
UNIT 1 SERVLET PROGRAMMING

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	5
1.2 How to install Servlet Engine/Web Server	6
1.3 Your First Servlet	7
1.4 Servlet Life Cycle	9
1.5 Http Servlet Request Interface	12
1.6 Http Servlet Response Interface	13
1.7 Session Tracking	14
1.8 Database Connectivity with Servlets	19
1.9 Inter-Servlet Communication	21
1.10 Summary	25
1.11 Solutions/Answers	26
1.12 Further Readings/References	32

1.0 INTRODUCTION

We have already learnt about core Java and how to compile and learn the Java programs. In this unit we shall cover the basics of Java Servlet and different interfaces of Servlet. Java Servlets are the small, platform-independent Java programs that runs in a web server or application server and provides server-side processing such as accessing a database and e-commerce transactions. Servlets are widely used for web processing. Servlets are designed to handle HTTP requests (get, post, etc.) and are the standard Java replacement for a variety of other methods, including CGI scripts, Active Server Pages (ASPs). Servlets always run inside a Servlet Container. A Servlet Container is nothing but a web Server, which handles user requests and generates response. Servlet Container is different from Web Server because it is only meant for Servlet and not for other files (like .html etc). In this unit, we shall also see how the Servlet Container is responsible for maintaining lifecycle of the Servlet. Servlets classes should always implement the *javax.servlet.Servlet* interface. This interface contains five methods, which must be implemented. We shall learn how these methods can be used in Servlet programming and how Servlet and JDBC combination has proved a simple elegant way to connect to database.

1.1 OBJECTIVES

After going through this unit, you should be able to know:

- how to install the Servlet Engine / Web Server;
- basics of Servlet and how it is better than other server extensions;
- how Servlet engine maintains the Servlet Life Cycle;
- where do we use *HttpServletRequest* Interface and some of its basic methods;
- where do we use *HttpServletResponse* Interface and some of its basic methods;
- what is session tracking;
- different ways to achieve Session Tracking like *HttpSession* & persistent cookies, and
- different ways to achieve *InterServlet* communication.

1.2 HOW TO INSTALL SERVLET ENGINE/WEB SERVER

A Servlet is a Java class and therefore needs to be executed in a Java VM by a service we call a Servlet engine. This Servlet engine is mostly contained in Servlet Engine or may be added as a module. Some Web servers, such as Sun's Java Web Server (JWS), W3C's Jigsaw and Gefion Software's LiteWebServer (LWS) are implemented in Java and have a built-in Servlet engine. Other Web servers, such as Netscape's Enterprise Server, Microsoft's Internet Information Server (IIS) and the Apache Group's Apache, require a Servlet engine add-on module. Examples of servlet engine add-ons are Gefion Software's WAICoolRunner, IBM's WebSphere, Live Software's JRun and New Atlanta's ServletExec.

We will be using Tomcat4.0 for our Servlets. You'll need to have JDK 1.3 installed on your system in order for Tomcat 4.0 to work. If you don't already have it, you can get it from java.sun.com.

Obtaining Tomcat 4.0

Tomcat 4.0 is an open source and free Servlet Container and JSP Engine. It is developed by Apache Software Foundation's Jakarta Project and is available for download at <http://jakarta.apache.org/tomcat>, or more specifically at <http://jakarta.apache.org/site/binindex.html>.

Choose the latest Tomcat 4.0 version. Once you have downloaded Tomcat 4.0, proceed to the next step. Installing Tomcat 4.0 Unzip the file to a suitable directory. In Windows, you can unzip to C:\ which will create a directory like C:\jakarta-tomcat-4.0-b5 containing Tomcat files. Now you'll have to create two environment variables, CATALINA_HOME and JAVA_HOME. Most probably you'll have JAVA_HOME already created if you have installed Java Development Kit on your system. If not then you should create it. The values of these variables will be something like.

```
CATALINA_HOME : C:\jakarta-tomcat-4.0-b5
JAVA_HOME : C:\jdk1.3
```

To create these environment variables in Windows 2000, go to Start -> Settings -> Control Panel -> System -> Advanced -> Environment Variables -> System variables -> New. Enter the name and value for CATALINA_HOME and also for JAVA_HOME if not already there. Under Windows 95/98, you can set these variables by editing C:\autoexec.bat file. Just add the following lines and reboot your system
:SET CATALINA_HOME=C:\jakarta-tomcat-4.0-b5
:SET JAVA_HOME=C:\jdk1.3
Now copy C:\jakarta-tomcat-4.0-b5\common\lib\servlet.jar file and replace all other servlet.jar files present on your computer in the %CLASSPATH% with this file. To be on the safe side, you should rename your old servlet.jar file/s to servlet.jar_ so that you can use them again if you get any error or if any need arises. You can search for servlet.jar files on your system by going to Start -> Search -> For Files and Folders -> Search for Files and Folders named and typing servlet.jar in this field. Then every servlet.jar file you find should be replaced by C:\jakarta-tomcat-4.0-b5\common\lib\servlet.jar file. The idea is that there should be ONLY ONE VERSION OF SERVLET.JAR FILE on your system, and that one should be C:\jakarta-tomcat-4.0-b5\common\lib\servlet.jar or you might get errors when trying to run JSP pages with Tomcat.

1.3 YOUR FIRST JAVA SERVLET

Servlets are basically developed for server side applications and designed to handle http requests. The servlet-programming interface (Java Servlet API) is a standard part of the J2EE platform and has the following advantages over other common server extension mechanisms:

- They are faster than other server extensions, like, CGI scripts because they use a different process model.
- They use a standard API that is supported by many Web servers.
- Since Servlets are written in Java, Servlets are portable between servers and operating systems. They have all of the advantages of the Java language, including ease of development.

They can access the large set of APIs available for the Java platform.

Now, after installing the Tomcat Server, let us create our first Java Servlet. Create a new text file and save it as 'HelloWorld.java' in the

'C:\jakarta-tomcat-4.0-b5\webapps\saurabh\WEB-INF\classes\com\stardeveloper\servlets' folder. 'C:\jakarta-tomcat-4.0-b5\webapps\' is the folder where web applications that we create should be kept in order for Tomcat to find them. '\saurabh' is the name of our sample web application and we have created it earlier in Installing Tomcat. '\WEB-INF\classes' is the folder to keep Servlet classes. '\com\stardeveloper\servlets' is the package path to the Servlet class that we will create. Now type the following text into the 'HelloWorld.java' file we created earlier:

```
// Your First Servlet program
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HTML>");
        out.println("<HEAD><TITLE>Hello, Welcome to the World of Servlet
Programming</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<BIG>Hello World</BIG>");
        out.println("</BODY></HTML>");
    }
}
```

Our HelloWorld program is extremely simple and displays Hello World in the Browser. We override method doGet() of HttpServlet class. In the doGet() method we set the content type to 'text/html' (telling the client browser that it is an HTML document). Then get hold of PrintWriter object and using it print our own HTML content to the client. Once done we close() it.

Compiling and Running the Servlet

Now, we will compile this Servlet using the following command at the DOS prompt:

```
C:\jakarta-tomcat-4.0-b5\webapps\star\WEB-INF\classes\com\stardeveloper\servlets>javac HelloWorld.java
```

If all goes well a 'HelloWorld.class' file will be created in that folder. Now, open your browser and point to the following address:

<http://localhost:8080/star/servlet/com.stardeveloper.servlets.HelloWorld>

You should see response from the Servlet showing "Helloworld"

I have assumed here that you are running Tomcat at port 8080 (which is the default for Tomcat).

Just like an applet, a servlet does not have a main() method. Instead, certain methods of a servlet are invoked by the server in the process of handling requests. Each time the server dispatches a request to a servlet, it invokes the servlet's service() method. A generic servlet should override its service() method to handle requests as appropriate for the servlet. The service() method accepts two parameters: a request object and a response object. The request object tells the servlet about the request, while the response object is used to return a response. *Figure 1* shows how a generic servlet handles requests.

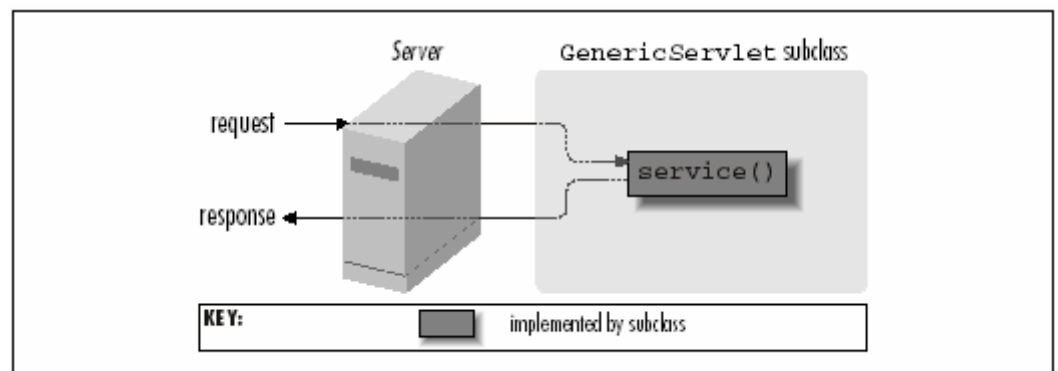


Figure 1: A generic Servlet handling a request

In contrast, an HTTP servlet usually does not override the service() method. Instead, it overrides doGet() to handle GET requests and doPost() to handle POST requests. An HTTP servlet can override either or both of these methods, depending on the type of requests it needs to handle. The service() method of HttpServlet handles the setup and dispatching to all the doXXX() methods, which is why it usually should not be overridden. *Figure 2* shows how an HTTP servlet handles GET and POST requests.

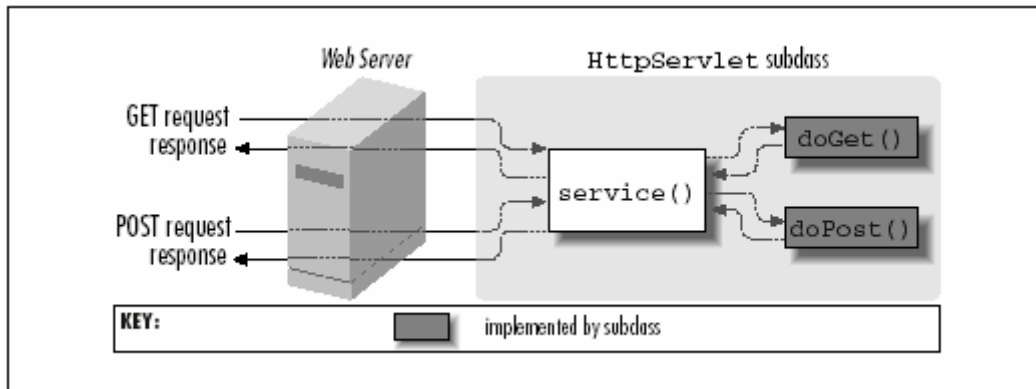


Figure 2: An Http Servlet handling get and post Requests

An HTTP servlet can override the doPut() and doDelete() methods to handle PUT and DELETE requests, respectively. However, HTTP servlets generally don't touch doHead(), doTrace(), or doOptions(). For these, the default implementations are almost always sufficient.

1.4 SERVLET LIFE CYCLE

After learning the basics of Servlet now, we shall study the life cycle of a Servlet. Servlets are normal Java classes, which are created when needed and destroyed when not needed. A Java Servlet has a lifecycle that defines how the Servlet is loaded and initialized, how it receives and responds to requests, and how it is taken out of service. In code, the Servlet lifecycle is defined by the javax.servlet.Servlet interface. Since Servlets run within a Servlet Container, creation and destruction of Servlets is the duty of Servlet Container. Implementing the init() and destroy() methods of Servlet interface allows you to be told by the Servlet Container that when it has created an instance of your Servlet and when it has destroyed that instance. An important point to remember is that your Servlet is not created and destroyed for every request it receives, rather it is created and kept in memory where requests are forwarded to it and your Servlet then generates response. We can understand Servlet Life Cycle with the help of Figure 3:

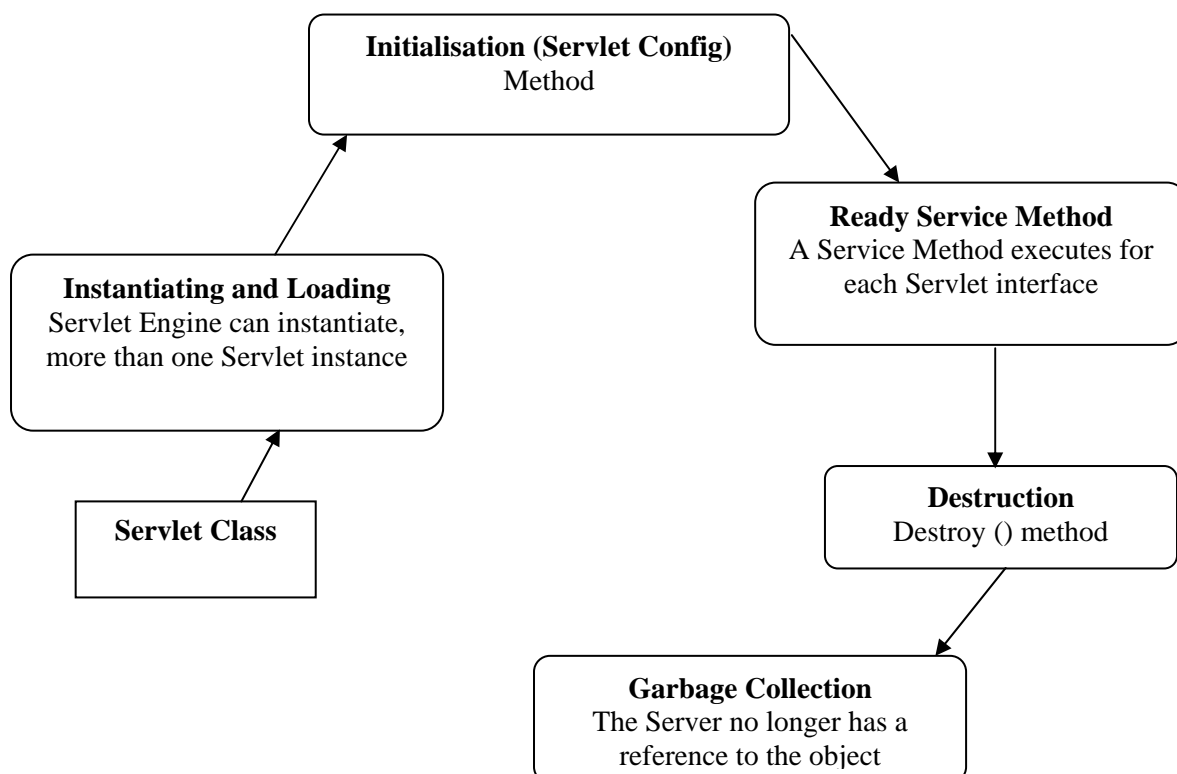


Figure 3: The Servlet Life Cycle

There are three principal stages in the life of a Java servlet, namely:

- 1) **Servlet Initialisation:** In this first stage, the servlet's constructor is called together with the servlet method `init()` this is called automatically once during the servlet's execution life cycle and can be used to place any one-off initialisation such as opening a connection to a database.

So you have created your Servlet class in above-mentioned example, by extending the `HttpServlet` class and have placed it in `/WEB-INF/classes/` directory of your application. Now, when will Servlet Container create an instance of your Servlet? There are a few situations:

- When you have specifically told the Servlet Container to preload your Servlet when the Servlet Container starts by setting the `load-on-startup` tag to a non-zero value e.g.

```
<web-app>
<servlet>
  <servlet-name>test</servlet-name>
  <servlet-class>com.stardeveloper.servlets.TestServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
</web-app>
```

So, when the Servlet Container starts it will preload your Servlet in the memory.

- If your Servlet is not already loaded, its instance will be created as soon as a request is received for it by the Servlet Container.
 - During the loading of the Servlet into the memory, Servlet Container will call your Servlet's `init()` method.
- 2) **Servlet Execution:** Once your Servlet is initialised and its `init()` method called, any request that the Servlet Container receives will be forwarded to your Servlet's `service()` method. `HttpServlet` class breaks this `service()` method into more useful `doGet()`, `doPost()`, `doDelete()`, `doOptions()`, `doPut()` and `doTrace()` methods depending on the type of HTTP request it receives. So in order to generate response you should override the `doGet()` or `doPost()` method as per your requirement.

At this moment all the requests will be forwarded to the appropriate `doGet()` or `doPost()` or whatever method as required. No new instance will be created for your Servlet.

When a Servlet request is made to the Servlet engine, the Servlet engine receives all the request parameters (such as the IP address of client), user information and user data and constructs a Servlet request object, which encapsulates all this information.

Another object, a Servlet response object is also created that contains all the information the Servlet needs to return output to the requesting client.

- 3) **Servlet Destruction:** When your application is stopped or Servlet Container shuts down, your Servlet's `destroy()` method will be called to clean up any resources allocated during initialisation and to shutdown gracefully. Hence, this acts as a good place to deallocate resources such as an open file or open database connection.

Point to remember: init() and destroy() methods will be called only once during the life time of the Servlet while service() and its broken down methods (doGet(), doPost() etc) will be called as many times as requests are received for them by the Servlet Container.

Let us understand all the phases of Servlet Life Cycle with the help of an example:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {
    private String name = "saurabh Shukla";
    public void init() throws ServletException
    {
        System.out.println("Calling the method TestServlet : init");
    }
    public void destroy()
    {
        System.out.println("Calling the method TestServlet : destroy");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        // print content
        out.println("<html><head>");
        out.println("<title>TestServlet ( by ");
        out.println( name );
        out.println(" )</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>TestServlet ( by ");
        out.println( name );
        out.println(" ) :</p>");
        out.println("</body></html>");
        out.close();
    }
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException {
        doGet(req, res);
    }
}
```

Our TestServlet is extremely simple to demonstrate the life cycle of a Servlet, which is displaying the name of the author to the user. We override four methods of HttpServlet class. init() is called by the Servlet Container only once during the initialization of the Servlet. Similarly destroy() is called once when removing the Servlet instance. We can thus put initialisation code in init() method. doGet() is called when a GET client request is received and doPost() is called when POST client request is received. In the doGet() method we set the content type to 'text/html' (telling the client browser that it is an HTML document). Then get hold of PrintWriter object and using it print our own HTML content to the client. Once done we close() it. Now, we will compile this Servlet in the same manner as we have compiled earlier:

```
C:\jakarta-tomcat-4.0-b5\webapps\star\WEB-INF\classes\
com\stardeveloper\servlets>javac TestServlet.java
```

If all goes well a 'TestServlet.class' file will be created in that folder. Now, open your browser and point to the following address:

http://localhost:8080/star/servlet/com.stardeveloper.servlets.TestServlet

You should see response from the Servlet showing “TestServlet”.

Check Your Progress 1

- 1) State True or False:
 - a) Servlet is not a Java Class. T ☐ F ☐
 - b) Tomcat 4.0 is an open source and free Servlet Container and JSP Engine. T ☐ F ☐
 - c) *init()* and *destroy()* methods will be called only once during the life time of the Servlet. T ☐ F ☐
- 2) What are the advantages of servlets over other common server extension mechanisms?
.....
.....
.....
- 3) Write a Servlet program to display “Welcome to Fifth semester of MCA”
.....
.....
.....
- 4) Explain different methods of service() method (of Servlet) to implement the request and response.
.....
.....
.....
- 5) Draw, to represent the different phases of Servlet Life Cycle.

1.5 HTTPSERVLETREQUEST INTERFACE

There are two important interfaces included in the servlet API `HttpServletRequest` and `HttpServletResponse`

The interface `HttpServletRequest` encapsulates the functionality for a request object that is passed to an HTTP Servlet. It provides access to an input stream and so allows the servlet to read data from the client. The interface also provides methods for parsing the incoming HTTP FORM data and storing the individual data values - in particular `getParameterNames()` returns the names of all the FORM's control/value pairs (request parameters). These control/value pairs are usually stored in an Enumeration object – such objects are often used for working with an ordered collection of objects.

Every call to `doGet` or `doPost` for an `HTTPServlet` receives an object that implements interface `HttpServletRequest`. The web server that executes the servlet creates an `HTTPRequest` object and passes this to servlet's service method, hence this object contains the request from the client. A variety of methods are provided to enable the servlet to process the client's request. Some of these methods are listed below:

a) getCookies

```
public Cookie[] getCookies();
```

It returns an array containing all the cookies present in this request. Cookies can be used to uniquely identify clients to servlet. If there are no cookies in the request, then an empty array is returned.

b) getQueryString

```
public String getQueryString();
```

It returns query string present in the request URL if any. A query string is defined as any information following a ? character in the URL. If there is no query string, this method returns null.

c) getSession

```
public HttpSession getSession();
public HttpSession getSession(boolean create);
```

Returns the current valid session associated with this request. If this method is called with no arguments, a session will be created for the request if there is not already a session associated with the request. If this method is called with a Boolean argument, then the session will be created only if the argument is true.

To ensure the session is properly maintained, the servlet developer must call this method before the response is committed.

If the create flag is set to false and no session is associated with this request, then this method will return null.

d) getHeader

```
public String getHeader(String name);
```

Returns the value of the requested header. The match between the given name and the request header is case-insensitive. If the header requested does not exist, this method returns null.

e) getParameter(String Name)

```
public String getParameter(String name)
```

Returns the value associated with a parameter sent to the servlet as a part of a GET or POST request. The name argument represents the parameter name.

1.6 HTTP SERVLET RESPONSE INTERFACE

The interface `HttpServletResponse` encapsulates the functionality for a response object that is returned to the client from an HTTP Servlet. It provides access to an output stream and so allows the servlet to send data to the client. A key method is `getWriter()` which obtains a reference to a `PrintWriter` object for it is this `PrintWriter` object that is used to send the text of the HTML document to the client.

Every call to `doGet` or `doPost` for an `HTTPServlet` receives an object that implements interface `HTTPServletResponse`. The web server that executes the servlet creates an `HttpServletRequest` object and passes this to servlet's service method, hence this object contains the response to the client. A variety of methods are provided to formulate the response to client. Some of these methods are listed below:

a) addCookie

```
public void addCookie(Cookie cookie);
```

It is used to add the specified cookie to the header of response. This method can be called multiple times to set more than one cookie. This method must be called before the response is committed so that the appropriate headers can be set. The cookies maximum age and whether the client allows Cookies to be saved determine whether or not Cookies will be stored on the client.

b) sendError

```
public void sendError(int statusCode) throws IOException;  
public void sendError(int statusCode, String message) throws IOException;
```

It sends an error response to the client using the specified status code. If a message is provided to this method, it is emitted as the response body, otherwise the server should return a standard message body for the error code given. This is a convenience method that immediately commits the response. No further output should be made by the servlet after calling this method.

c) getWriter

```
public PrintWriter getWriter()
```

It obtains a character-based output stream that enables text data to be sent to the client.

d) getOutputStream()

```
public ServletOutputStream getOutputStream()
```

It obtains a byte-based output stream that enables binary data to sent to the client.

e) sendRedirect

```
public void sendRedirect(String location) throws IOException;
```

It sends a temporary redirect response to the client (SC_MOVED_TEMPORARILY) using the specified location. The given location must be an absolute URL. Relative URLs are not permitted and throw an `IllegalArgumentException`.

This method must be called before the response is committed. This is a convenience method that immediately commits the response. No further output is be made by the servlet after calling this method.

1.7 SESSION TRACKING

Many web sites today provide custom web pages and / or functionality on a client-by-client basis. For example, some Web sites allow you to customize their home page to suit your needs. An excellent example of this the *Yahoo!* Web site. If you go to the site <http://my.yahoo.com/>

You can customize how the Yahoo! Site appears to you in future when you revisit the website. HTTP is a stateless protocol: it provides no way for a server to recognise that a sequence of requests is all from the same client. Privacy advocates may consider this a feature, but it causes problems because many web applications aren't stateless. The

shopping cart application is another classic example—a client can put items in his virtual cart, accumulating them until he checks out several page requests later.

Obviously the server must distinguish between clients so the company can determine the proper items and charge the proper amount for each client.

Another purpose of customizing on a client-by-client basis is marketing. Companies often track the pages you visit throughout a site so they display advertisements that are targeted to user's browsing needs.

To help the server distinguish between clients, each client must identify itself to the server. There are a number of popular techniques for distinguishing between clients. In this unit, we introduce one of the techniques called as Session Tracking.

Session tracking is wonderfully elegant. Every user of a site is associated with a `javax.servlet.http.HttpSession` object that servlets can use to store or retrieve information about that user. You can save any set of arbitrary Java objects in a session object. For example, a user's session object provides a convenient location for a servlet to store the user's shopping cart contents.

A servlet uses its request object's `getSession()` method to retrieve the current `HttpSession` object:

```
public HttpSession HttpServletRequest.getSession(boolean create)
```

This method returns the current session associated with the user making the request. If the user has no current valid session, this method creates one if `create` is `true` or returns `null` if `create` is `false`. To ensure the session is properly maintained, this method must be called at least once before any output is written to the response.

You can add data to an `HttpSession` object with the `putValue()` method:

```
public void HttpSession.putValue(String name, Object value)
```

This method binds the specified object value under the specified name. Any existing binding with the same name is replaced. To retrieve an object from a session, use

```
getValue():
public Object HttpSession.getValue(String name)
```

This methods returns the object bound under the specified name or `null` if there is no binding. You can also get the names of all of the objects bound to a session with

```
getValueNames():
public String[] HttpSession.getValueNames()
```

This method returns an array that contains the names of all objects bound to this session or an empty (zero length) array if there are no bindings. Finally, you can remove an object from a session with `removeValue()`:

```
public void HttpSession.removeValue(String name)
```

This method removes the object bound to the specified name or does nothing if there is no binding. Each of these methods can throw a `java.lang.IllegalStateException` if the session being accessed is invalid.

A Hit Count Using Session Tracking

Let us understand session tracking with a simple servlet to count the number of times a client has accessed it, as shown in example below. The servlet also displays all the bindings for the current session, just because it can.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SessionTracker extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
res.setContentType("text/html");
PrintWriter out = res.getWriter();
// Get the current session object, create one if necessary
HttpSession session = req.getSession(true);
// Increment the hit count for this page. The value is saved
// in this client's session under the name "tracker.count".
Integer count = (Integer)session.getValue("tracker.count");
if (count == null)
count = new Integer(1);
else
count = new Integer(count.intValue() + 1);
session.putValue("tracker.count", count);
out.println("<HTML><HEAD><TITLE>SessionTracker</TITLE></HEAD>");
out.println("<BODY><H1>Session Tracking Demo</H1>");
// Display the hit count for this page
out.println("You've visited this page " + count + ((count.intValue() == 1) ? " time." : "
times."));
out.println("<P>");
out.println("<H2>Here is your session data:</H2>");
String[] names = session.getValueNames();
for (int i = 0; i < names.length; i++) {
out.println(names[i] + ": " + session.getValue(names[i]) + "<BR>");
}
out.println("</BODY></HTML>");
}}
```

The output will appear as:

```
Session Tracking Demo
You've visited this page 12 times
Here is your session Data
movie.level : beginner
movie.zip : 50677
tracker.count : 12
```

This servlet first gets the HttpSession object associated with the current client. By passing true to *getSession()*, it asks for a session to be created if necessary. The servlet then gets the Integer object bound to the name “tracker.count”. If there is no such object, the servlet starts a new count. Otherwise, it replaces the Integer with a new Integer whose value has been incremented by one. Finally, the servlet displays the current count and all the current name/value pairs in the session.

Session Tracking using persistent Cookies

Another technique to perform session tracking involves **persistent cookies**. A cookie is a bit of information sent by a web server to a browser is stored it on a client machine that can later be read back from that browser. Persistent cookies offer an elegant, efficient, easy way to implement session tracking. Cookies provide as automatic introduction for each request as you could hope for. For each request, a cookie can automatically provide a client’s session ID or perhaps a list of the client’s preferences. In addition, the ability to customize cookies gives them extra power and

versatility. When a browser receives a cookie, it saves the cookie and thereafter sends the cookie back to the server each time it accesses a page on that server, subject to certain rules. Because a cookie's value can uniquely identify a client, cookies are often used for session tracking.

Note: Cookies were first introduced in Netscape Navigator. Although they were not part of the official HTTP specification, cookies quickly became a de facto standard supported in all the popular browsers including Netscape 0.94 Beta and up and Microsoft Internet Explorer 2 and up.

Problem: The biggest problem with cookies is that all the browsers don't always accept cookies. Sometimes this is because the browser doesn't support cookies. Version 2.0 of the Servlet API provides the `javax.servlet.http.Cookie` class for working with cookies. The HTTP header details for the cookies are handled by the Servlet API. You create a cookie with the `Cookie()` constructor:

```
public Cookie(String name, String value)
```

This creates a new cookie with an initial name and value..

A servlet can send a cookie to the client by passing a `Cookie` object to the `addCookie()` method of `HttpServletResponse`:

```
public void HttpServletResponse.addCookie(Cookie cookie)
```

This method adds the specified cookie to the response. Additional cookies can be added with subsequent calls to `addCookie()`. Because cookies are sent using HTTP headers, they should be added to the response before you send any content. Browsers are only required to accept 20 cookies per site, 300 total per user, and they can limit each cookie's size to 4096 bytes.

The code to set a cookie looks like this:

```
Cookie cookie = new Cookie("ID", "123");
res.addCookie(cookie);
```

A servlet retrieves cookies by calling the `getCookies()` method of `HttpServletRequest`:

```
public Cookie[] HttpServletRequest.getCookies()
```

This method returns an array of `Cookie` objects that contains all the cookies sent by the browser as part of the request or null if no cookies were sent. The code to fetch cookies looks like this:

```
Cookie[] cookies = req.getCookies();
if (cookies != null) {
    for (int i = 0; i < cookies.length; i++) {
        String name = cookies[i].getName();
        String value = cookies[i].getValue();
    }
}
```

You can set a number of attributes for a cookie in addition to its name and value. The following methods are used to set these attributes. As you will see, there is a corresponding get method for each set method. The get methods are rarely used, however, because when a cookie is sent to the server, it contains only its name, value, and version.

Here some of the methods of cookies are listed below which are used for session tracking:

public void Cookie.setMaxAge(int expiry)

Specifies the maximum age of the cookie in seconds before it expires. A negative value indicates the default, that the cookie should expire when the browser exits. A zero value tells the browser to delete the cookie immediately.

public int getMaxAge();

This method returns the maximum specified age of the cookie. If no maximum age was specified, this method returns -1.

public void Cookie.setVersion(int v)

Sets the version of a cookie. Servlets can send and receive cookies formatted to match either Netscape persistent cookies (Version 0) or the newer

public String getDomain();

Returns the domain of this cookie, or null if not defined.

public void Cookie.setDomain(String pattern)

This method sets the domain attribute of the cookie. This attribute defines which hosts the cookie should be presented to by the client. A domain begins with a dot (.foo.com) and means that hosts in that DNS zone (www.foo.com but not a.b.foo.com) should see the cookie. By default, cookies are only returned to the host which saved them.

public void Cookie.setPath(String uri)

It indicates to the user agent that this cookie should only be sent via secure channels (such as HTTPS). This should only be set when the cookie's originating server used a secure protocol to set the cookie's value.

public void Cookie.setValue(String newValue)

Assigns a new value to a cookie.

public String getValue()

This method returns the value of the cookie.

Let us understand how we use persistent cookies for the session tracking with the help of shopping cart example

```
// Session tracking using persistent cookies
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShoppingCartViewerCookie extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws
        ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        // Get the current session ID by searching the received cookies.
        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 Products/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() {
    String uid = new java.rmi.server.UID().toString(); // guaranteed unique
    return java.net.URLEncoder.encode(uid); // encode any special chars
}
private static String[] getItemsFromCart(String sessionid) {
    // Not implemented
}
}

```

This servlet first tries to fetch the client's session ID by iterating through the cookies it received as part of the request. If no cookie contains a session ID, the servlet generates a new one using `generateSessionId()` and adds a cookie containing the new session ID to the response.

1.8 DATABASE CONNECTIVITY WITH SERVLETS

Now we shall study how we can connect servlet to database. This can be done with the help of JDBC (Java Database Connectivity). Servlets, with their enduring life cycle, and JDBC, a well-defined database-independent database connectivity API, are an elegant and efficient combination and solution for webmasters who require to connect their web sites to back-end databases.

The advantage of servlets over CGI and many other technologies is that JDBC is database-independent. A servlet written to access a Sybase database can, with a two-line modification or a change in a properties file, begin accessing an Oracle database. One common place for servlets, especially servlets that access a database, is in what's called the middle tier. A middle tier is something that helps connect one endpoint to another (a servlet or applet to a database, for example) and along the way adds a little something of its own. The middle tier is used between a client and our ultimate data source (commonly referred to as middleware) to include the business logic. Let us understand the database connectivity of servlet with table with the help of an example. The following example shows a very simple servlet that uses the MS-Access JDBC driver to perform a simple query, printing names and phone numbers for all employees listed in a database table. We assume that the database contains a table named CUSTOMER, with at least two fields, NAME and ADDRESS.

```
/* Example to demonstrate how JDBC is used with Servlet to connect to a customer  
table and to display its records*/  
import java.io.*;  
import java.sql.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
public class DBPhoneLookup extends HttpServlet {  
    public void doGet(HttpServletRequest req, HttpServletResponse res)  
        throws ServletException, IOException {  
        Connection con = null;  
        Statement stmt = null;  
        ResultSet rs = null;  
        res.setContentType("text/html");  
        PrintWriter out = res.getWriter();  
        try {  
            // Load (and therefore register) the Oracle Driver  
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
            // Get a Connection to the database  
            Connection con = DriverManager.getConnection("jdbc:odbc:Access");  
            // Create a Statement object  
            stmt = con.createStatement();  
            // Execute an SQL query, get a ResultSet  
            rs = stmt.executeQuery("SELECT NAME, ADDRESS FROM CUSTOMER");  
            // Display the result set as a list  
            out.println("<HTML><HEAD><TITLE>Phonebook</TITLE></HEAD>");  
            out.println("<BODY>");  
            out.println("<UL>");  
            while(rs.next()) {  
                out.println("<LI>" + rs.getString("name") + " " + rs.getString("address"));  
            }  
            out.println("</UL>");  
            out.println("</BODY></HTML>");  
        }  
        catch(ClassNotFoundException e) {  
            out.println("Couldn't load database driver: " + e.getMessage());  
        }  
        catch(SQLException e) {  
            out.println("SQLException caught: " + e.getMessage());  
        }  
        finally {  
            // Always close the database connection.  
            try {  
                if (con != null) con.close();  
            }  
        }  
    }  
}
```



```
catch (SQLException e1) { }
}
}
}
```

In the above example a simple servlet program is written to connect to the database, and which executes a query that retrieves the names and phone numbers of everyone in the employees table, and display the list to the user.

1.9 INTER-SERVLET COMMUNICATION

Now, we shall study why we need InterServlet communication. Servlets which are running together in the same server have several ways to communicate with each other. There are three major reasons to use interservlet communication:

a) Direct servlet manipulation / handling

A servlet can gain access to the other currently loaded servlets and perform some task on each. The servlet could, for example, periodically ask every servlet to write its state to disk to protect against server crashes.

Direct servlet manipulation / handling involves one servlet accessing the loaded servlets on its server and optionally performing some task on one or more of them. A servlet obtains information about other servlets through the ServletContext object.

Use `getServlet()` to get a particular servlet:

`public Servlet ServletContext.getServlet(String name)` throws `ServletException`

This method returns the servlet of the given name, or null if the servlet is not found.

The specified name can be the servlet's registered name (such as "file") or its class name (such as "com.sun.server.webserver.FileServlet"). The server maintains one servlet instance per name, so `getServlet("file")` returns a different servlet instance than `getServlet("com.sun.server.webserver .FileServlet")`.

You can also get all of the servlets using `getServlets()`:

`public Enumeration ServletContext.getServlets()`

This method returns an Enumeration of the servlet objects loaded in the current ServletContext. Generally there's one servlet context per server, but for security or convenience, a server may decide to partition its servlets into separate contexts. The enumeration always includes the calling servlet itself.

Let us take an example to understand how we can view the currently loaded servlets.

```
//Example Checking out the currently loaded servlets
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Loaded extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
res.setContentType("text/plain");
PrintWriter out = res.getWriter();
ServletContext context = getServletContext();
Enumeration names = context.getServletNames();
while (names.hasMoreElements()) {
String name = (String)names.nextElement();
Servlet servlet = context.getServlet(name);
```

```
out.println("Servlet name: " + name);
out.println("Servlet class: " + servlet.getClass().getName());
out.println("Servlet info: " + servlet.getServletInfo());
out.println();
}
}
}
```

In the above example, it retrieves its `ServletContext` to access the other servlets loaded in the server. Then it calls the context's `getServletNames()` method. This returns an enumeration of `String` objects that the servlet iterates over in a while loop. For each name, it retrieves the corresponding servlet object with a call to the context's `getServlet()` method. Then it prints three items of information about the servlet: its name, its class name, and its `getServletInfo()` text.

b) Servlet reuse

Another use for interservlet communication is to allow one servlet to reuse the abilities (the public methods) of another servlet. The major challenge with servlet reuse is for the “user” servlet to obtain the proper instance of “usee” servlet when the usee servlet has not yet been loaded into the server. For example a servlet named as `ChatServlet` was written as a server for chat applets, but it could be reused (unchanged) by another servlet that needed to support an HTML-based chat interface. Servlet can be done with the user servlet to ask the server to load the usee servlet, then call `getServlet()` to get a reference to it. Unfortunately, the Servlet API distinctly lacks any methods whereby a servlet can control the servlet life cycle, for itself or for other servlets. This is considered a security risk and is officially “left for future consideration.” Fortunately, there’s a backdoor we can use today. A servlet can open an HTTP connection to the server in which it’s running, ask for the unloaded servlet, and effectively force the server to load the servlet to handle the request. Then a call to `getServlet()` gets the proper instance.

c) Servlet collaboration

Sometimes servlets have to cooperate, usually by sharing some information. We call this type of communication as servlet collaboration. Collaborating servlets can pass the shared information directly from one servlet to another through method invocations. This approach requires each servlet to know the other servlets with which it is collaborating. The most common situation involves two or more servlets sharing state information. For example, a set of servlets managing an online store could share the store’s product inventory count. Session tracking can be considered as a special case of servlet collaboration.

Collaboration using system property list:

One simple way for servlets to share information is by using Java’s system-wide Properties list, found in the `java.lang.System` class. This Properties list holds the standard system properties, such as `java.version` and `path.separator`, but it can also hold application-specific properties. Servlets can use the properties list to hold the information they need to share. A servlet can add (or change) a property by calling: `System.getProperties().put(“key”, “value”);` That servlet, or another servlet running in the same JVM, can later get the value of the property by calling: `String value = System.getProperty(“key”);` The property can be removed by calling: `System.getProperties().remove(“key”);`

The Properties class is intended to be String based, meaning that each key and value is supposed to be a String.

Collaboration through a shared object :

Another way for servlets to share information is through a shared object. A shared object can hold the pool of shared information and make it available to each servlet as needed. In a sense, the system Properties list is a special case example of a shared object. By generalising the technique into sharing any sort of object, however, a servlet is able to use whatever shared object best solves its particular problem.

Often the shared object incorporates a fair amount of business logic or rules for manipulating the object's data. This business logic protects the shared object's actual data by making it available only through well-defined methods.

There's one thing to watch out for when collaborating through a shared object is the garbage collector. It can reclaim the shared object if at any time the object isn't referenced by a loaded servlet. To keep the garbage collector at bay, every servlet using a shared object should save a reference to the object.

Check Your Progress 2

- 1) What are the main functions of HttpServletRequest Interface? Explain the methods which are used to obtain cookies and querystring from the request object.
.....
.....
.....
- 2) What are the main functions of HttpServletResponse Interface? Explain the methods which are used to add cookies to response and send an error response.
.....
.....
.....
- 3) Explain the various purposes for which we use Session tracking. Also, Explain in brief the two ways to handle Session Tracking in Servlets.
.....
.....
.....
- 4) Assume there is a table named Product in MS-access with fields (Product_id, Prod_name, Price, Qty). Write a code for Servlet which will display all the fields of product table in Tabular manner.
.....
.....
.....
- 5) What are the two ways used for Servlet collaboration
.....
.....
.....
- 6) How do I call a servlet with parameters in the URL?
.....
.....
.....

- 7) How do I deserialize an httpsession?
.....
.....
.....
- 8) How do I restrict access to servlets and JSPs?
.....
.....
.....
- 9) What is the difference between JSP and servlets ?
.....
.....
.....
- 10) Difference between GET and POST .
.....
.....
.....
- 11) Can we use the constructor, instead of init(), to initialize servlet?
.....
.....
.....
- 12) What is servlet context ?
.....
.....
.....
- 13) What are two different types of servlets ? Explain the differences between these two.
.....
.....
.....
- 14) What is the difference between ServletContext and ServletConfig?
.....
.....
.....
- 15) What are the differences between a session and a cookie?
.....
.....
.....
- 16) How will you delete a cookie?
.....
.....
.....

- 17) What is the difference between Context init parameter and Servlet init parameter?

- 18) What are the different types of ServletEngines?

- 19) What is Servlet chaining?

1.10 SUMMARY

Java servlets are the small, platform-independent Java programs that run in a web server or application server and provide server-side processing such as accessing a database and e-commerce transactions. Servlets are widely used for web processing. Servlets dynamically extend the functionality of a web server. A servlet engine can only execute servlet which is contained in the web-servers like, JWS or JIGSAW.

Servlets are basically developed for the server side applications and designed to handle http requests. They are better than other common server extensions like, CGI as they are faster, have all the advantages of Java language and supported by many of the browsers.

A Java Servlet has a lifecycle that defines how the servlet is loaded and initialised, how it receives and responds to requests, and how it is taken out of service. Servlets run within a Servlet Container, creation and destruction of servlets is the duty of Servlet Container. There are three principal stages in the life of a Java Servlet, namely: Servlet Initialisation, Servlet Execution and Servlet Destruction. In this first stage, the servlet's constructor is called together with the servlet method `init()` - this is called automatically once during the servlet's execution life cycle. Once your servlet is initialised, any request that the Servlet Container receives will be forwarded to your Servlet's `service()` method. `HttpServlet` class breaks this `service()` method into more useful `doGet()`, `doPost()`, `doDelete()`, `doOptions()`, `doPut()` and `doTrace()` methods depending on the type of HTTP request it receives. When the application is stopped or Servlet Container shuts down, your Servlet's `destroy()` method will be called to clean up any resources allocated during initialisation and to shutdown gracefully.

There are two important interfaces included in the servlet API. They are `HttpServletRequest` and `HttpServletResponse`. `HttpServletRequest` encapsulates the functionality for a request object that is passed to an HTTP Servlet. It provides access to an input stream and so allows the servlet to read data from the client and it has methods like, `getCookies()`, `getQueryString()` & `getSession` etc. `HttpServletResponse` encapsulates the functionality for a response object that is returned to the client from an HTTP Servlet. It provides access to an output stream and so allows the servlet to send data to the client and it has methods like, `addCookie()`, `sendError()` and `getWriter()` etc.

Session tracking is another important feature of servlet. Every user of a site is associated with a `javax.servlet.http.HttpSession` object that servlets can use to store or retrieve information about that user. A servlet uses its request object's `getSession()`

method to retrieve the current HttpSession object and can add data to an HttpSession object with the putValue() method. Another technique to perform session tracking involves persistent cookies. A cookie is a bit of information sent by a web server to a browser and stores it on a client machine that can later be read back from that browser. For each request, a cookie can automatically provide a client's session ID or perhaps a list of the client's preferences.

Servlet along JDBC API can be used to connect to the different databases like, Sybase, Oracle etc. A servlet written to access a Sybase database can, with a two-line modification or a change in a properties file, begin accessing an Oracle database. It again uses the objects and methods of java.sql.* package.

Servlets, which are running together in the same server, have several ways to communicate with each other. There are three reasons to use InterServlet communication. First is Direct Servlet manipulation handling in which servlet can gain access to the other currently loaded servlets and perform some task on each. Second is Servlet Reuse that allows one servlet to reuse the abilities (the public methods) of another servlet. Third is Servlet collaboration that allows servlets to cooperate, usually by sharing some information.

1.11 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) True/ False
 - a) False b) True c) True
- 2) A servlet is a Java class and therefore needs to be executed in a Java VM by a service we call a Servlet engine. Servlets dynamically extend the functionality of a web server and basically developed for the server side applications. Servlets have the following advantages over other common server extension mechanisms:
 - They are faster than other server extensions like, CGI scripts because they use a different process model.
 - They use a standard API that is supported by many web servers.
 - It executes within the address space of a web server.
 - Since servlets are written in Java, servlets are portable between servers and operating systems. They have all of the advantages of the Java language, including ease of development and platform independence.
 - It does not require creation of a separate process for each client request.
- 3) Code to display **“Welcome to Fifth semester of MCA”**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HTML>");
    }
}
```

```

out.println("<HEAD><TITLE>5th Semester MCA </TITLE></HEAD>");
out.println("<BODY>");
out.println("<B>Welcome to Fifth Semester of MCA</B>");
out.println("</BODY></HTML>");
}
}

```

- 4) Servlets are the Java classes which are created when needed and destroyed when not needed. Since servlets run within a Servlet Container, creation and destruction of servlets is the duty of Servlet Container. There are three principal stages in the life of a Java Servlet Life Cycle, namely:

- i) **Servlet Initialisation:** In this first stage, the servlet's constructor is called together with the servlet method `init()` - this is called automatically once during the servlet's execution life cycle and can be used to place any one-off initialisation such as opening a connection to a database.
- ii) **Servlet Execution:** Once your servlet is initialized and its `init()` method called, any request that the Servlet Container receives will be forwarded to your Servlet's `service()` method. `HttpServlet` class breaks this `service()` method into more useful `doGet()`, `doPost()`, `doDelete()`, `doOptions()`, `doPut()` and `doTrace()` methods depending on the type of HTTP request it receives. So in order to generate response, the `doGet()` or `doPost()` method should be overridden as per the requirement.

When a servlet request is made to the Servlet engine, the Servlet engine receives all the request parameters (such as the IP address of client), user information and user data and constructs a Servlet request object, which encapsulates all this information.

- iii) **Servlet Destruction:** When the application is stopped or Servlet Container shuts down, Servlet's `destroy()` method will be called to clean up any resources allocated during initialisation and to shutdown gracefully. Hence, it acts as a place to deallocate resources such as an open file or open database connection.

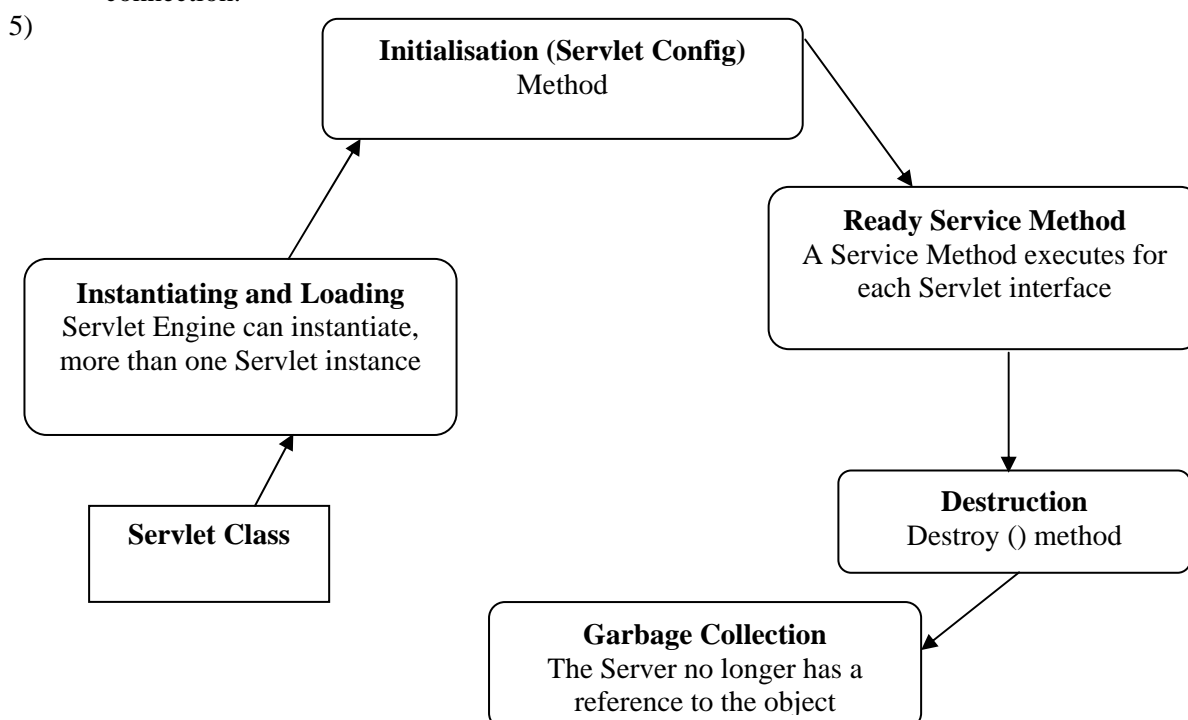


Figure 4: The servlet life cycle

Check Your Progress 2

1. Main functions of HttpServletRequest Interface are the following:

- The interface HttpServletRequest encapsulates the functionality for a request object that is passed to an HTTP Servlet.
- It provides access to an input stream and so allows the servlet to read data from the client.
- It provides methods for parsing the incoming HTTP FORM data and storing the individual data values - in particular `getParameterNames()` returns the names of all the FORM's control/value pairs
- It contains the request from the client

getCookies is the method which is used to obtain cookies from the request object and following is the its syntax:

```
public Cookie[] getCookies();
```

It returns an array containing all the cookies present in this request. Cookies can be used to uniquely identify clients to servlet. If there are no cookies in the request, then an empty array is returned.

getQueryString is the method used to obtain the querystring from the request object. The syntax used is

```
public String getQueryString();
```

It returns query string present in the request URL if any. A query string is defined as any information following a ? character in the URL. If there is no query string, this Method returns null.

2) Main functions of HttpServletResponse Interface are:

- It encapsulates the functionality for a response object that is returned to the client from an HTTP Servlet.
- It provides access to an output stream and so allows the servlet to send data to the client.
- It uses `getWriter()` method to obtain a reference to a `PrintWriter` object. and `PrintWriter` object is used to send the text of the HTML document to the client.
- The web server that executes the servlet creates an `HttpRequest` object and passes this to servlet's service method.
- It contains the response to the client.

addCookie is the method which is used to add cookies to the response object. Syntax is

```
public void addCookie(Cookie cookie);
```

It is used to add the specified cookie to the header of response. This method can be called multiple times to set more than one cookie. This method must be called before the response is committed so that the appropriate headers can be set.

sendError is the method used to send an error response. Syntax is :

```
public void sendError(int statusCode) throws IOException;
```


public void sendError(int statusCode, String message) throws IOException;

It sends an error response to the client using the specified status code. If a message is provided to this method, it is emitted as the response body, otherwise the server should return a standard message body for the error code given.

3) Various purposes for the session tracking are:

- To know the client's preferences
- To distinguish between different clients
- To customize the website like shopping cart as per the user requirement or preference.

Session Tracking using persistent Cookies: A cookie is a bit of information sent by a web server to a browser and stores it on a client machine that can later be read back from that browser. Persistent cookies offer an elegant, efficient, easy way to implement session tracking. For each request, a cookie can automatically provide a client's session ID or perhaps a list of the client's preferences. In addition, the ability to customize cookies gives them extra power and versatility. When a browser receives a cookie, it saves the cookie and thereafter sends the cookie back to the server each time it accesses a page on that server, subject to certain rules. Because a cookie's value can uniquely identify a client, cookies are used for session tracking.

Cookie can be created with the Cookie() constructor:

```
public Cookie(String name, String value)
```

A servlet can send a cookie to the client by passing a Cookie object to the

addCookie() method of HttpServletResponse:

```
public void HttpServletResponse.addCookie(Cookie cookie)
```

4) Code for servlet to display all the fields of PRODUCT Table:

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DBPhoneLookup extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    try {
        // Load (and therefore register) the Oracle Driver
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        // Get a Connection to the database
        Connection con = DriverManager.getConnection ("jdbc:odbc:Access");
        // Create a Statement object
        stmt = con.createStatement();
        // Execute an SQL query, get a ResultSet
        rs = stmt.executeQuery("SELECT Product_id, Prod_name, Price, Qty FROM
PRODUCT");
        // Display the result set as a list
```

```
out.println("<HTML><HEAD><TITLE>Phonebook</TITLE></HEAD>");
out.println("<BODY>");
out.println("<Table>");
while(rs.next()) {
out.println("<TR>");
out.println("<TD>" + rs.getString("Product_id") + "</TD><TD>" +
rs.getString("Prod_name") + "</TD><TD>" + rs.getString("Price") +
"</TD><TD>" + rs.getString("Qty") + "</TD>");
out.println("</TR>");

}
out.println("</Table>");
out.println("</BODY></HTML>");
}
catch(ClassNotFoundException e) {
out.println("Couldn't load database driver: " + e.getMessage());
}
catch(SQLException e) {
out.println("SQLException caught: " + e.getMessage());
}
finally {
// Always close the database connection.
try {
if (con != null) con.close();
}
catch (SQLException e1) { }
}}}

```

5) The two ways used for servlet collaboration are the following:

a) Collaboration using system property list: One simple way for servlets to share information is by using Java's system-wide Properties list, found in the `java.lang.System` class. This Properties list holds the standard system properties, such as `java.version` and `path.separator`, but it can also hold application-specific properties. Servlets can use the properties list to hold the information they need to share. A servlet can add(or change) a property by calling:
`System.getProperties().put("key", "value");`

b) Collaboration through a shared object : Another way for servlets to share information is through a shared object. A shared object can hold the pool of shared information and make it available to each servlet as needed, the system Properties list is a special case example of a shared object. The shared object may incorporate the business logic or rules for manipulating the object's data.

6) The usual format of a servlet parameter is a name=value pair that comes after a question-mark (?) at the end of the URL. To access these parameters, call the `getParameter()` method on the `HttpServletRequest` object, then write code to test the strings. For example, if your URL parameters are "func=topic," where your URL appears as:

`http://www..ignou.ac.in/myservlet?func=topic` then you could parse the parameter as follows, where "req" is the `HttpServletRequest` object:

```
String func = req.getParameter("func");
if (func.equalsIgnoreCase("topic"))
{ . . . Write an appropriate code }
```

- 7) To deserialise an httpsession, construct a utility class that uses the current thread's contextclassloader to load the user defined objects within the application context. Then add this utility class to the system CLASSPATH.
- 8) The Java Servlet API Specification v2.3 allows you to declaratively restrict access to specific servlets and JSPs using the Web Application deployment descriptor. You can also specify roles for EJBs and Web applications through the Administration Console.
- 9) JSP is used mainly for presentation only. A JSP can only be HttpServlet that means the only supported protocol in JSP is HTTP. But a servlet can support any protocol like, HTTP, FTP, SMTP etc.
- 10) In GET entire form submission can be encapsulated in one URL, like a hyperlink. Query length is limited to 255 characters, not secure, faster, quick and easy. The data is submitted as part of URL. Data will be visible to user.

In POST data is submitted inside body of the HTTP request. The data is not visible on the URL and it is more secure.

- 11) Yes. But you will not get the servlet specific things from constructor. The original reason for init() was that ancient versions of Java couldn't dynamically invoke constructors with arguments, so there was no way to give the constructor a ServletConfig. That no longer applies, but servlet containers still will only call your no-arg constructor. So you won't have access to a ServletConfig or ServletContext.
- 12) The servlet context is an object that contains information about the web application and container. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.
- 13) GenericServlet and HttpServlet. HttpServlet is used to implement HTTP protocol, whereas Generic servlet can implement any protocol.

By extending GenericServlet we can write a servlet that supports our own custom protocol or any other protocol.

- 14) The ServletConfig gives the information about the servlet initialization parameters. The servlet engine implements the ServletConfig interface in order to pass configuration information to a servlet. The server passes an object that implements the ServletConfig interface to the servlet's init() method. The ServletContext gives information about the container. The ServletContext interface provides information to servlets regarding the environment in which they are running. It also provides standard way for servlets to write events to a log file.
- 15) Session is stored in server but cookie stored in client. Session should work regardless of the settings on the client browser. There is no limit on the amount of data that can be stored on session. But it is limited in cookie. Session can store objects and cookies can store only strings. Cookies are faster than session.
- 16) `Cookie c = new Cookie ("name", null);`

```
c.setMaxAge(0);
```

```
response.addCookie(killCookie);
```

- 17) Servlet init parameters are for a single servlet only. No body outside that servlet can access that. It is declared inside the <servlet> tag inside Deployment Descriptor, whereas context init parameter is for the entire web application. Any servlet or JSP in that web application can access context init parameter. Context parameters are declared in a tag <context-param> directly inside the <web-app> tag. The methods for accessing context init parameter is `getServletContext ().getInitParamter ("name")` whereas method for accessing servlet init parameter is `getServletConfig ().getInitParamter ("name")`;
- 18) The different types of ServletEngines available are: Standalone ServletEngine: This is a server that includes built-in support for servlets. Add-on ServletEngine: It is a plug-in to an existing server. It adds servlet support to a server that was not originally designed with servlets in mind.
- 19) Servlet chaining is a technique in which two or more servlets can cooperate in servicing a single request. In servlet chaining, one servlet's output is the input of the next servlet. This process continues until the last servlet is reached. Its output is then sent back to the client. We are achieving Servlet Chaining with the help of RequestDispatcher.

1.12 FURTHER READINGS/REFERENCES

- Dietel and Dietel, *Internet & World wide Web Programming*, Prentice Hall
- Potts, Stephen & Pestrikov, Alex, *Java 2 Unleashed*, Sams
- Keogh, James, *J2EE: The Complete Reference*, McGraw-Hill
- Inderjeet Singh & Bet Stearns, *Designing Enterprise Application with j2EE*, Second Edition platform, Addison Wesley
- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions*, New Riders Publishing.
- Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible*, Hungry Minds
- Marty Hall, *Core Servlets and JavaServer Pages (JSP)*, Prentice Hall.

Reference websites:

- www.apl.jhu.edu
- www.java.sun.com
- www.novocode.com
- www.javaskyline.com
- www.stardeveloper.com

UNIT 2 JAVA DATABASE CONNECTIVITY

Structure	Page Nos.
2.0 Introduction	33
2.1 Objectives	33
2.2 JDBC Vs ODBC	33
2.3 How Does JDBC Work?	34
2.4 JDBC API	35
2.5 Types of JDBC Drivers	36
2.6 Steps to connect to a Database	39
2.7 Using JDBC to Query a Database	41
2.8 Using JDBC to Modify a Database	43
2.9 Summary	45
2.10 Solutions/Answers	46
2.11 Further Readings/References	51

2.0 INTRODUCTION

In previous blocks of this course we have learnt the basics of Java Servlets. In this UNIT we shall cover the database connectivity with Java using JDBC. JDBC (the Java Database Connectivity) is a standard SQL database access interface that provides uniform access to a wide range of relational databases like MS-Access, Oracle or Sybase. It also provides a common base on which higher-level tools and interfaces can be built. It includes ODBC Bridge. The Bridge is a library that implements JDBC in terms of the ODBC standard C API. In this unit we will first go through the different types of JDBC drivers and then JDBC API and its different objects like connection, statement and ResultSet. We shall also learn how to query and update the database using JDBC API.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the basics of JDBC and ODBC;
 - understand the architecture of JDBC API and its objects;.
 - understand the different types of statement objects and their usage;
 - understand the different Types of JDBC drivers & their advantages and disadvantages;
 - steps to connect a database;
 - how to use JDBC to query a database and,
 - understand, how to use JDBC to modify the database.
-

2.2 JDBC Vs. ODBC

Now, we shall study the comparison between JDBC and ODBC. The most widely used interface to access relational databases today is Microsoft's ODBC API. ODBC stands for Open Database Connectivity, a standard database access method developed by the SQL Access group in 1992. Through ODBC it is possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a *database*

driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.

Microsoft ODBC API offers connectivity to almost all databases on almost all platforms and is the most widely used programming interface for accessing relational databases. But ODBC cannot be used directly with Java Programs due to various reasons described below.

- 1) ODBC cannot be used directly with Java because, it uses a C interface. This will have drawbacks in the security, implementation, and robustness.
- 2) ODBC makes use of Pointers, which have been removed from Java.
- 3) ODBC mixes simple and advanced features together and has complex structure.

Hence, JDBC came into existence. If you had done Database Programming with Visual Basic, then you will be familiar with ODBC. You can connect a VB Application to MS-Access Database or an Oracle Table directly via ODBC. Since Java is a product of Sun Microsystems, you have to make use of JDBC with ODBC in order to develop Java Database Applications.

JDBC is an API (Application Programming Interface) which consists of a set of Java classes, interfaces and exceptions. With the help of JDBC programming interface, Java programmers can request a connection with a database, then send query statements using SQL and receive the results for processing.

According to Sun, specialised JDBC drivers are available for all major databases — including relational databases from Oracle Corp., IBM, Microsoft Corp., Informix Corp. and Sybase Inc. — as well as for any data source that uses Microsoft's Open Database Connectivity system.

The combination of Java with JDBC is very useful because it lets the programmer run his/ her program on different platforms. Some of the advantages of using Java with JDBC are:

- Easy and economical
- Continued usage of already installed databases
- Development time is short
- Installation and version control simplified.

2.3 HOW DOES JDBC WORK?

Simply, JDBC makes it possible to do the following things within a Java application:

- Establish a connection with a data source
- Send SQL queries and update statements to the data source
- Process the results at the front-end

Figure 1 shows the components of the JDBC model

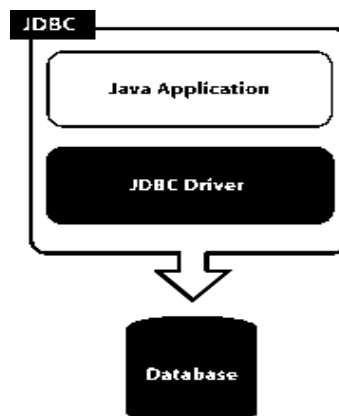


Figure 1: Components of java database connectivity

The Java application calls JDBC classes and interfaces to submit SQL statements and retrieve results.

2.4 JDBC API

Now, we will learn about the JDBC API. The JDBC API is implemented through the JDBC driver. The JDBC Driver is a set of classes that implement the JDBC interfaces to process JDBC calls and return result sets to a Java application. The database (or data store) stores the data retrieved by the application using the JDBC Driver.

The API interface is made up of 4 main interfaces:

- `java.sql.DriverManager`
- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.ResultSet`

In addition to these, the following support interfaces are also available to the developer:

- `java.sql.CallableStatement`
- `java.sql.DatabaseMetaData`
- `java.sql.Driver`
- `java.sql.PreparedStatement`
- `java.sql.ResultSetMetaData`
- `java.sql.DriverPropertyInfo`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `java.sql.Types`
- `java.sql.Numeric`

The main objects of the JDBC API include:

- A **DataSource** object is used to establish connections. Although the Driver Manager can also be used to establish a connection, connecting through a DataSource object is the preferred method.
- A **Connection** object controls the connection to the database. An application can alter the behavior of a connection by invoking the methods associated with this object. An application uses the connection object to create statements.
- **Statement** objects are used for executing SQL queries.

Different types of JDBC SQL Statements

- java.sql.Statement** : Top most interface which provides basic methods useful for executing SELECT, INSERT, UPDATE and DELETE SQL statements.
- java.sql.PreparedStatement** : An enhanced version of `java.sql.Statement` which allows precompiled queries with parameters. A *PreparedStatement* object is used when an application plans to specify parameters to your SQL queries. The statement can be executed multiple times with different parameter values specified for each execution.

- c) **java.sql.CallableStatement** : It allows you to execute stored procedures within a RDBMS which supports stored procedures. The Callable Statement has methods for retrieving the return values of the stored procedure.

A **ResultSet** Object act like a workspace to store the results of query. A ResultSet is returned to an application when a SQL query is executed by a statement object. The ResultSet object provides several methods for iterating through the results of the query.

2.5 TYPES OF JDBC DRIVERS

We have learnt about JDBC API. Now we will study different types of drivers available in java of which some are pure and some are impure.

To connect with individual databases, JDBC requires drivers for each database. There are four types of drivers available in Java for database connectivity. Types 3 and 4 are pure drivers whereas Types 1 and 2 are impure drivers. Types 1 and 2 are intended for programmers writing applications, while Types 3 and 4 are typically used by vendors of middleware or databases.

Type 1: JDBC-ODBC Bridge

They are JDBC-ODBC Bridge drivers. They delegate the work of data access to ODBC API. ODBC is widely used by developers to connect to databases in a non-Java environment. This kind of driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important.

Note: Some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver.

Advantages: It acts as a good approach for learning JDBC. It may be useful for companies that already have ODBC drivers installed on each client machine — typically the case for Windows-based machines running productivity applications. It may be the only way to gain access to some low-end desktop databases.

Disadvantage: It is not suitable for large-scale applications. They are the slowest of all. The performance of system may suffer because there is some overhead associated with the translation work to go from JDBC to ODBC. It doesn't support all the features of Java. User is limited by the functionality of the underlying ODBC driver, as it is product of different vendor.

Type 2: Native-API partly Java technology-enabled driver

They mainly use native API for data access and provide Java wrapper classes to be able to be invoked using JDBC drivers. It converts the calls that a developer writes to the JDBC application programming interface into calls that connect to the client machine's application programming interface for a specific database, such as IBM, Informix, Oracle or Sybase, like, the bridge driver, this style of driver requires that some binary code be loaded on each client machine.

Advantage: It has a better performance than that of Type 1, in part because the Type 2 driver contains *compiled code* that's optimised for the back-end database server's operating system.

Disadvantage: For this, User needs to make sure the JDBC driver of the database vendor is loaded onto each client machine. Must have compiled code for every operating system that the application will run on. Best use is for controlled environments, such as an intranet.

Type 3: A net-protocol fully Java technology-enabled driver

They are written in 100% Java and use vendor independent Net-protocol to access a vendor independent remote listener. This listener in turn maps the vendor independent calls to vendor dependent ones. This extra step adds complexity and decreases the data access efficiency. It is pure Java driver for database middleware, which translates JDBC API calls into a DBMS-independent net protocol, which is then translated, to a DBMS protocol by a server. It translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software. This net server middleware is able to connect all of its Java technology-based clients to many different databases. In general, this is the most flexible JDBC API alternative.

Advantage: It has better performance than Types 1 and 2. It can be used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them. Since, it is server-based, so there is no requirement for JDBC driver code on client machine. For performance reasons, the back-end server component is optimized for the operating system that the database is running on.

Disadvantage: It needs some database-specific code on the middleware server. If the middleware is to run on different platforms, then Type 4 driver might be more effective.

Type 4: A native-protocol fully Java technology-enabled driver

It is direct-to-database pure Java driver. It converts JDBC technology calls into the network protocol used by different DBMSs directly. Basically it converts JDBC calls into packets that are sent over the network in the proprietary format used by the specific database. Allows a direct call from the client machine to the database.

Advantage: It again has better performance than Types 1 and 2 and there is no need to install special software on client or server. It can be downloaded dynamically.

Disadvantage: It is not optimized for server operating system, so the driver can't take advantage of operating system features. (The driver is optimized for the specific database and can take advantage of the database vendor's functionality.). For this, user needs a different driver for each different database.

The following figure shows a side-by-side comparison of the implementation of each JDBC driver type. All four implementations show a Java application or applet using the JDBC API to communicate through the JDBC Driver Manager with a specific JDBC driver.

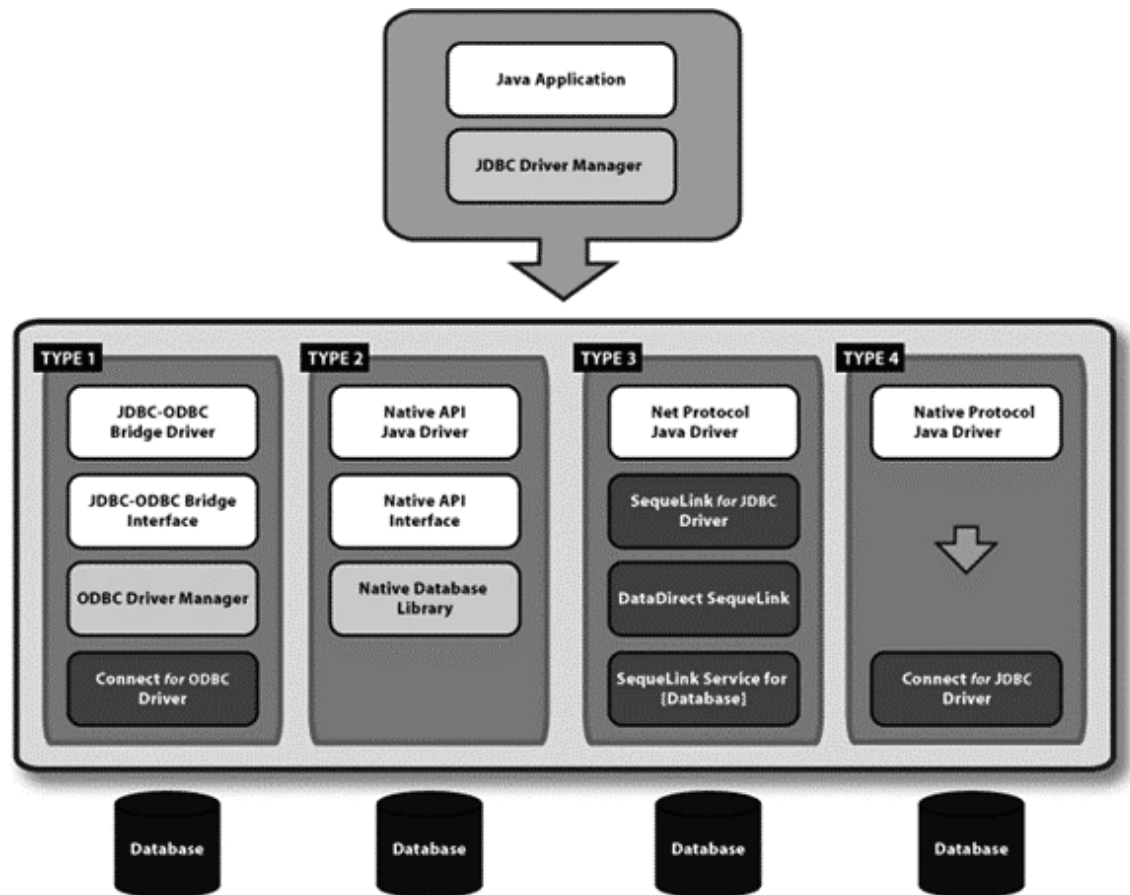


Figure 2: Comparison of different JDBC drivers

Check Your Progress 1

1) State True or False:

a) CallableStatement are used to call SQL stored procedures.

T ☐ F ☐

b) ODBC make use of pointers which have been removed from java.

T ☐ F ☐

To give answers of following questions:

2) What are the advantages of using JDBC with java?

.....

.....

.....

3) Briefly explain the advantages / disadvantages of different types of drivers of JDBC.

.....

.....

.....

4) Why ODBC cannot be used directly with Java programs?

.....

.....

.....

- 5) What are 3 different types of statements available in JDBC? Where do we use these statements?
-
-
-

2.6 STEPS TO CONNECT TO A DATABASE

Now, we shall learn step-by-step process to connect a database using Java. The interface and classes of the JDBC API are present inside the package called as java.sql package. There any application using JDBC API must import java.sql package in its code.

```
import java.sql.* ;
```

STEP 1: Load the Drivers

The first step in accessing the database is to load an appropriate driver. You can use one driver from the available four drivers which are described earlier. However, JDBC-ODBC Driver is the most preferred driver among developers. If you are using any other type of driver, then it should be installed on the system (usually this requires having the driver jar file available and its path name in your classpath. In order to load the driver, you have to give the following syntax:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

We can also register the driver (if third party driver) with the use of method **registerMethod()** whose syntax is as follows :

```
DriverManager.registerDriver(Driver dr);
```

Where dr is the new JDBC driver to be registered with the DriverManager.
There are a number of alternative ways to do the actual loading:

1. Use new to explicitly load the Driver class. This hard codes the driver and (indirectly) the name of the database into your program and is not recommended as changing the driver or the database or even the name or location of the database will usually require recompiling the program.
2. Class.forName takes a string class name and loads the necessary class dynamically at runtime as specified in the above example. This is a safe method that works well in all Java environments although it still requires extra coding to avoid hard coding the class name into the program.
3. The System class has a static Property list. If this has a Property jdbc.drivers set to a ':' separated list of driver class names, then all of these drivers will be loaded and registered automatically. Since there is support for loading property lists from files easily in Java, this is a convenient mechanism to set up a whole set of drivers. When a connection is requested, all loaded drivers are checked to see which one can handle the request and an appropriate one is chosen. Unfortunately, support for using this approach in servlet servers is patchy so we will stay with method 2 above but use the properties file method to load the database url and the driver name at runtime:

```
Properties props = new Properties() ;
FileInputStream in = new FileInputStream("Database.Properties") ;
props.load(in);
```

```
String drivers = props.getProperty("jdbc.drivers") ;  
Class.forName(drivers) ;
```

The Database.Properties file contents look like this:

```
# Default JDBC driver and database specification  
jdbc.drivers =  
sun.jdbc.odbc.JdbcOdbcDriver  
database.Shop = jdbc:odbc:Shop
```

STEP 2: Make the Connection

The getConnection() method of the Driver Manager class is called to obtain the Connection Object. The syntax looks like this:

```
Connection conn = DriverManager.getConnection("jdbc:odbc:<DSN NAME>");
```

Here note that getConnection() is a static method, meaning it should be accessed along with the class associated with the method. The DSN (Data Source name) Name is the name, which you gave in the Control Panel->ODBC while registering the Database or Table.

STEP 3: Create JDBC Statement

A Statement object is used to send SQL Query to the Database Management System. You can simply create a statement object and then execute it. It takes an instance of active connection to create a statement object. We have to use our earlier created Connection Object “conn” here to create the Statement object “stmt”. The code looks like this:

```
Statement stmt = conn.createStatement();
```

As mentioned earlier we may use Prepared Statement or callable statement according to the requirement.

STEP 4: Execute the Statement

In order to execute the query, you have to obtain the ResultSet object similar to Record Set in Visual Basic and call the **executeQuery()** method of the Statement interface. You have to pass a SQL Query like select * from students as a parameter to the executeQuery() method. Actually, the ResultSet object contains both the data returned by the query and the methods for data retrieval. The code for the above step looks like this:

```
ResultSet rs = stmt.executeQuery("select * from student");
```

If you want to select only the name field you have to issue a SQL Syntax like
Select Name from Student

The executeUpdate() method is called whenever there is a delete or an update operation.

STEP 5: Navigation or Looping through the ResultSet

The ResultSet object contains rows of data that is parsed using the next() method like rs.next(). We use the **getXXX()** like, (getInt to retrieve Integer fields and getString for String fields) method of the appropriate type to retrieve the value in each field. If the first field in each row of ResultSet is Name (Stores String value), then getString method is used. Similarly, if the Second field in each row stores int type, then getInt() method is used like:

```
System.out.println(rs.getInt("ID"));
```

STEP 6: Close the Connection and Statement Objects

After performing all the above steps, you must close the Connection, statement and Resultset Objects appropriately by calling the close() method. For example, in our above code we will close the object as:

ResultSet object with

```
rs.close();
```

and statement object with

```
stmt.close();
```

Connection object with

```
conn.close();
```

2.7 USING JDBC TO QUERY A DATABASE

Let us take an example to understand how to query or modify a database. Consider a table named as CUSTOMER is created in MS-ACCESS, with fields cust_id, name, ph_no, address etc.

```
import java.sql.*;

public class JdbcExample1 {

    public static void main(String args[]) {

        Connection con = null;

        Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);

        Connection Conn = DriverManager.getConnection (jdbc:odbc:Access);

        Statement Stmt = Conn.createStatement();

        // To create a string of SQL.

        String sql = "SELECT * FROM CUSTOMERS";

        // Next we will attempt to send the SQL command to the database.

        // If it works, the database will return to us a set of results that JDBC will

        // store in a ResultSet object.

        try

        {

            ResultSet results = Stmt.executeQuery(sql);

            // We simply go through the ResultSet object one element at a time and print //out the
fields. In this example, we assume that the result set will contain three //fields

            while (results.next())

            {
```

```
        System.out.println("Field One: " + results.getString(1) + "Field Two: " +
results.getString(2) + "Field Three: " + results.getString(3));

    }

}

// If there was a problem sending the SQL, we will get this error.

catch (Exception e)

{

    System.out.println("Problem with Sending Query: " + e);

}

finally

{

    result.close();

    stmt.close();

    Conn.close();

}

} // end of main method

} // end of class
```

Note that if the field is an Integer, you should use the `getInt()` method in `ResultSet` instead of `getString()`. You can use either an ordinal position (as shown in the above example) which starts from 1 for the first field or name of field to access the values from the `ResultSet` like `result.getString("CustomerID")`;

Compiling JdbcExample1.java

To compile the `JdbcExample1.java` program to its class file and place it according to its package statements, open command prompt and `cd` (change directory) into the folder containing `JdbcExample2.java`, then execute this command:

javac -d . JdbcExample2.java

If the program gets compiled successfully then you should get a new Java class under the current folder with a directory structure `JdbcExample2.class` in your current working directory.

Running JdbcExample1.java

To run and to see the output of this program execute following command from the command prompt from the same folder where `JdbcExample2.java` is residing:

java JdbcExample2

2.8 USING JDBC TO MODIFY A DATABASE

Modifying a database is just as simple as querying a database. However, instead of using `executeQuery()`, you use `executeUpdate()` and you don't have to worry about a result set. Consider the following example:

```
import java.sql.*;

public class JdbcExample1
{
    public static void main(String args[])
    {
        Connection con = null;

        Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);

        Connection Conn = DriverManager.getConnection (jdbc:odbc:Access);

        Statement Stmt = Conn.createStatement();

        // We have already seen all the above steps

        String sql = "INSERT INTO CUSTOMERS +
            " (CustomerID, Firstname, LastName, Email)" +
            " VALUES (004, 'Selena', 'Sol' " + " 'selena@extropia.com')";

        // Now submit the SQL....

        try
        {
            Stmt.executeUpdate(sql);
        } catch (Exception e)
        {
            System.out.println("Problem with Sending Query: " + e);
        }

        finally
        {
            result.close();

            stmt.close();

            Conn.close();
        }
    }
}
```



```
}  
  
} // end of main method  
  
} // end of class
```

As you can see, there is not much to it. Add, modify and delete are all handled by the executeUpdate() method or executeUpdate(String str) where str is a SQL Insert, Update or Delete Statement.

Check Your Progress 2

- 1) What is the most important package used in JDBC?
.....
.....
.....
- 2) Explain different methods to load the drivers in JDBC.
.....
.....
.....
- 3) Assume that there is a table named as Student in MS-Access with the following fields : Std_id, name, course, ph_no. Write a Java program to insert and then display the records of this table using JDBC.
.....
.....
.....
- 4) What is the fastest type of JDBC driver?
.....
.....
.....
- 5) Is the JDBC-ODBC Bridge multi-threaded?
.....
.....
.....
- 6) What's the JDBC 3.0 API?
.....
.....
.....
- 7) Can we use JDBC-ODBC Bridge with applets?
.....
.....
.....

- 8) How should one start debugging problems related to the JDBC API?
.....
.....
.....
- 9) Why sometimes programmer gets the error message “java.sql.DriverManager class” not being found? How can we remove these kind of errors?
.....
.....
.....
- 10) How can one retrieve a whole row of data at once, instead of calling an individual ResultSet.getXXX method for each column?
.....
.....
.....
- 11) Why do one get a NoClassDefFoundError exception when I try and load my driver?
.....
.....
.....

2.9 SUMMARY

JDBC is an API, which stands for Java Database connectivity, provides an interface through which one can have a uniform access to a wide range of relational databases like MS-Access, Oracle or Sybase. It also provides bridging to ODBC (Open Database Connectivity) by JDBC-ODBC Bridge, which implements JDBC in terms of ODBC standard C API. With the help of JDBC programming interface, Java programmers can request a connection with a database, then send query statements using SQL and receive the results for processing. The combination of Java with JDBC is very useful because it helps the programmer to run the program on different platforms in an easy and economical way. It also helps in implementing the version control.

JDBC API mainly consists of 4 main interfaces i.e. *java.sql DriverManager*, *java. sql .Connection*, *java. sql. Statement*, *java.sql.Resultset*. The main objects of JDBC are DataSource, Connection and statement. A DataSource object is used to establish connections. A Connection object controls the connection to the database. Statement object is used for executing SQL queries. Statement Object is further divided into three categories, which are statement, Prepared Statement and callable statement. Prepared Statement object is used to execute parameterized SQL queries and Callable statement is used to execute stored procedures within a RDBMS.

JDBC requires drivers to connect any of the databases. There are four types of drivers available in Java for database connectivity. First two drivers Type1 and Type 2 are impure Java drivers whereas Type 3 and Type 4 are pure Java drivers. Type 1 drivers act as a JDBC-ODBC bridge. It is useful for the companies that already have ODBC drivers installed on each client machine. Type2 driver is Native-API partly Java technology-enabled driver. It converts the calls that a developer writes to the JDBC application-programming interface into calls that connect to the client machine's application programming interface for a specific database like Oracle. Type-3 driver is A net-protocol fully Java technology-enabled driver and is written in 100% Java. . It

translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software. Type 4 is a native-protocol fully Java technology-enabled driver. It is Direct-to-database pure Java driver. It converts JDBC technology calls into the network protocol used by different DBMSs directly.

To connect a database we should follow certain steps. First step is to load an appropriate driver from any of the four drivers. We can also register the driver by `registerMethod()`. Second step is to make the connection with the database using a `getConnection()` method of the Driver Manager class which can be with DSN OR DSN-less connection. Third step is create appropriate JDBC statement to send the SQL query. Fourth step is to execute the query using `executeQuery()` method and to store the data returned by the query in the Resultset object. Then we can navigate the ResultSet using the `next()` method and `getXXX()` to retrieve the integer fields. Last step is to close the connection and statement objects by calling the `close()` method. In this way we can query the database, modify the database, delete the records in the database using the appropriate query.

2.10 SOLUTIONS / ANSWERS

Check Your Progress 1

- 1) True or False
 - a) True
 - b) True
- 2) JDBC is a standard SQL database access interface that provides uniform access to a wide range of relational databases. It also provides a common base on which higher level tools and interfaces can be built. The advantages of using Java with JDBC are:
 - Easy and economical
 - Continued usage of already installed databases
 - Development time is short
 - Installation and version control simplified
- 3) There are basically 4 types of drivers available in Java of which 2 are partly pure and 2 are pure java drivers.

Type 1: JDBC-ODBC Bridge.

They delegate the work of data access to ODBC API. This kind of driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important.

Advantages: It acts as a good approach for learning JDBC. It may be useful for companies that already have ODBC drivers installed on each client machine — typically the case for Windows-based machines running productivity applications.

Disadvantage: It is not suitable for large-scale applications. They are the slowest of all. The performance of system may suffer because there is some overhead associated with the translation work to go from JDBC to ODBC. It doesn't support all the features of Java.

Type 2: Native-API partly Java technology-enabled driver

They mainly use native API for data access and provide Java wrapper classes to be able to be invoked using JDBC drivers. It converts the calls that a developer writes to

the JDBC application programming interface into calls that connect to the client machine's application programming interface for a specific database, such as IBM, Informix, Oracle or Sybase

Advantage: It has a better performance than that of Type 1, in part because the Type 2 driver contains *compiled code* that's optimized for the back-end database server's operating system.

Disadvantage: In this, user needs to make sure the JDBC driver of the database vendor is loaded onto each client machine. It must have compiled code for every operating system that the application will run on.

Type 3: A net-protocol fully Java technology-enabled driver

They are written in 100% Java and use vendor independent Net-protocol to access a vendor independent remote listener. This listener in turn maps the vendor independent calls to vendor dependent ones.

Advantage: It has better performance than Types 1 and 2. It can be used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them. Since, it is server-based, so there is no requirement for JDBC driver code on client machine.

Disadvantage: It needs some database-specific code on the middleware server. If the middleware is to run on different platforms, then Type 4 driver might be more effective.

Type 4: A native-protocol fully Java technology-enabled driver

It is Direct-to-database pure Java driver. It converts JDBC technology calls into the network protocol used by different DBMSs directly. It allows a direct call from the client machine to the database.

Advantage: It again has better performance than Types 1 and 2 and there is no need to install special software on client or server. It can be downloaded dynamically.

Disadvantage: It is not optimized for server operating system, so the driver can't take advantage of operating system features. For this, user needs a different driver for each different database.

- 4) ODBC (Open Database Connectivity) cannot be used directly with java due to the following reasons:
 - a) ODBC cannot be used directly with Java because it uses a C interface. This will have drawbacks in the Security, implementation, and robustness.
 - b) ODBC makes use of Pointers, which have been removed from Java.
 - c) ODBC mixes simple and advanced features together and has complex structure.

- 5) Statement object is one of the main objects of JDBC API, which is used for executing the SQL queries. There are 3 different types of JDBC SQL Statements are available in Java:
 - a) **java.sql.Statement:** It is the topmost interface which provides basic methods useful for executing SELECT, INSERT, UPDATE and DELETE SQL statements.

b) **java.sql.PreparedStatement:** It is an enhanced version of `java.sql.Statement` which is used to execute SQL queries with parameters and can be executed multiple times.

c) **java.sql.CallableStatement:** It allows you to execute stored procedures within a RDBMS which supports stored procedures.

Check Your Progress 2

- 1) The most important package used in JDBC is `java.sql` package.
- 2) After importing the `java.sql.*` package the next step in database connectivity is load the appropriate driver. There are various ways to load these drivers:

1. `Class.forName` takes a string class name and loads the necessary class dynamically at runtime as specified in the example To load the JDBC-ODBC driver following syntax may be used :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

2. To register the third party driver one can use the method **registerMethod()** whose syntax is as follows :

```
DriverManager.registerDriver(Driver dr);
```

3. Use `new` to explicitly load the Driver class. This hard codes the driver and (indirectly) the name of the database into your program

4. The `System` class has a static `Property` list. If this has a `Property jdbc.drivers` set to a ':' separated list of driver class names, then all of these drivers will be loaded and registered automatically.

3)

```
import java.sql.*;
public class Student_JDBC {
public static void main(String args[])
{
    Connection con = null;
    Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);
    Connection Conn = DriverManager.getConnection (jdbc:odbc:Access);
    Statement Stmt = Conn.createStatement();
    Statement Stmt2 = Conn.createStatement();
    String sql1 = "INSERT INTO STUDENT + " (Std_id, name, course, ph_no)" +
        " VALUES (004, 'Sahil', 'MCA' " + "'SAHIL@rediffmail.com')";
    // Now submit the SQL....
    try
    {
        Stmt.executeUpdate(sql);
        String sql2 = "SELECT * FROM STUDENT";
        ResultSet results = Stmt2.executeQuery(sql);
        while (results.next())
        {
            System.out.println("Std_id: " + results.getString(1) + " Name: " +
results.getString(2) + " , course: " + results.getString(3) + " , ph_no: " +
results.getString(4));
        }
    }
}
// If there was a problem sending the SQL, we will get this error.
```

```

catch (Exception e)
{
    System.out.println("Problem with Sending Query: " + e);
}
finally
{
    result.close();
    stmt.close();
    stmt2.close();
    Conn.close();
}
} // end of main method
} // end of class

```

Compile and execute this program to see the output of this program. The assumption is made that table named as STUDENT with the required columns are already existing in the MS-Access.

- 4) Type 4 (JDBC Net pure Java Driver) is the fastest JDBC driver. Type 1 and Type 3 drivers will be slower than Type 2 drivers (the database calls are made at least three translations versus two), and Type 4 drivers are the fastest (only one translation).
- 5) No. The JDBC-ODBC Bridge does not support multi threading. The JDBC-ODBC Bridge uses synchronized methods to serialize all of the calls that it makes to ODBC. Multi-threaded Java programs may use the Bridge, but they won't get the advantages of multi-threading.
- 6) The JDBC 3.0 API is the latest update of the JDBC API. It contains many features, including scrollable result sets and the SQL:1999 data types.
- 7) We are not allowed to use of the JDBC-ODBC bridge from an untrusted applet running in a browser, such as Netscape Navigator. The JDBC-ODBC bridge doesn't allow untrusted code to call it for security reasons. This is good because it means that an untrusted applet that is downloaded by the browser can't circumvent Java security by calling ODBC. As we know that ODBC is native code, so once ODBC is called the Java programming language can't guarantee that a security violation won't occur. On the other hand, Pure Java JDBC drivers work well with applets. They are fully downloadable and do not require any client-side configuration.

Finally, we would like to note that it is possible to use the JDBC-ODBC bridge with applets that will be run in appletviewer since appletviewer assumes that applets are trusted. In general, it is dangerous to turn applet security off, but it may be appropriate in certain controlled situations, such as for applets that will only be used in a secure intranet environment. Remember to exercise caution if you choose this option, and use an all-Java JDBC driver whenever possible to avoid security problems.

- 8) There is one facility available to find out what JDBC calls are doing is to enable JDBC tracing. The JDBC trace contains a detailed listing of the activity occurring in the system that is related to JDBC operations.

If you use the DriverManager facility to establish your database connection, you use the DriverManager.setLogWriter method to enable tracing of JDBC operations. If you use a DataSource object to get a connection, you use the DataSource.setLogWriter method to enable tracing. (For pooled connections, you use the ConnectionPoolDataSource.setLogWriter method, and for

connections that can participate in distributed transactions, you use the `XADataSource.setLogWriter` method.)

- 9) This problem can be caused by running a JDBC applet in a browser that supports the JDK 1.0.2, such as Netscape Navigator 3.0. The JDK 1.0.2 does not contain the JDBC API, so the `DriverManager` class typically isn't found by the Java virtual machine running in the browser.

To remove this problem one doesn't require any additional configuration of your web clients. As we know that classes in the `java.*` packages cannot be downloaded by most browsers for security reasons. Because of this, many vendors of all-Java JDBC drivers supply versions of the `java.sql.*` classes that have been renamed to `jdbc.sql.*`, along with a version of their driver that uses these modified classes. If you import `jdbc.sql.*` in your applet code instead of `java.sql.*`, and add the `jdbc.sql.*` classes provided by your JDBC driver vendor to your applet's codebase, then all of the JDBC classes needed by the applet can be downloaded by the browser at run time, including the `DriverManager` class.

This solution will allow your applet to work in any client browser that supports the JDK 1.0.2. Your applet will also work in browsers that support the JDK 1.1, although you may want to switch to the JDK 1.1 classes for performance reasons. Also, keep in mind that the solution outlined here is just an example and that other solutions are possible.

- 10) The `ResultSet.getXXX` methods are the only way to retrieve data from a `ResultSet` object, which means that you have to make a method call for each column of a row. There is very little chance that it can cause performance problem. However, because it is difficult to see how a column could be fetched without at least the cost of a function call in any scenario.
- 11) The classpath may be incorrect and Java cannot find the driver you want to use.

To set the class Path:

The `CLASSPATH` environment variable is used by Java to determine where to look for classes referenced by a program. If, for example, you have an import statement for `my.package.mca`, the compiler and JVM need to know where to find the `my/package/mca` class.

In the `CLASSPATH`, you do not need to specify the location of normal J2SE packages and classes such as `java.util` or `java.io.IOException`.

You also do not need an entry in the `CLASSPATH` for packages and classes that you place in the ext directory (normally found in a directory such as `C:\j2sdk\jre\lib\ext`). Java will automatically look in that directory. So, if you drop your JAR files into the ext directory or build your directory structure off the ext directory, you will not need to do anything with setting the `CLASSPATH`.

Note that the `CLASSPATH` environment variable can specify the location of classes in directories and in JAR files (and even in ZIP files).

If you are trying to locate a jar file, then specify the entire path and jar file name in the `CLASSPATH`. (Example: `CLASSPATH=C:\myfile\myjars\myjar.jar`).

If you are trying to locate classes in a directory, then specify the path up to but not including the name of the package the classes are in. (If the classes are in a package called `my.package` and they are located in a directory called

C:\myclasses\here\my\package, you would set the classpath to be
CLASSPATH=C:\myclasses\classname).

The classpath for both entries would be
CLASSPATH=C:\myfile\myjars\myjar.jar;C:\myclasses\here.

2.11 FURTHER READINGS/REFERENCES

- Bernard Van Haeckem, *Jdbc: Java Database Connectivity*, Wiley Publication
- Bulusu Lakshman, *Oracle and Java Development*, Sams Publications.
- Prateek Patel, *Java Database Programming*, Corollis
- Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible*, Hungry Minds, Inc.
- Jaworski, *Java 2 Platform*, Techmedia

Reference websites:

- <http://www.developers.sun.com>
- www.javaworld.com
- www.jdbc.postgresql.org
- www.jtds.sourceforge.net
- www.en.wikipedia.org
- www.apl.jhu.edu
- <http://www.learnxpress.com>
- <http://www.developer.com>

UNIT 3 JAVA SERVER PAGES-I

Structure	Page Nos.
3.0 Introduction	52
3.1 Objectives	53
3.2 Overview of JSP	53
3.3 Relation of Applets and Servlets with JSP	56
3.4 Scripting Elements	58
3.5 JSP Expressions	59
3.6 JSP Scriptlets	59
3.7 JSP Declarations	60
3.8 Predefined Variables	61
3.9 Creating Custom JSP Tag Libraries using Nested Tags	65
3.10 Summary	69
3.11 Solutions/Answers	70
3.12 Further Readings/References	73

3.0 INTRODUCTION

Nowadays web sites are becoming very popular. These web sites are either static or dynamic. With a *static web page*, the client requests a web page from the server and the server responds by sending back the requested file to the client. Therefore with a static web page receives an exact replica of the page that exists on the server.

But these days web site requires a lot more than static content. Therefore, these days' dynamic data is becoming very important to everything on the Web, from online banking to playing games. *Dynamic web pages* are created at the time they are requested and their content gets based on specified criteria. For example, a Web page that displays the current time is dynamic because its content changes to reflect the current time. Dynamic pages are generated by an application on the server, receiving input from the client, and responding appropriately.

Therefore, we can conclude that in today's environment, dynamic content is critical to the success of any Web site. In this unit we will learn about Java Server Pages (JSP) i.e., an exciting new technology that provides powerful and efficient creation of dynamic contents. It is a presentation layer technology that allows static Web content to be mixed with Java code. JSP allows the use of standard HTML, but adds the power and flexibility of the Java programming language. JSP does not modify static data, so page layout and "look-and-feel" can continue to be designed with current methods. This allows for a clear separation between the page design and the application. JSP also enables Web applications to be broken down into separate components. This allows HTML and design to be done without much knowledge of the Java code that is generating the dynamic data. As the name implies, JSP uses the Java programming language for creating dynamic content. Java's object-oriented design, platform independence, and protected-memory model allow for rapid application development. Built-in networking and enterprise Application Programming Interfaces (APIs) make Java an ideal language for designing client-server applications. In addition, Java allows for extremely efficient code reuse by supporting the Java Bean and Enterprise Java Bean component models.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the need of JSP;
- understand the functioning of JSP;
- understand the relation of applets and servlets with JSP;
- know about various elements of JSP;
- explain various scripting elements of JSP;
- explain various implicit objects of JSP, and
- understand the concept of custom tags and process of creating custom tag libraries in JSP.

3.2 OVERVIEW OF JSP

As you have already studied in previous units, servlets offer several improvements over other server extension methods, but still suffer from a lack of presentation and business logic separation. Therefore, developers created some servlet-based environments that provided the desired separation. Some of these servlet-based environments gained considerable acceptance in the marketplace e.g., FreeMarker and WebMacro. Parallel to the efforts of these individual developers, the Java community worked to define a standard for a servlet-based server pages environment. The outcome was what we now know as JSP. Now, let us look at a brief overview of JSP: JSP is an extremely powerful choice for Web development. It is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. This gives developers a scripting interface to create powerful Java Servlets.

JSP uses server-side scripting that is actually translated into servlets and compiled before they are run

JSP pages provide tags that allow developers to perform most dynamic content operations without writing complex Java code. Advanced developers can add the full power of the Java programming language to perform advanced operations in JSP pages.

Server Pages

The goal of the server pages approach to web development is to support dynamic content without the performance problems or the difficulty of using a server API. The most effective way to make a page respond dynamically would be to simply modify the static page. Ideally, special sections to the page could be added that would be changed dynamically by the server. In this case pages become more like a page *template* for the server to process before sending. These are no longer normal web pages—they are now *server pages*.

The most popular server page approaches today are Microsoft Active Server Pages (ASP), JSP from Sun Microsystems Inc., and an open-source approach called PHP. Now, as you know, server pages development simplifies dynamic web development by allowing programmers to embed bits of program logic directly into their HTML pages. This embedded program logic is written in a simple scripting language, which depends on what your server supports. This scripting language could be VBScript, JavaScript, Java, or something else. At runtime, the server interprets this script and returns the results of the script's execution to the client. This process is shown in

Figure 1. In this Figure, the client requests a server page; the server replaces some sections of a template with new data, and sends this newly modified page to the client.

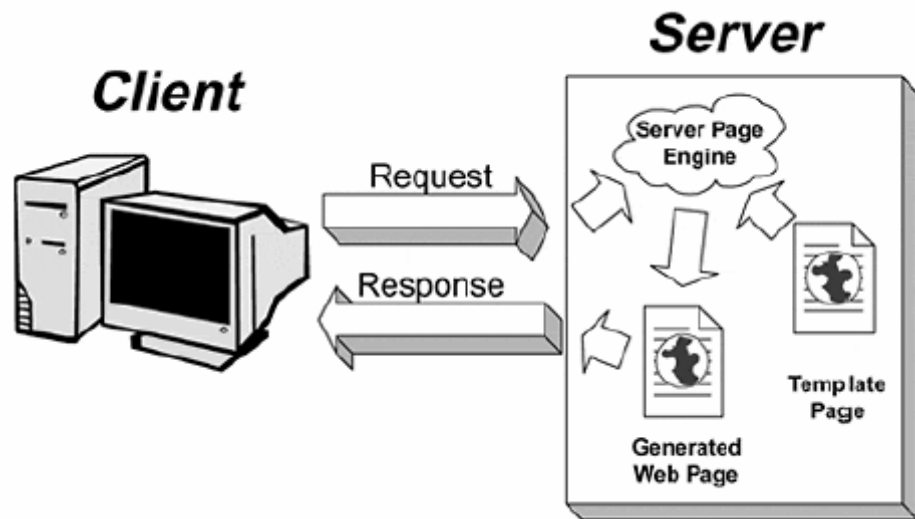


Figure 1: Server page

Separating Business and Presentation Logic

One of the greatest challenges in web development is in cleanly separating presentation and business logic. Most of the web server extension methods have suffered from this obstacle.

What does it mean to separate these layers? To start with, we can partition any application into two parts:

- **Business logic**

It is the portion of the application that solves the business need, e.g., the logic to look into the user's account, draw money and invest it in a certain stock. Implementing the business logic often requires a great deal of coding and debugging, and is the task of the programmer.

- **Presentation layer**

Presentation layer takes the results from the business logic execution and displays them to the user. The goal of the presentation layer is to create dynamic content and return it to the user's browser, which means that those responsible for the presentation layer are graphics designers and HTML developers.

Now, the question arises that if, applications are composed of a presentation layer and a business logic layer, what separates them, and why would we want to keep them apart? Clearly, there needs to be interaction between the presentation layer and the business logic, since, the presentation layer presents the business logic's results. But how much interaction should there be, and where do we place the various parts? At one extreme, the presentation and the business logic are implemented in the same set of files in a tightly coupled manner, so there is no separation between the two. At the other extreme, the presentation resides in a module totally separate from the one implementing the business logic, and the interaction between the two is defined by a set of well-known interfaces. This type of application provides the necessary

separation between the presentation and the business logic. But this separation is so crucial. Reason is explained here:

In most cases the developers of the presentation layer and the business logic are different people with different sets of skills. Usually, the developers of the presentation layer are graphics designers and HTML developers who are not necessarily skilled programmers. Their main goal is to create an easy-to-use, attractive web page. The goal of programmers who develop the business logic is to create a stable and scalable application that can feed the presentation layer with data. These two developers differ in the tools they use, their skill sets, their training, and their knowledge. When the layers aren't separated, the HTML and program code reside in the same place, as in CGI. Many sites built with those techniques have code that executes during a page request and returns HTML. Imagine how difficult it is to modify the User Interface if the presentation logic, for example HTML, is embedded directly in a script or compiled code. Though developers can overcome this difficulty by building template frameworks that break the presentation away from the code, this requires extra work for the developer since the extension mechanisms don't natively support such templating. Server pages technologies are not any more helpful with this problem. Many developers simply place Java, VBScript, or other scripting code directly into the same page as the HTML content. Obviously, this implies maintenance challenges as the server pages now contain content requiring the skills of both content developers and programmers. They must check that each updating of content to a specific server goes through without breaking the scripts inside the server page. This check is necessary because the server page is cluttered with code that only the business developer understands. This leaves the presentation developer walking on eggshells out of concern for preserving the work of the business logic developer. Worse, this arrangement can often cause situations in which both developers need to modify a single file, leaving them the tedious task of managing file ownership. This scenario can make maintaining a server pages-based application an expensive effort.

Separating these two layers is a problem in the other extension mechanisms, but the page-centric nature associated with server pages applications makes the problem much more pronounced. JSP separates the presentation layer (i.e., web interface logic) from the business logic (i.e. back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

Static and Dynamic contents in a JSP page

JSP pages usually contain a mixture of both static data and dynamic elements. *Static data* is never changed in the server page, and *dynamic elements* will always be interpreted and replaced before reaching the client.

JSP uses HTML or XML to incorporate static elements in a web page. Therefore, format and layout of the page in JSP is built using HTML or XML.

As well as these static elements a JSP page also contains some elements that will be interpreted and replaced by the server before reaching the client. In order to replace sections of a page, the server needs to be able to recognise the sections it needs to change. For this purpose a JSP page usually has a special set of tags to identify a portion of the page that should be modified by the server. JSP uses the `<%` tag to note the start of a JSP section, and the `%>` tag to note the end of a JSP section. JSP will interpret anything within these tags as a special section. These tags are known as scriptlets.

When the client requests a JSP page, the server translates the server page and client receives a document as HTML. This translation process used at server is displayed in

Figure 2. Since, the processing occurs on the server, the client receives what appears to be static data. As far as the client is concerned there is no difference between a server page and a standard web page. This creates a solution for dynamic pages that does not consume client resources and is completely browser neutral.

Resulting HTML

Template

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.0 Final//EN">
<HTML>
<HEAD>
<TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
  The time on the server is
  Wed Aug 17 17:10:05 PST 2006
</BODY>
</HTML>
```

Server Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.0 Final//EN">
<HTML>
<HEAD>
<TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
  The time on the server is
  <%= new java.util.Date() %>
</BODY>
</HTML>
```



Scriptlets

Figure 2: A Server Page into HTML Data

3.3 RELATION OF APPLETS AND SERVLETS WITH JSP

Now, in this topic we shall compare applets, servlets and JSP and shall try to make a relationship among these.

Let us start with the **Applets**. These are small programs that are downloaded into a Java Enabled Web Browser, like, Netscape Navigator or Microsoft Internet Explorer. The browser will execute the applet on the client's machine. The downloaded applet has very limited access to the client machine's file system and network capabilities. These limitations ensure that the applet can't perform any malicious activity on the client's machine, such as deleting files or installing viruses. By default, a downloaded applet cannot read or write files from the file system, and may use the network only to communicate back to the server of origin. Using security certificates, applets can be given permission to do anything on the user's machine that a normal Java application can do. This may be impractical for Extranet applications; however, as users may require support to give these permissions or may not trust an organization enough to grant such permission.

Applets greatly *enhance* the user interface available through a browser. Applets can be created to act exactly like any other client-server GUI application including menus, popup dialog windows, and many other user-friendly features not otherwise available in a web browser environment.

But main *problem* with applet is it's long setup time over modems. Applets need to be downloaded over the Internet. Instead of just downloading the information to be displayed, a browser must download the whole application to execute it. The more functionality the applet provides, the longer it will take to download. Therefore, applets are best suited for applications that either run on an Intranet, or are small enough to download quickly and don't require special security access.

Next, **Servlet** is a Java program that runs in conjunction with a Web Server. A servlet is executed in response to an HTTP request from a client browser. The servlet executes and then returns an HTML page back to the browser.

Some major advantages of servlets are:

- Servlets handle multiple requests. Once a servlet is started it remains in memory and can handle multiple HTTP requests. In contrast, other server side script e.g. CGI program ends after each request and must be restarted for each subsequent request, reducing performance.
- Servlets support server side execution. Servlets do not run on the client, all execution takes place on the server. While, they provide the advantages of generating dynamic content, they do not levy the same download time requirement as applets.

Major *problem* with servlets is their limited functionality. Since they deliver HTML pages to their clients, the user interface available through a servlet is limited by what the HTML specification supports.

Next, as you know, a **JSP** is text document that describes how a server should handle specific requests. A JSP is run by a JSP Server, which interprets the JSP and performs the actions the page describes. Frequently, the *JSP server compiles the JSP into a servlet* to enhance performance. The server would then periodically check the JSP for changes and if there is any change in JSP, the server will recompile it into a servlet. *JSPs have the same advantages and disadvantages as servlets when compared to applets.*

- **JSP is Easier to Develop and Maintain than Servlets**

To the developer, JSPs look very similar to static HTML pages, except that they contain special tags used to identify and define areas that contain Java functionality. Because of the close relationship between JSPs and the resulting HTML page, JSPs are easier to develop than a servlet that performs similar operations. Because they do not need to be compiled, JSPs are easier to maintain and enhance than servlets.

- **JSP's Initial Access Slower than Servlets**

However, because they need to be interpreted or compiled by the server, response time for initial accesses may be slower than servlets.

Check Your Progress 1

Give right choice for the following:

- 1) JSP uses server-side scripting that is actually translated into ----- and compiled before they are run
 - a) Applet
 - b) Servlets
 - c) HTML
- 2) Presentation layer defines -----
 - a) Web interface logic
 - b) Back-end content generation logic

Explain following question in brief

- 3) What is JSP ? Explain its role in the development of web sites.

.....
.....
.....

3.4 SCRIPTING ELEMENTS

Now, after going through the basic concepts of JSP, we will understand different types of tags or scripting elements used in JSP.

A JSP page contains HTML (or other text-based format such as XML) mixed with elements of the JSP syntax.

There are five basic types of elements, as well as a special format for comments. These are:

1. **Scriptlets** :The Scriptlet element allows Java code to be embedded directly into a JSP page.
2. **Expressions**:An expression element is a Java language expression whose value is evaluated and returned as a string to the page.
3. **Declarations**: A declaration element is used to declare methods and variables that are initialized with the page.
4. **Actions**: Action elements provide information for the translation phase of the JSP page, and consist of a set of standard, built-in methods. Custom actions can also be created in the form of custom tags. This is a new feature of the JSP 1.1 specification.
5. **Directives**: Directive elements contain global information that is applicable to the whole page.

The first three elements—Scriptlets, Expressions, and Declarations—are collectively called **scripting elements**.

There are two different *formats* in which these elements can be used in a JSP page:

- **JSP Syntax**

The first type of format is called the JSP syntax. It is based on the syntax of other Server Pages, so it might seem very familiar. It is symbolized by: `<% script %>`. The JSP specification refers to this format as the “friendly” syntax, as it is meant for hand-authoring.

JSP Syntax: <code><% code %></code>
--

- **XML Standard Format**

The second format is an XML standard format for creating JSP pages. This format is symbolized by: `<jsp:element />`.

XML syntax would produce the same results, but JSP syntax is recommended for authoring.

XML Syntax: <jsp:scriptlet > code </jsp:scriptlet>

Now, we will discuss about these scripting elements in detail.

3.5 JSP EXPRESSIONS

JSP Syntax: <%= code %>

XML Syntax: <jsp:expression > code </jsp:expression>

Printing the output of a Java fragment is one of the most common tasks utilized in JSP pages. For this purpose, we can use the `out.println()` method. But having several `out.println()` method tends to be cumbersome. Realizing this, the authors of the JSP specification created the Expression element. The Expression element begins with the standard JSP start tag followed by an equals sign (<%=).

Look at example 3.1. In this example, notice that the `out.println()` method is removed, and immediately after the opening JSP tag there is an equals symbol.

Example 3.1 date.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">
```

```
<HTML>
```

```
  <HEAD>
```

```
    <TITLE>Current Date</TITLE>
```

```
  </HEAD>
```

```
  <BODY>
```

```
    The current date is:
```

```
    <%= new java.util.Date() %>
```

```
  </BODY>
```

```
</HTML>
```

Expression element

3.6 JSP SCRIPTLETS

JSP Syntax: <% code %>

XML Syntax: <jsp:scriptlet > code </jsp:scriptlet>


Scriptlets are the most common JSP syntax element. As you have studied above, a scriptlet is a portion of regular Java code embedded in the JSP content within <% ... %> tags. The Java code in scriptlets is executed when the user asks for the page. Scriptlets can be used to do absolutely anything the Java language supports, but some of their more common tasks are:

- Executing logic on the server side; for example, accessing a database.
- Implementing conditional HTML by posing a condition on the execution of portions of the page.
- Looping over JSP fragments, enabling operations such as populating a table with dynamic content.

A simple use of these scriptlet tags is shown in Example 3.2. In this example you need to notice the two types of data in the page, i.e., static data and dynamic data. Here, you need not to worry too much about what the JSP page is doing; that will be covered in later chapters.

Example 3.2 simpleDate.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">
<HTML>
<HEAD>
    <TITLE>A simple date example</TITLE>
</HEAD>
<BODY COLOR=#ffffff>
    The time on the server is
    <%= new java.util.Date() %>
</BODY>
</HTML>
```



A line points from the text "Scriptlets" to the scriptlet tag `<%= new java.util.Date() %>` in the code block above.

When the client requests this JSP page, the client will receive a document as HTML. The translation process used at server is displayed in *Figure 2*.

3.7 JSP DECLARATIONS

JSP Syntax: `<%! code %>`

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

The third type of Scripting element is the Declaration element. The purpose of a declaration element is to initialize variables and methods and make them available to other Declarations, Scriptlets, and Expressions. Variables and methods created within Declaration elements are effectively global. The syntax of the Declaration element begins with the standard JSP open tag followed by an exclamation point (`<%!`). The Declaration element must be a complete Java statement. It ends with a semicolon, just as the Scriptlet element does.

Look at example 3.3. In this example an instance variable named `Obj` and the initialization and finalisation methods `jspInit` and `jspDestroy`, have been done using declaration element.

Example 3.3 Declaration element

```
<%!
    private ClasJsp Obj;
    public void jspInit()
    {
        ...
    }
    public void jspDestroy()
    {
        ...
    }
%>
```

Check Your Progress 2

Give right choice for the following:

- 1) ----- contain global information that is applicable to the whole page.
 - a) Actions elements
 - b) Directive elements
 - c) Declarations elements
 - d) Scriptlets elements
- 2) For incorporating Java code with HTML, we will use -----.
 - a) Actions elements
 - b) Directive elements
 - c) Declarations elements
 - d) Scriptlets elements

Explain the following question in brief

- 3) Explain various scripting elements used in JSP.

.....

.....

.....

3.8 PREDEFINED VARIABLES

To simplify code in JSP expressions and scriptlets, Servlet also creates several objects to be used by the JSP engine; these are sometimes called implicit objects (or predefined variables). Many of these objects are called directly without being explicitly declared. These objects are:

1. The out Object
2. The request Object
3. The response Object
4. The pageContext Object
5. The session object
6. The application Object
7. The config Object
8. The page Object
9. The exception Object

• The out Object

The major function of JSP is to describe data being sent to an output stream in response to a client request. This output stream is exposed to the JSP author through the implicit out object. The out object is an instantiation of a `javax.servlet.jsp.JspWriter` object. This object may represent a direct reference to the output stream, a filtered stream, or a nested `JspWriter` from another JSP. Output should never be sent directly to the output stream, because there may be several output streams during the lifecycle of the JSP.

The initial `JspWriter` object is instantiated differently depending on whether the page is buffered or not. By default, every JSP page has buffering turned on, which almost

always improves performance. Buffering be easily turned off by using the buffered= 'false' attribute of the page directive.

A buffered out object collects and sends data in blocks, typically providing the best total throughput. With buffering the PrintWriter is created when the first block is sent, actually the first time that flush() is called.

With unbuffered output the PrintWriter object will be immediately created and referenced to the out object. In this situation, data sent to the out object is immediately sent to the output stream. The PrintWriter will be created using the default settings and header information determined by the server.

In the case of a buffered out object the OutputStream is not established until the first time that the buffer is flushed. When the buffer gets flushed depends largely on the autoFlush and bufferSize attributes of the page directive. It is usually best to set the header information before anything is sent to the out object. It is very difficult to set page headers with an unbuffered out object. When an unbuffered page is created the OutputStream is established almost immediately.

The sending headers after the OutputStream has been established can result in a number of unexpected behaviours. Some headers will simply be ignored, others may generate exceptions such as IllegalStateException.

The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions. In JSP these exceptions need to be explicitly caught and dealt with. More about the out object is covered in Chapter 6.

Setting the autoFlush= 'false' attribute of the page directives will cause a buffer overflow to throw an exception.

- **The request Object**

Each time a client requests a page the JSP engine creates a new object to represent that request. This new object is an instance of javax.servlet.http.HttpServletRequest and is given parameters describing the request. This object is exposed to the JSP author through the request object.

Through the request object the JSP page is able to react to input received from the client. Request parameters are stored in special name/value pairs that can be retrieved using the request.getParameter(name) method.

The request object also provides methods to retrieve header information and cookie data. It provides means to identify both the client and the server, e.g., it uses request.getRequestURI() and request.getServerName() to identify the server.

The request object is inherently limited to the request scope. Regardless of how the page directives have set the scope of the page, this object will always be recreated with each request. For each separate request from a client there will be a corresponding request object.

- **The response Object**

Just as the server creates the request object, it also creates an object to represent the response to the client.

The object is an instance of `javax.servlet.http.HttpServletResponse` and is exposed to the JSP author as the response object.

The response object deals with the stream of data back to the client. The out object is very closely related to the response object. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP author can add new cookies or date stamps, change the MIME content type of the page, or start “server-push” methods. The response object also contains enough information on the HTTP to be able to return HTTP status codes, such as forcing page redirects.

- **The pageContext Object**

The pageContext object is used to represent the entire JSP page. It is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The application, config, session, and out objects are derived by accessing attributes of this object. The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope. The pageContext object does more than just act as a data repository. It is this object that manages nested JSP pages, performing most of the work involved with the forward and include actions. The pageContext object also handles uncaught exceptions.

From the perspective of the JSP author this object is useful in deriving information about the current JSP page's environment. This can be particularly useful in creating components where behavior may be different based on the JSP page directives.

- **The session object**

The session object is used to track information about a particular client while using stateless connection protocols, such as HTTP. Sessions can be used to store arbitrary information between client requests.

Each session should correspond to only one client and can exist throughout multiple requests. Sessions are often tracked by URL rewriting or cookies, but the method for tracking of the requesting client is not important to the session object.

The session object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

- **The application Object**

The application object is direct wrapper around the `ServletContext` object for the generated Servlet. It has the same methods and interfaces that the `ServletContext` object does in programming Java Servlets.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method, the JSP page is recompiled, or the JVM crashes. Information stored in this object remains available to any object used within the JSP page.

The application object also provides a means for a JSP to communicate back to the server in a way that does not involve “requests”. This can be useful for finding out information about the MIME type of a file, sending log information directly out to the server's log, or communicating with other servers.

- **The config Object**

The config object is an instantiation of `javax.servlet.ServletConfig`. This object is a direct wrapper around the `ServletConfig` object for the generated servlet. It has the same methods and interfaces that the `ServletConfig` object does in programming Java Servlets. This object allows the JSP author access to the initialisation parameters for the Servlet or JSP engine. This can be useful in deriving standard global information, such as the paths or file locations.

- **The page Object**

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. When the JSP page is first instantiated the page object is created by obtaining a reference to this object. So, the page object is really a direct synonym for this object.

However, during the JSP lifecycle, this object may not refer to the page itself. Within the context of the JSP page, the page object will remain constant and will always represent the entire JSP page.

- **The exception Object**

The error handling method utilises this object. It is available only when the previous JSP page throws an uncaught exception and the `<% @ pageerrorPage= "..." %>` tag was used. The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

A summarized picture of these predefined variables (implicit objects) is given in *Table 4*.

Table 4: Implicit Objects in JSP

Variable	Class	Description
out	<code>javax.servlet.jsp.JspWriter</code>	The output stream.
request	Subtype of <code>javax.servlet.HttpServletRequest</code>	The request triggering the execution of the JSP page.
response	Subtype of <code>javax.servlet.HttpServletResponse</code>	The response to be returned to the client. Not typically used by JSP page authors.
pageContext	<code>javax.servlet.jsp.PageContext</code>	The context for the JSP page. Provides a single API to manage the various scoped attributes described in Sharing Information. This API is used extensively when implementing tag handlers.
Session	<code>javax.servlet.http.HttpSession</code>	The session object for the client..
application	<code>javax.servlet.ServletContext</code>	The context for the JSP page's servlet and any Web components contained in the same application.
config	<code>javax.servlet.ServletConfig</code>	Initialization information for the JSP page's servlet.
page	<code>java.lang.Object</code>	The instance of the JSP page's servlet processing the current request. Not typically used by JSP page authors.
exception	<code>java.lang.Throwable</code>	Accessible only from an error page.

3.9 CREATING CUSTOM JSP TAG LIBRARIES USING NESTED TAGS

Ok up to now, you have studied the basic elements of JSP. Now in this topic we will learn about the creation of custom tag libraries in JSP.

As you already know, a **tag** is a group of characters read by a program for the purpose of instructing the program to perform an action. In the case of HTML tags, the program reading the tags is a web browser, and the actions range from painting words or objects on the screen to creating forms for data collection.

In the same way, a **custom tag** is a user-defined JSP language element. Custom JSP tags are also interpreted by a program; but, unlike HTML, JSP tags are interpreted on the server side not client side. The program that interprets custom JSP tags is the runtime engine in your application server as Tomcat, JRun, WebLogic etc. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of *features*. They can

- Be customised via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other. You can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag.
- Be nested within one another, allowing for complex interactions within a JSP page.

Custom Tag Syntax

The syntax of custom tag is exactly the same as the syntax of JSP actions. A slight difference between the syntax of JSP actions and custom tag is that the JSP action prefix is `jsp`, while a custom tag prefix is determined by the prefix attribute of the `taglib` directive used to instantiate a set of custom tags. The prefix is followed by a colon and the name of the tag itself.

As shown below, the format of a standard custom tag looks like:

```
<utility:repeat number= "12">Hello World!</utility:repeat>
```

Here, a tag library named `utility` is referenced. The specific tag used is named `repeat`. The tag has an attribute named `number`, which is assigned a value of `"12"`. The tag contains a body that has the text `"Hello World!"`, and then the tag is closed.

The Components That Make Up a Tag Library

To use custom JSP tags, you need to define three separate components:

- a) **Tag handler class** that defines the tag's behaviour,
- b) **Tag library descriptor file** that maps the XML element names to the tag implementations and
- c) **The JSP file** that uses the tag library.

Now in the following section, we will read an overview of each of these components, and learn how to build these components for various styles of tags.

- **The Tag Handler Class**

To define a new tag, first you have to define a Java class that tells the system what to do when it sees the tag. This class must implement the `javax.servlet.jsp.tagext.Tag` interface. This is usually accomplished by extending the `TagSupport` or `BodyTagSupport` class. Example 3.4 is an example of a simple tag that just inserts “Custom tag example (coreservlets.tags.ExampleTag)” into the JSP page wherever the corresponding tag is used.

Don’t worry about understanding the exact behavior of this class. For now, just note that it is in the `coreservlets.tags` class and is called `ExampleTag`. Thus, with Tomcat 3.1, the class file would be in `install_dir/webapps/ROOT/WEB-INF/classes/coreservlets/tags/ExampleTag.class`.

Example 3.4 ExampleTag.java

```
package coreservlets.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Very simple JSP tag that just inserts a string
 * (“Custom tag example...”) into the output.
 * The actual name of the tag is not defined here;
 * that is given by the Tag Library Descriptor (TLD)
 * file that is referenced by the taglib directive
 * in the JSP file.
 */

public class ExampleTag extends TagSupport {
    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print(“Custom tag example “ +
                “(coreservlets.tags.ExampleTag)”);
        } catch(IOException ioe) {
            System.out.println(“Error in ExampleTag: ” + ioe);
        }
        return(SKIP_BODY);
    }
}
```

- **The Tag Library Descriptor File**

After defining a tag handler, your next task is to identify the class to the server and to associate it with a particular XML tag name. This task is accomplished by means of a tag library descriptor file (in XML format) like the one shown in Example 3.5. This

file contains some fixed information, an arbitrary short name for your library, a short description, and a series of tag descriptions. The bold part of the example is the same in virtually all tag library descriptors.

Don't worry about the format of tag descriptions. For now, just note that the tag element defines the main name of the tag (really tag suffix, as will be seen shortly) and identifies the class that handles the tag. Since, the tag handler class is in the `coreservlets.tags` package, the fully qualified class name of `coreservlets.tags.ExampleTag` is used. Note that this is a class name, not a URL or relative path name. The class can be installed anywhere on the server that beans or other supporting classes can be put. With Tomcat 3.1, the standard base location is `install_dir/webapps/ROOT/WEB-INF/classes`, so `ExampleTag` would be in `install_dir/webapps/ROOT/WEB-INF/classes/coreservlets/tags`. Although it is always a good idea to put your servlet classes in packages, a surprising feature of Tomcat 3.1 is that tag handlers are required to be in packages.

Example 3.5 `csajsp-taglib.tld`

```
<?xml version= "1.0" encoding= "ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN "
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->
<taglib>
  <!-- after this the default space is
    "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
  -->
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    A tag library from Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>
  <tag>
    <name>example</name>
    <tagclass>coreservlets.tags.ExampleTag</tagclass>
    <info>Simplest example: inserts one line of output</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>
  <!-- Other tags defined later... -->
</taglib>
```


- **The JSP File**

Once, you have a tag handler implementation and a tag library description, you are ready to write a JSP file that makes use of the tag. Example 3.6 shows a JSP file. Somewhere before the first use of your tag, you need to use the taglib directive. This directive has the following form:

```
<%@ taglib uri= “...” prefix= “...” %>
```

The required uri attribute can be either an absolute or relative URL referring to a tag library descriptor file like the one shown in Example 3.5. To complicate matters a little, however, Tomcat 3.1 uses a web.xml file that maps an absolute URL for a tag library descriptor to a file on the local system. I don't recommend that you use this approach.

The prefix attribute, also required, specifies a prefix that will be used in front of whatever tag name the tag library descriptor defined. For example, if the TLD file defines a tag named tag1 and the prefix attribute has a value of test, the actual tag name would be test:tag1. This tag could be used in either of the following two ways, depending on whether it is defined to be a container that makes use of the tag body:

```
<test:tag1>
    Arbitrary JSP
</test:tag1>
or just
<test:tag1 />
```

To illustrate, the descriptor file of Example 3.5 is called csajsp-taglib.tld, and resides in the same directory as the JSP file shown in Example 3.6. Thus, the taglib directive in the JSP file uses a simple relative URL giving just the filename, as shown below.

```
<%@ taglib uri= “csajsp-taglib.tld” prefix= “csajsp” %>
```

Furthermore, since the prefix attribute is csajsp (for Core Servlets and JavaServer Pages), the rest of the JSP page uses csajsp:example to refer to the example tag defined in the descriptor file.

Example 3.6 SimpleExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
<TITLE><csajsp:example /></TITLE>
<LINK REL=STYLESHEET
    HREF="JSP-Styles.css"
    TYPE="text/css">
</HEAD>
<BODY>
<H1><csajsp:example /></H1>
<csajsp:example />
</BODY>
</HTML>
```



Figure 3: Custom tag example

Check Your Progress 3

Give right choice for the following:

- 1) ----- object is an instantiation of a `avax.servlet.jsp.JspWriter` object.
 - a) page
 - b) config
 - c) out
 - d) response
- 2) ----- object is used to track information about a particular client while using stateless connection protocols.
 - a) request
 - b) session
 - c) out
 - d) application

Explain following questions in brief

- 3) What are various implicit objects used with JSP..

.....

.....

.....
- 4) What is a custom tags. in JSP? What are the components that make up a tag library in JSP?

.....

.....

.....

3.10 SUMMARY

In this unit, we first studied the static and dynamic web pages. With a static web page, the client requests a web page from the server and the server responds by sending back the requested file to the client, server doesn't process the requested page at its

end. But dynamic web pages are created at the time they are requested and their content gets based on specified criteria. These pages are generated by an application on the server, receiving input from the client, and responding appropriately. For creation of these dynamic web pages, we can use JSP; it is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. It separates the presentation layer (i.e., web interface logic) from the business logic (i.e., back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

There are five basic types of elements in JSP. These are *scriptlets*, *expressions*, *declarations*, *actions* and *directives*. Among these elements the first three elements, i.e., scriptlets, expressions, and declarations, are collectively called scripting elements. Here **scriptlet** (`<%...%>`) element allows Java code to be embedded directly into a JSP page, **expression element** (`<%=...%>`) is used to print the output of a Java fragment and **declaration element** (`<%! code %>`) is used to initialise variables and methods and make them available to other declarations, scriptlets, and expressions. Next, we discussed about the implicit objects of JSP. Various implicit objects of JSP are out, request, response, pageContext, session, application, config, page and exception object. Here, **out object** refers to the output stream, **request object** contains parameters describing the request made by a client to JSP engine, **response object** deals with the stream of data back to the client, **pageContext object** is used to represent the entire JSP page, **session object** is used to track information about a particular client while using stateless connection protocols such as HTTP, **application object** is a representation of the JSP page through its entire lifecycle, **config object** allows the JSP author access to the initialisation parameters for the servlet or JSP engine, **page object** is an actual reference to the instance of the page and **exception object** is a wrapper containing the exception thrown from the previous page. Finally we studied about the custom tags. These are user-defined JSP language elements. Unlike HTML, these custom tags (JSP tags) are interpreted on the server side not client side. To use custom JSP tags, you need to define three separate components, i.e., tag handler class, tag library descriptor file and the JSP file. Here, tag handler class defines the tag's behaviour, tag library descriptor file maps the XML element names to the tag implementations and the JSP file uses the tag library.

3.11 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) b
- 2) a
- 3) JSP is an exciting new technology that provides powerful and efficient creation of dynamic contents. It allows static web content to be mixed with Java code. It is a technology using server-side scripting that is actually translated into servlets and compiled before they are run. This gives developers a scripting interface to create powerful Java Servlets.

Role of JSP in the development of websites:

In today's environment, dynamic content is critical to the success of any web site. There are a number of technologies available for incorporating the dynamic contents in a site. But most of these technologies have some problems. Servlets offer several improvements over other server extension methods, but still suffer from a lack of presentation and business logic separation. Therefore the Java community worked to define a standard for a servlet-based server pages environment and the outcome was what we now know as JSP. JSP separates the

presentation layer (i.e., web interface logic) from the business logic (i.e., back-end content generation logic) so that web designers and web developers can work on the same web page without getting in each other's way.

Check Your Progress 2

- 1) b
- 2) d
- 3) There are five basic types of elements used in JSP. These are:

(i) Scriptlets

JSP Syntax: `<% code %>`

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

The Scriptlet element allows Java code to be embedded directly into a JSP page.

(ii) Expressions

JSP Syntax: `<%= code %>`

XML Syntax: `<jsp:expression > code </jsp:expression>`

An expression element is a Java language expression whose value is evaluated and returned as a string to the page.

(iii) Declarations

JSP Syntax: `<%! code %>`

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

A declaration element is used to declare methods and variables that are initialized with the page.

(iv) Actions

Action elements provide information for the translation phase of the JSP page, and consist of a set of standard, built-in methods. Custom actions can also be created in the form of custom tags. This is a new feature of the JSP 1.1 specification.

(v) Directives

Directive elements contain global information that is applicable to the whole page.

The first three elements — Scriptlets, Expressions, and Declarations — are collectively called **scripting elements**.

Check Your Progress 3

- 1) c
- 2) c
- 3) To simplify code in JSP expressions and scriptlets, servlet creates several objects to be used by the JSP engine; these are sometimes called implicit objects. These objects are:

i) **The out object:** It refers to the output stream

- ii) **The request object:** It contains parameters describing the request made by a client to JSP engine.
 - iii) **The response object:** This object deals with the stream of data back to the client.
 - iv) **The pageContext object:** This object is used to represent the entire JSP page.
 - v) **The session object:** This object is used to track information about a particular client while using stateless connection protocols such as HTTP.
 - vi) **The application object:** This object is a representation of the JSP page through its entire lifecycle.
 - vii) **The config object:** This object allows the JSP author access to the initialization parameters for the servlet or JSP engine.
 - viii) **The page object:** This object is an actual reference to the instance of the page.
 - ix) **The exception object:** This object is a wrapper containing the exception thrown from the previous page.
- 4) A custom tag is a user-defined JSP language element. Custom JSP tags are also interpreted by a program; but, unlike HTML, JSP tags are interpreted on the server side not client side. The program that interprets custom JSP tags is the runtime engine in the application server as Tomcat, JRun, WebLogic, etc. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of *features*. They can

- Be customized via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other.

To use custom JSP tags, we need to define three separate components:

- a) **Tag handler class** that defines the tag's behaviour,
- b) **Tag library descriptor file** that maps the XML element names to the tag implementations, and
- c) **The JSP file** that uses the tag library.

3.12 FURTHER READINGS/REFERENCES

- Phil Hanna, *JSP: The Complete Reference*
- Hall, Marty, *Core Servlets and JavaServer Pages*
- Annunziato Jose & Fesler, Stephanie, *Teach Yourself JavaServer Pages in 24 Hours*,
- Bruce W. Perry, *Java Servlet & JSP, Cookbook* (Paperback).

UNIT 4 JAVA SERVER PAGES-II

Structure	Page Nos.
4.0 Introduction	73
4.1 Objectives	73
4.2 Database handling in JSP	74
4.3 Including Files and Applets in JSP Documents	78
4.4 Integrating Servlet and JSP	83
4.4.1 Forwarding Requests	
4.4.2 Including Static or Dynamic Content	
4.4.3 Forwarding Requests from JSP Pages	
4.5 Summary	89
4.6 Solutions/Answers	89
4.7 Further Readings/References	90

4.0 INTRODUCTION

In the previous unit, we have discussed the importance of JSP. We also studied the basic concepts of JSP. Now, in this unit, we will discuss some more interesting features of JSP. As you know, JSP is mainly used for server side coding.

Therefore, database handling is the core aspect of JSP. In this unit, first you will study about database handling through JSP. In that you will learn about administratively register a database, connecting a JSP to an Access database, insert records in a database using JSP, inserting data from HTML Form in a database using JSP, delete Records from Database based on Criteria from HTML Form and retrieve data from a database using JSP – result sets.

Then you will learn about how to include files and applets in JSP documents. In this topic you mainly learn about three main capabilities for including external pieces into a JSP document, i.e., `jsp:include` action, `include` directive and `jsp:plugin` action.

Finally, you will learn about integration of servlet and JSP. In this topic you learn about how to forward the requests, how to include static or dynamic content and how to forward requests from JSP Pages.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- understand how to connect JSP with a database;
- understand how to select, insert and delete records in database using JSP;
- understand how to include files and applets in JSP documents;
- understand how to include the output of JSP, HTML or plain text pages at the time the client requests the page;
- understand how to include JSP files at the time the main page is translated into a servlet;
- understand how to include applets that use the Java Plug-In, and
- understand how to integrate servlet and JSP.

4.2 DATABASE HANDLING IN JSP

We have already seen how to interface an HTML Form and a JSP. Now, we have to see how that JSP can talk to a database. In this section, we will understand how to:

1. Administratively register a database.
2. Connect a JSP to an Access database.
3. Insert records in a database using JSP.
4. Insert data from HTML Form in a database using JSP.
5. Delete Records from Database based on Criteria from HTML Form.
6. Retrieve data from a database using – JSP result sets.

- **Administratively Register a Database**

Java cannot talk to a database until, it is registered as a data source to your system. The easiest way to administratively identify or registrar the database to your system so your Java Server Page program can locate and communicate with it is to do the following:

- 1) Use MS Access to *create* a blank database in some directory such as D. (In my case, the database was saved as testCase001.mdb.) Make sure to *close* the database after it is created or you will get an invalid path *message* during the following steps.
- 2) Go to: Control panel > Admin tool > *ODBC* where you will identify the database as a so-called *data source*.
- 3) Under the *User DSN* tab, *un-highlight* any previously selected name and then click on the *Add* button.
- 4) On the window that then opens up, highlight *MS Access Driver* and click *Finish*.
- 5) On the ODBC Setup window that then opens, fill in the data source name. This is the name that you will use to refer to the database in your Java program such as Mimi.
- 6) Then click Select and *navigate* to the already created database in directory D. Suppose the file name is testCase001.mdb. After *highlighting* the named file, click OKs all the way back to the original window.

This completes the registration process. You could also use the create option to create the Access database from scratch. But the create setup option *destroys* any existing database copy. So, for an existing DB follow the procedure described above.

- **Connect a JSP to an Access Database**

We will now describe the Java code required to connect to the database although at this point we will not yet query it. To connect with database, the JSP program has to do several things:

1. Identify the source files for Java to handle SQL.
2. Load a software driver program that lets Java connect to the database.
3. Execute the driver to establish the connection.

A simplified JSP syntax required to do this follows is:

```
<%@ page import= "java.sql.*" %>
<%
    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection conn=null;
    conn = DriverManager.getConnection("jdbc:odbc:Mimi", "", "");
    out.println ("Database Connected");
%>
```

In this syntax, the so-called page directive at the top of the page allows the program to use methods and classes from the `java.sql.*` package that know how to handle SQL queries. The `Class.forName` method loads the driver for MS Access. Remember this is Java so the name is case-sensitive. If you misspell or misidentify the data source name, you'll get an error *message* "Data source name not found and no default driver specified". The `DriverManager.getConnection` method connects the program to the database identified by the data source *Mimi* allowing the program to make queries, inserts, selects, etc. Be careful of the punctuation too: those are colons (:) between each of the terms. If you use misspell or use dots, you'll get an error *message* about "No suitable driver". The `Connection` class has a different purpose. It contains the methods that let us work with SQL queries though this is not done in this example.

The `DriverManager` method creates a `Connection` object *conn* which will be used later when we make SQL queries. Thus,

1. In order to make queries, we'll need a `Statement` object.
2. In order to make a `Statement` object, we need a `Connection` object (*conn*).
3. In order to make a `Connection` object, we need to connect to the database.
4. In order to connect to the database, we need to load a driver that can make the connection.

The safer and more conventional code to do the same thing would include the database connection statements in a Java *try/catch* combination. This combination acts like a safety net. If the statements in the try section fail, then the Exceptions that they caused are caught in the catch section. The catch section can merely report the nature of the Exception or error (in the string *exc* shown here) or do more extensive backup processing, depending on the circumstances. The code looks like:

```
<%@ page import= "java.sql.*" %>
<%
    Connection conn=null;

    try
    {   Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        conn = DriverManager.getConnection("jdbc:odbc:Mimi", "",
        "");
    }
    catch (Exception exc)
    {   out.println(exc.toString() + "<br>"); }
```



```
        out.println ("Database Connected");  
        conn.close ();  
        out.println ("Database closed");  
    %>
```

We have also added another statement at the end that closes the connection to the data base (using the close () method of the connection object conn).

- **Insert Records in Database using JSP**

So far – other than connecting to the database and then closing the connection to the database – we have not had any tangible impact on the database. This section illustrates that we have actually communicated with the database by using simple SQL insert examples. In general, SQL inserts in a JSP environment would depend on data obtained from an HTML Form. But addressing that involves additional complications, so we shall stick here to simple fixed inserts with hardwired data. To do insert or indeed to use any kind of SQL queries we have to:

1. Create a Statement object - which has methods for handling SQL.
2. Define an SQL query - such as an insert query.
3. Execute the query.

Example *testCase002*. adds the following statements to insert an entry in the database:

```
Statement stm = conn.createStatement ();  
String      s = "INSERT INTO Managers VALUES ( 'Vivek' )";  
stm.executeUpdate (s);
```

The Connection object conn that was previously created has methods that allow us to in turn create a Statement object. The names for both these objects (in this case conn and stm) are of course just user-defined and could be anything we choose. The Statement object stm only has to be created once. The insert SQL is stored in a Java String variable. The table managers we created in the database (off-line) has a single text attribute (managerName) which is not indicated in the query. The value of a text attribute must be enclosed in single quotes. Finally, the Statement object's executeUpdate method is used to execute the insert query. If you run the testCase002.jsp example and check the managers table contents afterwards, you will see that the new record has been added to the table.

Generally, these statements are executed under try/catch control. Thus, the executeUpdate (s) method is handled using:

```
try                                {   stm.executeUpdate(s);   }  
catch (Exception exc) {   out.println(exc.toString());   }
```

as in *testCase002.jsp*. If the update fails because there is an error in the query string submitted (in s), then the catch clause takes over. The error or exception is set by the system in exc. The body of the catch can output the error message as shown. It could also do whatever other processing the programmer deemed appropriate.

- **Inserting Data from an HTML Form in Database using JSP**

The JSP acquires the data to be inserted into a database from an HTML Form. This interaction involves several elements:

1. An HTML Form with named input fields.

2. JSP statements that access the data sent to the server by the form.
3. Construction of an insert query based on this data by the JSP program.
4. Execution of the query by the JSP program.

To demonstrate this process, we have to define the HTML page that will be accessed by the JSP program. We will use testCase000.html which has three input fields: mName, mAge, and mSalary. It identifies the requested JSP program as testCase003.jsp. For simplicity, initially assume the Access table has a single text attribute whose value is picked up from the Form. The example testCase003.jsp must acquire the Form data, prep it for the query, build the query, then execute it. It also sends a copy of the query to the browser so you can see the constructed query. The relevant statements are:

```
String name = request.getParameter ("mName");
name = "" + name + " ";
String s = "INSERT INTO Managers VALUES (" ;
s += name ;
s += ")" ;
stm.executeUpdate (s);
out.println ("<br>" + s + "<br>");
```

In the above code, the first statement acquires the form data, the second preps it for the database insert by attaching single quotes fore and aft, the next three statements construct the SQL insert query, the next statement executes the query, and the final statement displays it for review on the browser.

- **Delete Records from Database based on Criteria from HTML Form**

We can illustrate the application of an SQL delete query using the same HTML Form. The difference between the insert and the delete is that the delete has a where clause that determines which records are deleted from the database. Thus, to delete a record where the attribute *name* has the text value *Rahul*, the fixed SQL is:

```
String s = "Delete From Managers Where name = '
Rahul' "
```

- **Retrieve Data from Database – JSP ResultSets**

Retrieving data from a database is slightly more complicated than inserting or deleting data. The retrieved data has to be put someplace and in Java that place is called a ResultSet. A ResultSet object, which is essentially a table of the returned results as done for any SQL Select, is returned by an executeQuery method, rather than the executeUpdate method used for inserts and deletes. The steps involved in a select retrieval are:

1. Construct a desired Select query as a Java string.
2. Execute the executeQuery method, saving the results in a ResultSet object r.
3. Process the ResultSet r using two of its methods which are used in tandem:
 - a) r.next () method moves a pointer to the next row of the retrieved table.

b) `r.getString (attribute-name)` method extracts the given attribute value from the currently pointed to row of the table.

A simple example of a Select is given in `testCase04` where a fixed query is defined. The relevant code is:

```
String      s      = "SELECT * FROM Managers";
ResultSet   r      = stm.executeQuery(s);

while ( r.next( ) )
{
    out.print ("<br>Name: " + r.getString ("name") );
    out.println("    Age :  " + r.getString ("age" ) );
}
```

The query definition itself is the usual SQL Select. The results are retrieved from the database using `stm.executeQuery (s)`. The while loop (because of its repeated invocation of `r.next()` advances through the rows of the table which was returned in the `ResultSet r`. If this table is empty, the while test fails immediately and exits. Otherwise, it points to the row currently available. The values of the attributes in that row are then accessed using the `getString` method which returns the value of the attribute "name" or "age". If you refer to an attribute that is not there or misspell, you'll get an error *message* "Column not found". In this case, we have merely output the retrieved data to the HTML page being constructed by the JSP. Once, the whole table has been scanned, `r.next()` returns False and the while terminates. The entire process can be included in a try/catch combination for safety.

Check Your Progress 1

Fill in the blanks:

- 1) method connects the program to the database identified by the data source.
- 2) method is used to execute the insert query.
- 3) method is used to execute the select query.
- 4) In Java the result of select query is placed in

4.3 INCLUDING FILES AND APPLETS IN JSP DOCUMENTS

Now in this section, we learn how to include external pieces into a JSP document. JSP has three main capabilities for including external pieces into a JSP document.

i) `jsp:include` action

This includes generated page, not JSP code. It cannot access environment of main page in included code.

ii) include directive

Include directive includes dynamic page, i.e., JSP code. It is powerful, but poses maintenance challenges.

iii) jsp:plugin action

This inserts applets that use the Java plug-in. It increases client side role in dialog.

Now, we will discuss in details about this capability of JSP.

• Including Files at Page Translation Time

You can include a file with JSP at the page translation time. In this case file will be included in the main JSP document at the time the document is translated into a servlet (which is typically the first time it is accessed).

To include file in this way, the syntax is:

```
<% @ include file= "Relative URL" %>
```

There are two consequences of the fact that the included file is inserted at page translation time, not at request time as with jsp: include.

First, you include the actual file itself, unlike with jsp:include , where the server runs the page and inserts its output. This approach means that the included file can contain JSP constructs (such as field or method declarations) that affect the main page as a whole.

Second, if the included file changes, all the JSP files that use it need to be updated.

• Including Files at Request Time

As you studied, the include directive provides the facility to include documents that contain JSP code into different pages. But this directive requires you to update the modification date of the page whenever the included file changes, which is a significant inconvenience.

Now, we will discuss about the **jsp:include** action. It includes files at the time of the client request and thus does not require you to update the main file when an included file changes. On the other hand, as the page has already been translated into a servlet at request time, thus the included files cannot contain JSP.

Although the included files cannot contain JSP, they can be the result of resources that use JSP to create the output. That is, the URL that refers to the included resource is interpreted in the normal manner by the server and thus can be a servlet or JSP page. This is precisely the behaviour of the include method of the RequestDispatcher class, which is what servlets use if they want to do this type of file inclusion. The jsp:include element has two required attributes (as shown in the sample below), these elements are:

- i) **page** : It refers to a relative URL referencing the file to be included
- ii) **flush**: This must have the value true.

```
<jsp:include page="Relative URL" flush="true" />
```

Although you typically include HTML or plain text documents, there is no requirement that the included files have any particular file extension.

- **Including Applets for the Java Plug-In**

To include ordinary applets with JSP, you don't need any special syntax; you need to just use the normal HTML APPLET tag. But, these applets must use JDK 1.1 or JDK 1.02, because neither Netscape 4.x nor Internet Explorer 5.x support the Java 2 platform (i.e., JDK 1.2). This lack of support imposes several restrictions on applets these are:

- In order to use Swing, you must send the Swing files over the network. This process is time consuming and fails in Internet Explorer 3 and Netscape 3.x and 4.01-4.05 (which only support JDK 1.02), since Swing depends on JDK 1.1.
- You cannot use Java 2D.
- You cannot use the Java 2 collections package.
- Your code runs more slowly, since most compilers for the Java 2 platform are significantly improved over their 1.1 predecessors.

To solve this problem, Sun developed a browser plug-in for Netscape and Internet Explorer that lets you use the Java 2 platform for applets in a variety of browsers. It is a reasonable alternative for fast corporate intranets, especially since applets can automatically prompt browsers that lack the plug-in to download it. But, since, the plug-in is quite large (several megabytes), it is not reasonable to expect users on the WWW at large to download and install it just to run your applets. As well as, the normal APPLET tag will not work with the plug-in, since browsers are specifically designed to use only their built-in virtual machine when they see APPLET. Instead, you have to use a long and messy OBJECT tag for Internet Explorer and an equally long EMBED tag for Netscape. Furthermore, since, you typically don't know which browser type will be accessing your page, you have to either include both OBJECT and EMBED (placing the EMBED within the COMMENT section of OBJECT) or identify the browser type at the time of the request and conditionally build the right tag. This process is straightforward but tedious and time consuming.

The **jsp:plugin** element instructs the server to build a tag appropriate for applets that use the plug-in. Servers are permitted some leeway in exactly how they implement this support, but most simply include both OBJECT and EMBED.

The simplest way to use jsp:plugin is to supply four attributes: type, code, width, and height. You supply a value of applet for the type attribute and use the other three attributes in exactly the same way as with the APPLET element, with two exceptions, i.e., the attribute names are case sensitive and single or double quotes are always required around the attribute values.

For example, you could replace

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

with

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

The jsp:plugin element has a number of other optional attributes. A list of these attributes is:

- **type:**

For applets, this attribute should have a value of applet. However, the Java Plug-In also permits you to embed JavaBeans elements in Web pages. Use a value of bean in such a case.

- **code :**

This attribute is used identically to the CODE attribute of APPLET, specifying the top-level applet class file that extends Applet or JApplet. Just remember that the name code must be lower case with jsp:plugin (since, it follows XML syntax), whereas with APPLET, case did not matter (since, HTML attribute names are never case sensitive).

- **width :**

This attribute is used identically to the WIDTH attribute of APPLET, specifying the width in pixels to be reserved for the applet. Just remember that you must enclose the value in single or double quotes.

- **height :**

This attribute is used identically to the HEIGHT attribute of APPLET, specifying the height in pixels to be reserved for the applet. Just remember that you must enclose the value in single or double quotes.

- **codebase :**

This attribute is used identically to the CODEBASE attribute of APPLET, specifying the base directory for the applets. The code attribute is interpreted relative to this directory. As with the APPLET element, if you omit this attribute, the directory of the current page is used as the default. In the case of JSP, this default location is the directory where the original JSP file resided, not the system-specific location of the servlet that results from the JSP file.

- **align:**

This attribute is used identically to the ALIGN attribute of APPLET and IMG, specifying the alignment of the applet within the web page. Legal values are left, right, top, bottom, and middle. With jsp:plugin, don't forget to include these values in single or double quotes, even though quotes are optional for APPLET and IMG.

- **hspace :**

This attribute is used identically to the HSPACE attribute of APPLET, specifying empty space in pixels reserved on the left and right of the applet. Just remember that you must enclose the value in single or double quotes.

- **vspace :**

This attribute is used identically to the VSPACE attribute of APPLET, specifying empty space in pixels reserved on the top and bottom of the applet. Just remember that you must enclose the value in single or double quotes.

- **archive :**

This attribute is used identically to the ARCHIVE attribute of APPLET, specifying a JAR file from which classes and images should be loaded.

- **name :**

This attribute is used identically to the NAME attribute of APPLET, specifying a name to use for inter-applet communication or for identifying the applet to scripting languages like JavaScript.

- **title :**

This attribute is used identically to the very rarely used TITLE attribute of APPLET (and virtually all other HTML elements in HTML 4.0), specifying a title that could be used for a tool-tip or for indexing.

- **jreversion :**

This attribute identifies the version of the Java Runtime Environment (JRE) that is required. The default is 1.1.

- **iepluginurl :**

This attribute designates a URL from which the plug-in for Internet Explorer can be downloaded. Users who don't already have the plug-in installed will be prompted to download it from this location. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.

- **nspluginurl :**

This attribute designates a URL from which the plug-in for Netscape can be downloaded. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.

The jsp:param and jsp:params Elements

The jsp:param element is used with jsp:plugin in a manner similar to the way that PARAM is used with APPLET, specifying a name and value that are accessed from within the applet by getParameter. There are two main differences between jsp:param and param with applet, these are :

First, since jsp:param follows XML syntax, attribute names must be lower case, attribute values must be enclosed in single or double quotes, and the element must end with />, not just >.

Second, all jsp:param entries must be enclosed within a jsp:params element. So, for example, you would replace

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

with

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

The jsp:fallback Element

The jsp:fallback element provides alternative text to browsers that do not support OBJECT or EMBED. You use this element in almost the same way as you would use alternative text placed within an APPLET element.

So, for example, you would replace

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

```
    <B>Error: this example requires Java.</B>
```

```
</APPLET>
```

with

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

```
    <jsp:fallback>
```

```
        <B>Error: this example requires Java.</B>
```

```
    </jsp:fallback>
```

```
</jsp:plugin>
```

4.4 INTEGRATING SERVLET AND JSP

As you have seen, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate GIF images on-the-fly, and perform many other tasks flexibly and efficiently. Therefore servlets are great when your application requires a lot of real programming to accomplish its task.

But, generating HTML with servlets can be tedious and can yield a result that is hard to modify. That's where JSP comes in; it let's you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your web content developers to work on your JSP documents.

Now, let us discuss the limitation of JSP. As you know, the assumption behind a JSP document is that it provides a single overall presentation. But what if you want to give totally different results depending on the data that you receive? Beans and custom tags, although extremely powerful and flexible, but they don't overcome the limitation that the JSP page defines a relatively fixed top-level page appearance. The solution is to use both servlets and Java Server Pages. If you have a complicated application that may require several substantially different presentations, a servlet can handle the initial request, partially process the data, set up beans, then forward the results to one of a number of different JSP pages, depending on the circumstances.

In early JSP specifications, this approach was known as the model 2 approach to JSP. Rather than completely forwarding the request, the servlet can generate part of the output itself, then include the output of one or more JSP pages to obtain the final result.

4.4.1 Forwarding Requests

The key to letting servlets forward requests or include external content is to use a requestDispatcher. You obtain a RequestDispatcher by calling the getRequestDispatcher method of ServletContext, supplying a URL relative to the server root.

For example, to obtain a `RequestDispatcher` associated with `http://yourhost/presentations/presentation1.jsp`, you would do the following:

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

Once, you have a `RequestDispatcher`, you use `forward` to completely transfer control to the associated URL and use `include` to output the associated URL's content. In both cases, you supply the `HttpServletRequest` and `HttpServletResponse` as arguments. Both methods throw `Servlet-Exception` and `IOException`. For example, the example 4.1 shows a portion of a servlet that forwards the request to one of three different JSP pages, depending on the value of the operation parameter. To avoid repeating the `getRequestDispatcher` call, I use a utility method called `gotoPage` that takes the URL, the `HttpServletRequest` and the `HttpServletResponse`; gets a `RequestDispatcher`; and then calls `forward` on it.

Example 4.1: Request Forwarding Example

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

```
    String operation = request.getParameter("operation");  
    if (operation == null) {  
        operation = "unknown";  
    }  
    if (operation.equals("operation1")) {  
        gotoPage("/operations/presentation1.jsp", request, response);  
    }  
    else if (operation.equals("operation2")) {  
        gotoPage("/operations/presentation2.jsp", request, response);  
    }  
    else {  
        gotoPage("/operations/unknownRequestHandler.jsp", request,  
            response);  
    }  
}
```

```
private void gotoPage(String address, HttpServletRequest request,  
    HttpServletResponse response)  
throws ServletException, IOException {  
    RequestDispatcher dispatcher  
=getServletContext().getRequestDispatcher(address);  
    dispatcher.forward(request, response);  
}
```

Using Static Resources

In most cases, you forward requests to a JSP page or another servlet. In some cases, however, you might want to send the request to a static HTML page. In an e-commerce site, for example, requests that indicate that the user does not have a valid account name might be forwarded to an account application page that uses HTML forms together the requisite information. With GET requests, forwarding requests to a static HTML page is perfectly legal and requires no special syntax; just supply the address of the HTML page as the argument to `getRequestDispatcher`. However, since, forwarded requests use the same request method as the original request, POST requests cannot be forwarded to normal HTML pages. The solution to this problem is to simply rename the HTML page to have a `.jsp` extension. Renaming `somefile.html` to `somefile.jsp` does not change its output for GET requests, but `somefile.html` cannot handle POST requests, whereas `somefile.jsp` gives an identical response for both GET and POST.

Supplying Information to the Destination Pages

To forward the request to a JSP page, a servlet merely needs to obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletContext`, then call forward on the result, supplying the `Http- ServletRequest` and `HttpServletResponse` as arguments. That's fine as far as it goes, but this approach requires the destination page to read the information it needs out of the `HttpServletRequest`. There are two reasons why it might not be a good idea to have the destination page look up and process all the data itself. *First*, complicated programming is easier in a servlet than in a JSP page. *Second*, multiple JSP pages may require the same data, so it would be wasteful for each JSP page to have to set up the same data. A better approach is for, the original servlet to set up the information that the destination pages need, then store it somewhere that the destination pages can easily access. There are two main places for the servlet to store the data that the JSP pages will use: in the `HttpServletRequest` and as a bean in the location specific to the scope attribute of `jsp:useBean`.

The originating servlet would store arbitrary objects in the `HttpServletRequest` by using

```
request.setAttribute("key1", value1);
```

The destination page would access the value by using a JSP scripting element to call

```
Type1 value1 = (Type1)request.getAttribute("key1");
```

For complex values, an even better approach is to represent the value as a bean and store it in the location used by `jsp:useBean` for shared beans. For example, a scope of application means that the value is stored in the `ServletContext`, and `ServletContext` uses `setAttribute` to store values. Thus, to make a bean accessible to all servlets or JSP pages in the server or Web application, the originating servlet would do the following:

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

The destination JSP page would normally access the previously stored value by using `jsp:useBean` as follows:

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

Alternatively, the destination page could use a scripting element to explicitly call `application.getAttribute("key1")` and cast the result to `Type1`. For a servlet to make

data specific to a user session rather than globally accessible, the servlet would store the value in the HttpSession in the normal manner, as below:

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

The destination page would then access the value by means of

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

The Servlet 2.2 specification adds a third way to send data to the destination page when using GET requests: simply append the query data to the URL. For example,

```
String address = "/path/resource.jsp?newParam=value";
RequestDispatcher dispatcher =getServletContext().getRequestDispatcher(address);
dispatcher.forward(request, response);
```

This technique results in an additional request parameter of newParam (with a value of value) being added to whatever request parameters already existed. The new parameter is added to the beginning of the query data so that it will replace existing values if the destination page uses getParameter (use the first occurrence of the named parameter) rather than get- ParameterValues (use all occurrences of the named parameter).

Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to arbitrary locations on the same server, the process is quite different from that of using the sendRedirect method of HttpServletResponse.

First, sendRedirect requires the client to reconnect to the new resource, whereas the forward method of RequestDispatcher is handled completely on the server.

Second, sendRedirect does not automatically preserve all of the request data; forward does.

Third, sendRedirect results in a different final URL, whereas with forward, the URL of the original servlet is maintained.

This final point means that, if the destination page uses relative URLs for images or style sheets, it needs to make them relative to the server root, not to the destination page's actual location. For example, consider the following style sheet entry:

```
<LINK REL=STYLESHEET HREF= "my-styles.css" TYPE= "text/css">
```

If the JSP page containing this entry is accessed by means of a forwarded request, my-styles.css will be interpreted relative to the URL of the originating servlet, not relative to the JSP page itself, almost certainly resulting in an error. The solution is to give the full server path to the style sheet file, as follows:

```
<LINK REL=STYLESHEET HREF= "/path/my-styles.css" TYPE= "text/css">
```

The same approach is required for addresses used in and .

4.4.2 Including Static or Dynamic Content

If a servlet uses the forward method of RequestDispatcher, it cannot actually send any output to the client—it must leave that entirely to the destination page. If the servlet wants to generate some of the content itself but use a JSP page or static HTML

document for other parts of the result, the servlet can use the include method of `RequestDispatcher` instead. The process is very similar to that for forwarding requests: Call the `getRequestDispatcher` method of `ServletContext` with an address relative to the server root, then call `include` with the `HttpServletRequest` and `HttpServletResponse`.

The two differences when `include` is used are that you can send content to the browser before making the call and that control is returned to the servlet after the `include` call finishes. Although the included pages (servlets, JSP pages, or even static HTML) can send output to the client, they should not try to set HTTP response headers. Here is an example:

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

The `include` method has many of the same features as the `forward` method. If the original method uses POST, so does the forwarded request. Whatever request data was associated with the original request is also associated with the auxiliary request, and you can add new parameters (in version 2.2 only) by appending them to the URL supplied to `getRequestDispatcher`. Version 2.2 also supports the ability to get a `RequestDispatcher` by name (`getNamedDispatcher`) or by using a relative URL (use the `getRequestDispatcher` method of the `HttpServletRequest`). However, `include` does one thing that `forward` does not: it automatically sets up attributes in the `HttpServletRequest` object that describe the original request path in case, the included servlet or JSP page needs that information. These attributes, available to the included resource by calling `getAttribute` on the `HttpServletRequest`, are listed below:

- `javax.servlet.include.request_uri`
- `javax.servlet.include.context_path`
- `javax.servlet.include.servlet_path`
- `javax.servlet.include.path_info`
- `javax.servlet.include.query_string`

Note that this type of file inclusion is not the same as the nonstandard servlet chaining supported as an extension by several early servlet engines. With servlet chaining, each servlet in a series of requests can see (and modify) the output of the servlet before it. With the `include` method of `RequestDispatcher`, the included resource cannot see the output generated by the original servlet. In fact, there is no standard construct in the servlet specification that reproduces the behaviour of servlet chaining.

Also note that this type of file inclusion differs from that supported by the JSP `include` directive. There, the actual source code of JSP files was included in the page by use of the `include` directive, whereas the `include` method of `RequestDispatcher` just includes the result of the specified resource. On the other hand, the `jsp:include` action has behavior similar to that of the `include` method, except that `jsp:include` is available only from JSP pages, not from servlets.

4.4.3 Forwarding Requests from JSP Pages

The most common request forwarding scenario is that the request first comes to a servlet and the servlet forwards the request to a JSP page. The reason is a servlet usually handles the original request is that checking request parameters and setting up beans requires a lot of programming, and it is more convenient to do this programming in a servlet than in a JSP document. The reason that the destination page is usually a JSP document is that JSP simplifies the process of creating the HTML content.

However, just because this is the usual approach doesn't mean that it is the only way of doing things. It is certainly possible for the destination page to be a servlet. Similarly, it is quite possible for a JSP page to forward requests elsewhere. For example, a request might go to a JSP page that normally presents results of a certain type and that forwards the request elsewhere only when it receives unexpected values. Sending requests to servlets instead of JSP pages requires no changes whatsoever in the use of the RequestDispatcher. However, there is special syntactic support for forwarding requests from JSP pages. In JSP, the `jsp:forward` action is simpler and easier to use than wrapping up Request-Dispatcher code in a scriptlet. This action takes the following form:

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

The page attribute is allowed to contain JSP expressions so that the destination can be computed at request time.

For example, the following sends about half the visitors to `http://host/examples/page1.jsp` and the others to `http://host/examples/page2.jsp`.

```
<%  
    String destination;  
    if (Math.random() > 0.5) {  
        destination = "/examples/page1.jsp";  
    }  
    else {  
        destination = "/examples/page2.jsp";  
    }  
%>  
<jsp:forward page= "<%= destination %>" />
```

Check Your Progress 2

Give right choice for the following:

- 1) ----- includes generated page, not JSP code.
 - a) include directive
 - b) `jsp:include` action
 - c) `jsp:plugin` action
- 2) ----- attribute of `jsp:plugin` designates a URL from which the plug-in for Internet Explorer can be downloaded.
 - a) Codebase
 - b) Archive
 - c) `Iepluginurl`
 - d) `Nspluginurl`

- 3) A RequestDispatcher can obtain by calling the ----- method of -----, supplying a URL relative to the server root.
- ServletContext, getRequestDispatcher
 - HttpServletRequest, getRequestDispatcher
 - getRequestDispatcher, HttpServletRequest
 - getRequestDispatcher, ServletContext

Explain following questions in brief:

- 4) Write a note on jsp:fallback Element.

.....

.....

.....

4.5 SUMMARY

In this unit, we first studied about database handling in JSP. In this topic we saw that to communicate with a database through JSP, first that database is needed to be registered on your system. Then JSP makes a connection with the database. For this purpose DriverManager.getConnection method is used. This method connects the program to the database identified by the data source by creating a connection object.

This object is used to make SQL queries. For making SQL queries, a statement object is used. This statement object is created by using connection object's createStatement () method. Finally statement object's executeUpdate method is used to execute the insert and delete query and executeQuery method is used to execute the SQL select query.

Next, we studied about how to include files and applets in JSP documents. In this, we studied that JSP has three main capabilities for including external pieces into a JSP document. These are:

- *jsp:include action*: It is used to include generated pages, not JSP code.
- *include directive*: It includes dynamic page, i.e., JSP code.
- *jsp:plugin action*: This inserts applets that use the Java plug-in.

Finally we studied about integrating servlet and JSP. The key to let servlets forward requests or include external content is to use a requestDispatcher. This RequestDispatcher can be obtained by calling the getRequestDispatcher method of ServletContext, supplying a URL relative to the server root. But if you want to forward the request from JSP, jsp:forward action is used.

4.6 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) DriverManager.getConnection
- 2) Statement object's executeUpdate
- 3) Statement object's executeQuery
- 4) ResultSet

Check Your Progress 2

- 1) b
- 2) c
- 3) d
- 4) The `jsp:fallback` element provides alternative text to browsers that do not support OBJECT or EMBED. This element can be used in almost the same way as alternative text is placed within an APPLET element.

For example,

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

```
    <B>Error: this example requires Java.</B>
```

```
</APPLET>
```

can be replaced with

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

```
    <jsp:fallback>
```

```
        <B>Error: this example requires Java.</B>
```

```
    </jsp:fallback>
```

```
</jsp:plugin>
```

4.7 FURTHER READINGS/REFERENCES

- Phil Hanna, *JSP: The Complete Reference*
- Hall, Marty, *Core Servlets and JavaServer Pages*
- Annunziato Jose & Fesler, Stephanie, *Teach Yourself JavaServer Pages in 24 Hours*,
- Bruce W. Perry, *Java Servlet & JSP, Cookbook* (Paperback),