

```
<body>
<center>
<form name="Form1"
action="http://localhost:8080/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>
```

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();
```

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/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();
}
}
```

COOKIES

Cookies are small bits of textual information that a Web server sends to a browser and that the browser later returns unchanged when visiting the same Web site or domain. By letting the server read information it sent the client previously, the site can provide visitors with a number of conveniences such as presenting the site the way the visitor previously customized it or letting identifiable visitors in without their having to reenter a password.

Benefits of Cookies

There are four typical ways in which cookies can add value to your site. We summarize these benefits below:

- Identifying a user during an e-commerce session - This type of short-term tracking is so important that another API is layered on top of cookies for this purpose.
- Remembering usernames and passwords - Cookies let a user log in to a site automatically, providing a significant convenience for users of unshared computers.
- Customizing sites - Sites can use cookies to remember user preferences.
- Focusing advertising - Cookies let the site remember which topics interest certain users and show advertisements relevant to those interests.

Sending cookies to the client involves three steps:

- Creating a Cookie object - You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.
- Setting the maximum age - If you want the browser to store the cookie on disk instead of just keeping it in memory, you use setMaxAge to specify how long (in seconds) the cookie should be valid.
- Placing the Cookie into the HTTP response headers - You use response.addCookie to accomplish this. If you forget this step, no cookie is sent to the browser!

To **read the cookies** that come back from the client, you should perform the following two tasks, which are summarized below:

- Call request.getCookies. This yields an array of Cookie objects.

- Loop down the array, calling getName on each one until you find the cookie of interest. You then typically call getValue and use the value in some application-specific way.

Here are the methods that set the cookie attributes:

public void setMaxAge(int lifetime)
public int getMaxAge()

These methods tell how much time (in seconds) should elapse before the cookie expires. A negative value, which is the default, indicates that the cookie will last only for the current browsing session (i.e., until the user quits the browser) and will not be stored on disk. Specifying a value of 0 instructs the browser to delete the cookie.

public String getName()

The getName method retrieves the name of the cookie. The name and the value are the two pieces you virtually always care about. However, since the name is supplied to the Cookie constructor, there is no setName method; you cannot change the name once the cookie is created. On the other hand, getName is used on almost every cookie received by the server. Since the getCookies method of HttpServletRequest returns an array of Cookie objects, a common practice is to loop down the array, calling getName until you have a particular name, then to check the value with getValue.

public void setValue(String cookieValue)
public String getValue()
The setValue method specifies the value associated with the cookie; getValue looks it up. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases, a name is used as a boolean flag and its value is ignored (i.e., the existence of a cookie with the designated name is all that matters). However, since the cookie value is supplied to the Cookie constructor, setValue is typically reserved for cases whenyou change the values of incoming cookies and then send them back out.

Example: Write a program which stores a Cookie and the read the cookie to display the information.

```
//Part I
<html><body><center>
<form name=form1 method=get action="AddCookieServlet">
<B>Enter a Value</B>
<input type=text name=data>
<input type=submit>
</form></center></body></html>

//Part II
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet
{
public void doGet(
HttpServletRequest req, HttpServletResponse res)
throws IOException, ServletException
```

```
{
String data = req.getParametar();
Cookie c = new Cookie("My Cookie",data);
res.addCookie(c);
res.setCountentType("text/html");
PrintWriter out = res.getWriter();
out.println("My Cookie has been set to");
out.println(data);
}
}
//Part III
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookie extends HttpServlet
{
public void doGet(
HttpServletRequest req,HttpServletResponse res)
throws IOException, ServletException
{
Cookie c[] = req.getCookies();
res.setContentType("text/html");
PrintWriter out = res.getWriter();
for(int i = 0; i < c.length; i++)
{
String name = c[i].getName();
String value = c[i].getValue();
out.println(name +"\t"+ value);
}
}
}
```

HTTP SESSION

It provides a way to identify a user across more than one page request or visit to a Web site. The servlet engine uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times.

The server can maintain a session either by using cookies or by rewriting URLs.

This interface allows servlets to

- View and manipulate information about a session, such as the session identifier, creation time, or context
- Bind objects to sessions, allowing you to use an online shopping cart to hold data that persists across multiple user connections. HttpSession defines methods that store these types of data:
- Standard session properties, such as a session identifier or session context
- Data that the application provides, accessed using this interface and stored using a dictionary-like interface

An HTTP session represents the server's view of the session. The server considers a session new under any of these conditions:

- The client does not yet know about the session.
- The session has not yet begun
- The client chooses not to join the session, for example, if the server supports only cookies and the client rejects the cookies the server sends

When the session is new, the isNew() method returns true.

Method Summary

String getId() Returns a string containing the unique identifier assigned to this session.

Object getValue(String name)

Returns the object bound with the specified name in this session or null if no object of that name exists.

String[]getValueNames() Returns an array containing the names of all the objects bound to this session.

boolean isNew() Returns true if the Web server has created a session but the client has not yet joined.

void setAttribute(String name, Object value)

This method binds an object to this session, using the name specified.

Object getAttribute(String name)

This method returns the object bound with the specified name in this session,
or null if no object is bound under the name.

Example: Create a form, which accepts user information such as name and background color. Session allows storing the client information about name and background. If the user is new then display the page asking for name and background else set the background and find number of visits to the page.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class SessionServlet extends HttpServlet
{
public void service(
HttpServletResponse res,HttpServletRequest req)
throws IOException,ServletException
{
try {
res.setContentType("Text/Html");
Integer hitCount;
PrintWriter out=res.getWriter();
```

```
HttpSession s=req.getSession(true);
if(s.isNew()){
out.println("<Html>");
out.println("<Form method="+"GET"+" action=http://localhost:8080/servlet/SessionServlet>");
out.println("<b>Please select bgcolor</b>");
out.println("<input type=radio name=optColor value=red>Red");
out.println("<input type=radio name=optColor value=green>Green");
out.println("<input type=radio name=optColor value=blue>Blue");
out.println("<input type=text name=txtName>");
out.println("<br><br>");
out.println("<input type=submit value=Submit>");

out.println("</form></Html>");
}//if
else{
String name=(String)req.getParameter("txtName");
String color=(String)req.getParameter("optColor");
if(name!=null && color!=null){
out.println("Name: "+name);
hitCount=new Integer(1);
out.println("<a href=SessionServlet>SessionServlet");
s.setAttribute("txtName",name);
s.setAttribute("optColor",color);
s.setAttribute("Hit",hitCount);
}else{
hitCount=(Integer)s.getValue("Hit");
hitCount=new Integer(hitCount.intValue()+1);
s.putValue("Hit",hitCount);
out.println("<Html><body text=cyan bgcolor="+s.getAttribute("optColor")+">");
out.println("You Have Been Selected"+s.getAttribute("optColor")+"Color");
out.println("<br><br>Your Name Is"+s.getAttribute("txtName"));
out.println("<br><br>Number Of Visits==>"+hitCount);
out.println("<br><br>");
out.println("<a href=SessionServlet>SessionServlet</a>");
out.println("</body></html>");
}
}
}//try
catch(Exception e){}
} }//class
```