UNIT 1 SERVLET PROGRAMMING

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 4.0Unzip 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-b5SET JAVA_HOME=C:\jdk1.3Now 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 by 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.
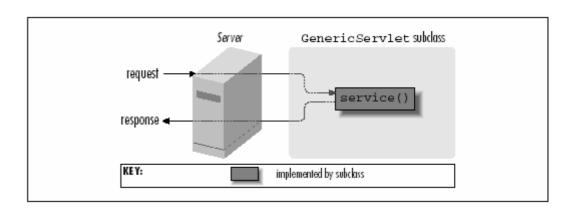

Figure 1: A generic Servlet handling a request

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.
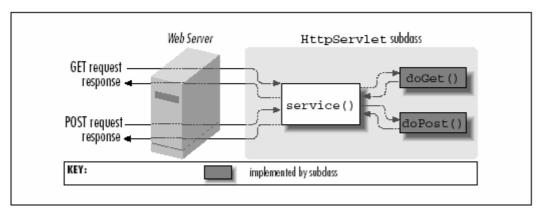
Figure 2: An Http Servlet handling get and post Requests

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*:
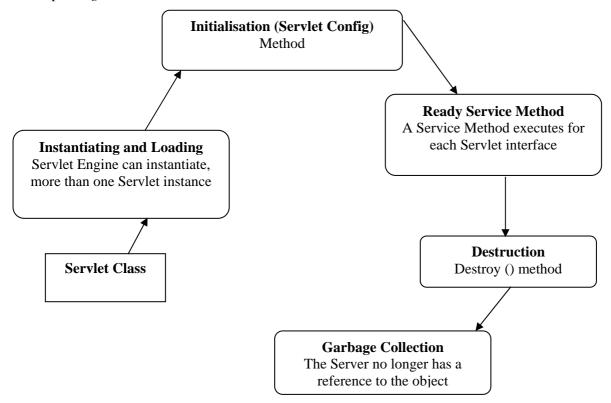

Figure 3: The Servlet Life Cycle

9

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:

```java
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 HTTPRequest 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 HttpServlet-Request:
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.

```java
/* 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.

Colloboration 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 colloboration 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.

5)
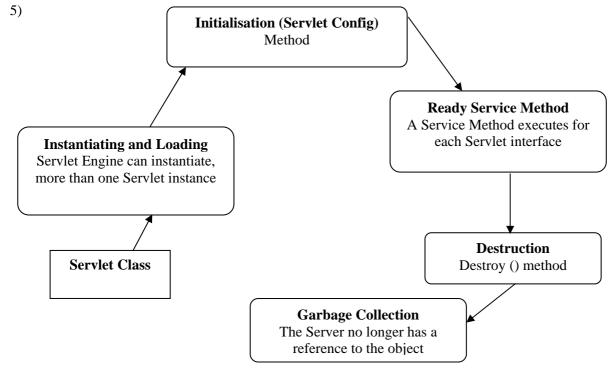
Figure 4: The servlet life cycle

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 HttpServlet Response 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) Colloboration 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

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 Micorsoft'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

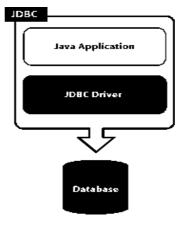*Figure 1* shows the components of the JDBC model

Figure 1: Components of java database connectivity

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

JAVA Database
Connectivity

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. DriverPropertymfo
- 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** object are used for executing SQL queries.

Different types of JDBC SQL Statements

a) **java.sql.Statement :** Top most interface which provides basic methods useful for executing SELECT, INSERT, UPDATE and DELETE SQL statements.

b) **java.sql.PreparedStatement :** An enhanced verion 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 vender 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.
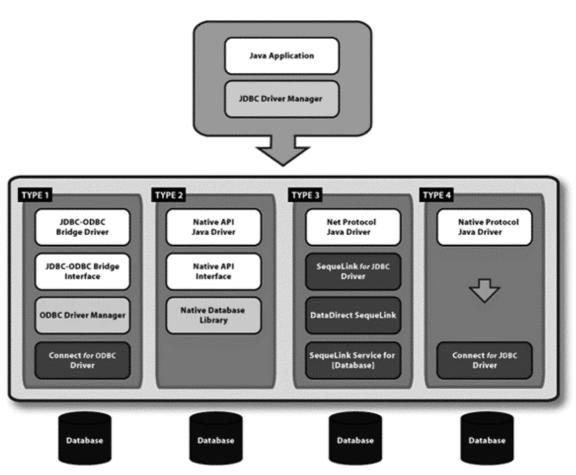
Figure 2: Comparison of different JDBC drivers

🗇 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 drviers 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 specificationjdbc.drivers =
sun.jdbc.odbc.JdbcOdbcDriverdatabase.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 RecordSet 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.our.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 vender 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)   OBDC (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.our.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 make 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

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 pag*e* 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

Figure1. 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.
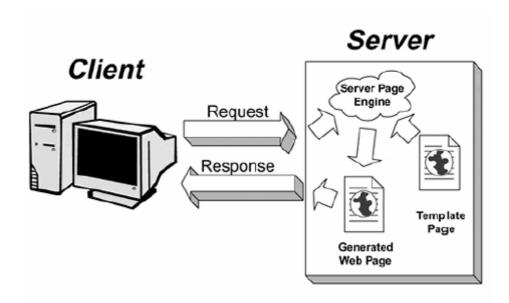

Figure 1: Server page

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	**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
   Wed Aug 17 17:10:05 PST 2006
 </BODY>
</HTML>
```

```
<! 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>
```

Scriplets

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 %>**

• **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() %>   **Expression element**

   </BODY>

</HTML>

3.6   JSP SCRIPLETS

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>
```

Scriplets

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" ethods. 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 servers 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	javax.servlet.jsp.JspWriter	The output stream.
request	Subtype of javax.servlet.ServletRequest	The request triggering the execution of the JSP page.
response	Subtype of javax.servlet.ServletResponse	The response to be returned to the client. Not typically used by JSP page authors.
pageContext	javax.servlet.jsp.PageContext	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	javax.servlet.http.HttpSession	The session object for the client..
application	javax.servlet.ServletContext	The context for the JSP page's servlet and any Web components contained in the same application.
config	javax.servlet.ServletConfig	Initialization information for the JSP page's servlet.
page	java.lang.Object	The instance of the JSP page's servlet processing the current request. Not typically used by JSP page authors.
exception	java.lang.Throwable	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.
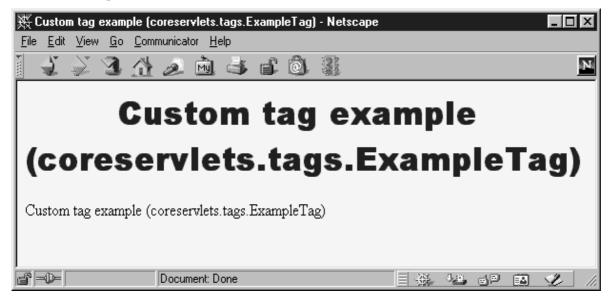
Example 3.6 SimpleExample.jsp

```
<!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>
```

Result of Example 3.6 is shown in *Figure 3.*

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/REFRENCES

- 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

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 execute**Query** 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 ofAPPLET, 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 togather 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 HttpServlet-Request 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 <IMG SRC=...> and
<A HREF=...>.

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 Http-ServletRequest 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 Http-ServletRequest, 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.

   a)   ServletContext, getRequestDispatcher

   b)   HttpServletRequest, getRequestDispatcher

   c)   getRequestDispatcher, HttpServletRequest

   d)   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/REFRENCES

- 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 1   INTRODUCTION TO JAVA BEANS

1.0   INTRODUCTION

Software has been evolving at a tremendous speed since its inception. It has gone through various phases of development from low level language implementation to high level language implementation to non procedural program models. The designers always make efforts to make programming easy for users and near to the real world implementation. Various programming models were made public like procedural programming, modular programming and non procedural programming model. Apart from these models reusable component programming model was introduced to put programmers at ease and to utilise the reusability as its best. Reusable Component model specifications were adopted by different vendors and they came with their own component model solutions.

1.1   OBJECTIVES

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

- define what is Java Beans;

- list EJB types;

- discuss about Java Architecture;

- make differences between different types of Beans;

- discuss the life cycle of a Beans, and

- describe the use of Message Driven Beans.

1.2   WHAT IS JAVABEAN?

JavaBeans are software component models. A JavaBean is a general-purpose component model. A Java Bean is a reusable software component that can be visually manipulated in builder tools. Their primary goal of a JavaBean is **WORA** (Write Once Run Anywhere). JavaBeans should adhere to portability, reusability and interoperability.

JavaBeans will look a plain Java class written with getters and setters methods. It's logical to wonder: "What is the difference between a Java Bean and an instance of a normal Java class?" What differentiates Beans from typical Java classes is **introspection**. Tools that recognize predefined patterns in method signatures and class definitions can "look inside" a Bean to determine its properties and behaviour.

A Bean's state can be manipulated at the time it is being assembled as a part within a larger application. The application assembly is referred to as **design time** in contrast to **run time**. For this scheme to work, method signatures within Beans must follow a certain pattern, for introspection tools to recognise how Beans can be manipulated, both at design time, and run time.

In effect, Beans publish their attributes and behaviours through special method signature patterns that are recognised by beans-aware application construction tools. However, you need not have one of these construction tools in order to build or test your beans. Pattern signatures are designed to be easily recognised by human readers as well as builder tools. One of the first things you'll learn when building beans is how to recognise and construct methods that adhere to these patterns.

Not all useful software modules should be Beans. Beans are best suited to software components intended to be visually manipulated within builder tools. Some functionality, however, is still best provided through a programatic (textual) interface, rather than a visual manipulation interface. For example, an SQL, or JDBC API would probably be better suited to packaging through a class library, rather than a Bean.

1.3   JAVABEAN CONCEPTS

JavaBeans is a complete component model. It supports the standard component architecture features of properties, events, methods, and persistence. In addition, JavaBeans provides support for introspection (to allow automatic analysis of a JavaBeans component) and customisation (to make it easy to configure a JavaBeans component). Typical unifying features that distinguish a Bean are:

- **Introspection:** Builder tools discover a Bean's features (ie its properties, methods, and events) by a process known as INTROSPECTION. Beans supports introspection in two ways:

1) **Low Level Introspection (Reflection) + Intermediate Level Introspection (Design Pattern):** Low Level Introspection is accomplished using **java.lang.reflect** package API. This API allows Java Objects to discover information about public fields, constructors, methods and events of loaded classes during program execution i.e., at Run-Time. Intermediate Level Introspection (Design Pattern) is accomplished using **Design Patterns**. Design Patterns are bean features naming conventions to which one has to adhere while writing code for Beans. java.beans.Introspector class examines Beans for these design patterns to discover Bean features. The Introspector class relies on the core reflection API. There are two types of methods namely, **accessor methods** and **interface methods**. Accessor methods are used on properties and are of

two sub-types (namely **getter methods and setters methods**). Interface methods are often used to support event handling.

2) **Higest Level or Explicit Introspection (BeanInfo):** It is accomplished by explicitly providing property, method, and event information with a related Bean Information Class. A Bean information class implements the BeanInfo interface. A BeanInfo class explicitly lists those Bean features that are to be exposed to the application builder tools. The Introspector recognises BeanInfo classes by their name. The name of a BeanInfo class is the name of the bean class followed by BeanInfo word e.g., for a bean named "Gizmo" the BeanInfo name would be "GizmoBeanInfo".

- **Properties:** Are a Bean's appearance and behaviour characteristics that can be changed at design time.

- **Customisation:** Beans expose properties so they can be customised during the design time.

- **Events:** Enables Beans to communicate and connect to each other.

- **Persistence:** The capability of permanently stored property changes is known as Persistence. Beans can save and restore their state i.e., they need to be persistent. It enables developers to customise Beans in an app builder, and then retrieve those Beans, with customised features intact, for future use. JavaBeans uses **Java Object Serialisation** to support persistence. **Serialisation** is the process of writing the current state of an object to a stream. To serialise an object, the class must implement either java.io.Serialisable or java.io.Externalisable interface. Beans that implement Serialisable are automatically saved and beans that implements Externalisable are responsible for saving themselves. The transient and static variables are not serialised i.e., these type of variables are not stored.

Beans can also be used just like any other Java class, manually (i.e., by hand programming), due to the basic Bean property, "Persistence". Following are the two ways:

- Simply instantiate the Bean class just like any other class.

- If you have a customised Bean (through some graphic tool) saved into a serialised file (say mybean.ser file), then use the following to create an instance of the Customised Bean class...

```
try {
        MyBean mybean = (MyBean)
        Beans.instantiate(null, "mybean");
    } catch (Exception e) {
    }
```

- **Connecting Events:** Beans, being primarily GUI components, generate and respond to events. The bean generating the event is referred to as *event source* and the bean listening for (and handling) the event is referred to as the *event listener*.

- **Bean Properties:** Bean properties can be categorised as follows...

  1) **Simple Property** are basic, independent, individual prperties like width, height, and colour.

  2) **Indexed Property** is a property that can take on an array of values.

  3) **Bound Property** is a property that alerts other objects when its value changes.

4) **Constrained Property** differs from Bound Property in that it notifies other objects of an impending change. Constrained properties give the notified objects the power to veto a property change.

Accessor Methods

1. **Simple Property :**

   If, a bean has a property named **foo** of type **fooType** that can be read and written, it should have the following accessor methods:

   ```
   public fooType getFoo( ) { return foo; }
   public void setFoo(fooType fooValue) {
         foo = fooValue; ...
   }
   ```

   If a property is boolean, getter methods are written using **is** instead of **get** eg isFoo( ).

2. **Indexed Property :**

   ```
   public widgetType getWidget(int index)
   public widgetType[] getWidget( )
   public void setWidget(int index, widgetType widgetValue)
   public void setWidget(widgetType[] widgetValues)
   ```

3. **Bound Property :**

   Getter and setter methods for bound propery are as described above based on whether it is simple or indexed. Bound properties require certain objects to be notified when they change. The change notification is accomplished through the generation of a **PropertyChangeEvent** (defined in java.beans). Objects that want to be notified of a property change to a bound property must register as listeners. Accordingly, the bean that's implementing the bound property supplies methods of the form:

   ```
   public void addPropertyChangeListener(ropertyChangeListener l)
   public void removePropertyChangeListener(PropertyChangeListener l)
   ```

   The preceding listener registeration methods do not identify specific bound properties. To register listeners for the PropertyChangeEvent of a specific property, the following methods must be provided:

   ```
   public void addPropertyNameListener(PropertyChangeListener l)
   public void emovePropertyNameListener(PropertyChangeListener l)
   ```

   In the preceding methods, **PropertyName** is replaced by the name of the bound property.

   Objects that implement the PropertyChangeListener interface must implement the **PropertyChange( )** method. This method is invoked by the bean for all registered listeners to inform them of a property change.

4. **Constrained Property :**

   The previously discussed methods used with simple and indexed properties also apply

to the constrained properties. In addition, the following event registeration methods
provided:

```
public void addVetoableChangeListener(VetoableChangeListener l)
public void removeVetoableChangeListener(VetoableChangeListener l)
public void addPropertyNameListener(VetoableChangeListener l)
public void removePropertyNameListener(VetoableChangeListener l)
```

Objects that implement the VetoableChangeListener interface must implement the
vetoableChange( ) method. This method is invoked by the bean for all of its registered
listeners to inform them of a property change. Any object that does not approve of a
property change can throw a PropertyVetoException within its vetoableChange( )
method to inform the bean whose constrained property was changed that the change
was not approved.

Inside java.beans package

The classes and packages in the java.beans package can be categorised into three types
(NOTE: following is not the complete list).

1) **Design Support**

Classes - Beans, PropertyEditorManager, PropertyEditorSupport
Interfaces - Visibility, VisibilityState, PropertyEditor, Customizer

2) **Introspection Support**.

Classes - Introspector, SimpleBeanInfo, BeanDescriptor, EventSetDescriptor,
FeatureDescriptor, IndexedPropertyDescriptor, MethodDescriptor,
ParameterDescriptor, PropertyDescriptor
Interfaces - BeanInfo

3) **Change Event-Handling Support**.

Classes - PropertyChangeEvent, VetoableChangeEvent, PropertyChangeSupport,
VetoableChangeSupport
Interfaces - PropertyChangeListener, VetoableChangeListener

1.4   WHAT IS EJB?

Enterprise JavaBeans are software component models, their purpose is to build/support
enterprise specific problems. EJB - is a reusable server-side software component.
Enterprise JavaBeans facilitates the development of distributed Java applications,
providing an object-oriented transactional environment for building distributed, multi-tier
enterprise components. An EJB is a remote object, which needs the services of an EJB
container in order to execute.

The primary goal of a EJB is **WORA** (Write Once Run Anywhere). Enterprise JavaBeans
takes a high-level approach to building distributed systems. It frees the application
developer and enables him/her to concentrate on programming only the business logic
while removing the need to write all the "plumbing" code that's required in any enterprise
application. For example, the enterprise developer no longer needs to write code that
handles transactional behaviour, security, connection pooling, networking or threading.
The architecture delegates this task to the server vendor.

1.5   EJB ARCHITECTURE

The Enterprise JavaBeans spec defines a server component model and specifies, how to create server-side, scalable, transactional, multiuser and secure enterprise-level components. Most important, EJBs can be deployed on top of existing transaction processing systems including traditional transaction processing monitors, Web, database and application servers.

A typical EJB architecture consists of:

- **EJB clients:** EJB client applications utilise the Java Naming and Directory   Interface (JNDI) to look up references to home interfaces and use home and remote EJB interfaces to utilise all EJB-based functionality.

- **EJB home interfaces (and stubs):** EJB home interfaces provide operations   for clients to create, remove, and find handles to EJB remote interface objects. Underlying stubs marshal home interface requests and unmarshal home interface responses for the client.

- **EJB remote interfaces (and stubs):** EJB remote interfaces provide business-specific client interface methods defined for a particular EJB. Underlying stubs marshal remote interface requests and unmarshal remote interface responses for the client.

- **EJB implementations:** EJB implementations are the actual EJB application components implemented by developers to provide any application-specific business method invocation, creation, removal, finding, activation, passivation, database storage, and database loading logic.

- **Container EJB implementations (skeletons and delegates):** The container manages the distributed communication skeletons used to marshal and unmarshal data sent to and from the client. Containers may also store EJB implementation instances in a pool and use delegates to perform any service-management operations related to a particular EJB before calls are delegated to the EJB implementation instance.

Some of the advantages of pursuing an EJB solution are:

- EJB gives developers architectural independence.

- EJB is WORA for server-side components.

- EJB establishes roles for application development.

- EJB takes care of transaction management.

- EJB provides distributed transaction support.

- EJB helps create portable and scalable solutions.

- EJB integrates seamlessly with CORBA.

- EJB provides for vendor-specific enhancements.

1.6   BASIC EJB EXAMPLE

To create an EJB we need to create Home, Remote and Bean classes.

Home Interface

```
import java.rmi.*;
import javax.ejb.*;
import java.util.*;
public interface HelloObject extends EJBObject {
  public String sayHello() throws RemoteException;
}
```

Remote Interface

```
import java.rmi.*;
import javax.ejb.*;
import java.util.*;


public interface HelloHome extends EJBHome {
  public HelloObject create() throws RemoteException,
CreateException;
}
```

Bean Implementation

```
import java.rmi.RemoteException;
import javax.ejb.*;

public class HelloBean implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext)
{
    this.sessionContext = sessionContext;
  }
  public String sayHello() throws java.rmi.RemoteException {
    return "Hello World!!!!!";
  }
}
```

Deployment Descriptor

```xml
<ejb-jar>
<description>HelloWorld deployment descriptor</description>
<display-name>HelloWorld</display-name>
<enterprise-beans>
<session>
<description> HelloWorld deployment descriptor
</description>
<display-name>HelloWorld</display-name>
<ejb-name>HelloWorld</ejb-name>
<home>HelloWorldHome</home>
<remote>HelloWorld</remote>
<ejb-class>HelloWorldBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>HelloWorld</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

EJB Client

```java
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.transaction.UserTransaction;
import javax.rmi.PortableRemoteObject;

public class HelloClient {
        public static void main(String args[]) {
            try {
                Context initialContext = new InitialContext();
                Object objref =
initialContext.lookup("HelloWorld");
                HelloWorldHome home =

        (HelloWorldHome)PortableRemoteObject.narrow(objref,
                HelloWorldHome.class);
                HelloWorld myHelloWorld = home.create();
                String message = myHelloWorld.sayHello();
                System.out.println(message);
                } catch (Exception e) {
                        System.err.println(" Erreur : " + e);
                        System.exit(2);
            }
        }
}
```

1.7   EJB TYPES

EJBs are distinguished along three main functional roles. Within each primary role, the EJBs are further distinguished according to subroles. By partitioning EJBs into roles, the programmer can develop an EJB according to a more focused programming model than, if, for instances such roles were not distinguished earlier. These roles also allow the EJB container to determine the best management of a particular EJB based on its programming model type.

There are three main types of beans:

- Session Bean

- Entity Beans

- Message-driven Beans

1.7.1   Session Bean

A session EJB is a non persistent object. Its lifetime is the duration of a particular interaction between the client and the EJB. The client normally creates an EJB, calls methods on it, and then removes it. If, the client fails to remove it, the EJB container will remove it after a certain period of inactivity. There are two types of session beans:

- **Stateless Session Beans:** A stateless session EJB is shared between a number of clients. It does not maintain conversational state. After each method call, the container may choose to destroy a stateless session bean, or recreate it, clearing itself out, of all the information pertaining the invocation of the last method. The algorithm for creating new instance or instance reuse is container specific.

- **Stateful Session Beans:** A stateful session bean is a bean that is designed to service business processes that span multiple method requests or transaction. To do this, the stateful bean retains the state for an individual client. If, the stateful bean's state is changed during method invocation, then, that same state will be available to the same client upon invocation.

1.7.1.1 Life Cycle of a Stateless Session Bean

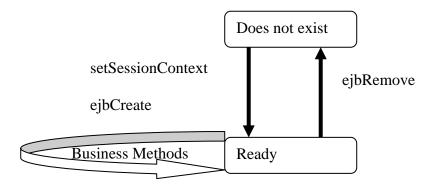The *Figure 1* shows the life cycle of a Stateless Session Bean.

Figure 1: Life Cycle of Stateless Session Bean

- **Does not exist:** In this state, the bean instance simply does not exist.

- **Ready state:** When EJB Server is first started, several bean instances are created and placed in the Ready pool. More instances might be created by the container as and when needed by the EJB container

1.7.1.2 Life Cycle of a Stateful Session Bean

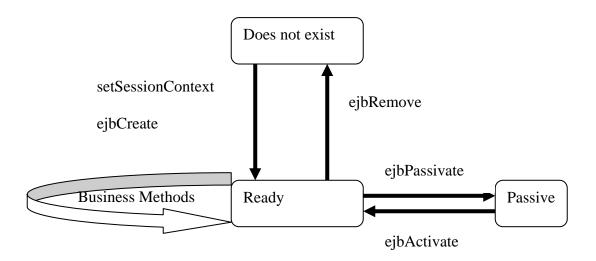The *Figure 2* shows the life cycle of a Stateful Session Bean.

Figure 2: Life cycle of a stateful session bean

- **Does not exist:** In this state, the bean instance simply does not exist.

- **Ready state:** A bean instance in the ready state is tied to a particular client and engaged in a conversation.

- **Passive state:** A bean instance in the passive state is passivated to conserve resources.

1.7.1.3 Required Methods in Session Bean

The following are the required methods in a Session Bean:

setSessionContext(SessionContext ctx) :
Associate your bean with a session context. Your bean can make a query to the context about its current transactional state, and its current security state.

ejbCreate(…) :

Initialise your session bean. You would need to define several ejbCreate (…) methods and then, each method can take up different arguments. There should be at least one ejbCreate() in a session bean.

ejbPassivate():

This method is called for, just before the session bean is passivated and releases any resource that bean might be holding.

ejbActivate():

This method is called just for, before the session bean is activated and acquires the resources that it requires.

ejbRemove():

This method is called for, by the ejb container just before the session bean is removed from the memory.

1.7.1.4 The use of a Session Bean

In general, one should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.

- The state of the bean is not persistent, existing only for a short period and therefore.

- The bean implements a web service.

Stateful session beans are appropriate if, any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.

- The bean needs to hold information about the client across method invocations.

- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.

- Behind the scenes, the bean manages the work flow of several enterprise beans.

To improve performance, one might choose a stateless session bean if, it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send a promotional email to several registered users.

1.7.2   Entity Bean

Entity EJBs represent persistent objects. Their lifetimes is not related to the duration of interaction with clients. In nearly all cases, entity EJBs are synchronised with relational databases. This is how persistence is achieved. Entity EJBs are always shared amongst clients. A client cannot get an entity EJB to itself. Thus, entity EJBs are nearly always used as a scheme for mapping relational databases into object-oriented applications. An important feature of entity EJBs is that they have identity—that is, one can be distinguished from another. This is implemented by assigning a primary key to each instance of the EJB, where 'primary key' has the same meaning as it does for database management. Primary keys that identify EJBs can be of any type, including programmer-defined classes.

There are two type of persistence that entity EJB supports. These persistence types are:

- **Bean-managed persistence (BMP):** The entity bean's implementation manages persistence by coding database access and updating statements in callback methods.

- **Container-managed persistence (CMP):** The container uses specifications made in the deployment descriptor to perform database access and update statements automatically.

1.7.2.1 Life Cycle of an Entity Bean
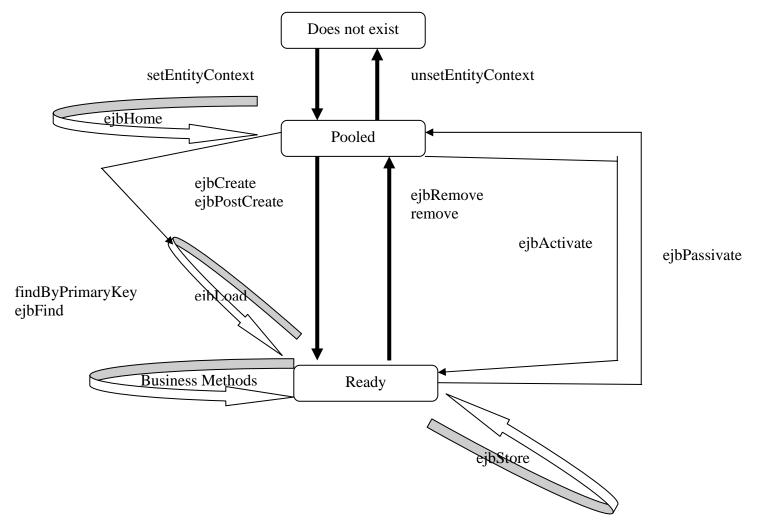
The *Figure 3* shows the life cycle of an entity bean.

```
                            ┌──────────────────┐
                            │  Does not exist  │
                            └──────────────────┘
     setEntityContext                              unsetEntityContext

        ejbHome                ┌──────────┐
                               │  Pooled  │
                               └──────────┘
           ejbCreate                            ejbRemove
           ejbPostCreate                        remove
                                                          ejbActivate        ejbPassivate

  findByPrimaryKey
  ejbFind              ejbLoad

        Business Methods       ┌──────────┐
                               │  Ready   │
                               └──────────┘
                                         ejbStore
```

Figure 3: Life cycle of an entity bean

An entity bean has the following three states:

- **Does not exist:** In this state, the bean instance simply does not exist.
- **Pooled state:** When the EJB server is first started, several bean instances are created and placed in the pool. A bean instance in the pooled state is not tied to a particular data, that is, it does not correspond to a record in a database table. Additional bean instances can be added to the pool as needed, and a maximum number of instances can be set.
- **Ready state:** A bean instance in the ready state is tied to a particular data, that is, it represents an instance of an actual business object.

1.7.2.2 Required Methods in Entity Bean

Entity beans can be bean managed or container managed. Here, are the methods that are required for entity beans:

setEntityContext():

This method is called for, if a container wants to increase its pool size of bean instances, then, it will instantiate a new entity bean instance. This method associates a bean with context information. Once this method is called for, then, the bean can access the information about its environment

ejbFind(..):

This method is also known as the Finder method. The Finder method locates one or more existing entity bean data instances in underlying persistent store.

ejbHome(..):

The Home methods are special business methods because they are called from a bean in the pool before the bean is associated with any specific data. The client calls for, home methods from home interface or local home interface.

ejbCreate():

This method is responsible for creating a new database data and for initialising the bean.

ejbPostCreate():

There must be one ejbPostCreate() for each ejbCreate(). Each method must accept the same parameters. The container calls for, ejbPostCreate() right after ejbCreate().

ejbActivate():

When a client calls for, a business method on a EJB object but no entity bean instance is bound to EJB object, the container needs to take a bean from the pool and transition into a ready state. This is called Activation. Upon activation the ejbActivate() method is called for by the ejb container.

ejbLoad():

This method is called for, to load the database in the bean instance.

ejbStore():

This method is used for, to update the database with new values from the memory. This method is also called for during ejbPassivate().

ejbPassivate():

This method is called for, by the EJB container when an entity bean is moved from the ready state to the pool state.

ejbRemove():
This method is used to destroy the database data. It does not remove the object. The object is moved to the pool state for reuse.

unsetEntityContext():

This method removes the bean from its environment. This is called for, just before destroying the entity bean.

1.7.2.3 The Use of the Entity Bean

You could probably use an entity bean under the following conditions:

- The bean represents a business entity and not a procedure. For example, BookInfoBean would be an entity bean, but BookInfoVerifierBean would be a session bean.

- The bean's state must be persistent. If the bean instance terminates or if the Application Server is shut down, the bean's state still exists in persistent storage (a database).

1.7.3   Message Driven Bean

A message-driven bean acts as a consumer of asynchronous messages. It cannot be called for, directly by clients, but is activated by the container when a message arrives. Clients interact with these EJBs by sending messages to the queues or topics to which they are listening. Although a message-driven EJB cannot be called for, directly by clients, it can call other EJBs itself.

1.7.3.1 Life Cycle of a Message Driven Bean

The *Figure 4* shows the life cycle of a Message Driven Bean:
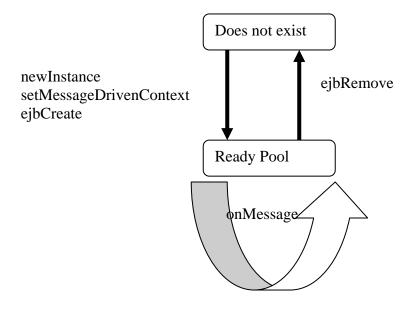
Figure 4: Life Cycle of a Message Driven Bean

A message driven bean has the following two states:

- **Does not exist:** In this state, the bean instance simply does not exist. Initially, the bean   exists in the; does not exist state.

- **Pooled state:** After invoking the ejbCreate() method, the MDB instance is in the ready pool, waiting to consume incoming messages. Since, MDBs are stateless, all instances of MDBs in the pool are identical; they're allocated to process a message and then return to the pool.

1.7.3.2 Method for Message Driven Bean

onMessage(Message):

This method is invoked for each message that is consumed by the bean. The container is responsible for serialising messages to a single message driven bean.

ejbCreate():

When this method is invoked, the MDB is first created and then, added to the 'to pool'.

ejbCreate():

When this method is invoked, the MDB is removed from the 'to pool'.

setMessageDrivenContext(MessageDrivenContext):

This method is called for, as a part of the event transition that message driven bean goes through, when it is being added to the pool. This is called for, just before the ejbCreate().

1.7.3.3 The Use of the Message Driven Bean

Session beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer not blocking synchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

⬚ Check Your Progress 1

1) What is the relationship between Enterprise JavaBeans and JavaBeans?

………………………………………………………………………………...…

…………………………………………………………………………….…

……………………………………………………………………………

2) Explain the different types of Enterprise beans briefly.

………………………………………………………………………………...…

…………………………………………………………………………..……

………………………………………………………………………………..

3) What is the difference between Java Bean and Enterprise Java Bean?

………………………………………………………………………………...…

…….……………………………………………………………………….…

……………………………………………………………………………

4) Can Entity Beans have no create() methods?

………………………………………………………………………………...…

……………………………………………………………………….……

………………………………………………………………………………..

5) What are the call back methods in the Session Bean?

………………………………………………………………………………...…

……………………………………………………………………….……

………………………………………………………………………………..

6) What are the call back methods of Entity Beans?

………………………………………………………………………………...…

……………………………………………………………………….……

………………………………………………………………………………..

7) Can an EJB send asynchronous notifications to its clients?

………………………………………………………………………………...…

……………………………………………………………………….……

………………………………………………………………………………..

8) What is the advantage of using an Entity bean for database operations, over directly using JDBC API to do database operations? When would I need to use one over the other?

………………………………………………………………………………...…

…….……………………………………………………………………….…

……………………………………………………………………………

9) What are the callback methods in Entity beans?

………………………………………………………………………………...…

…….…………………………………………………………………….…

……………………………………………………………………………

10) What is the software architecture of EJB?

………………………………………………………………………………...…

.………………………………………………………………………………...…

……………………………………………………………………………………

11) What are session Beans? Explain the different types.

………………………………………………………………………………...…

.………………………………………………………………………………...…

……………………………………………………………………………………

1.8 SUMMARY

Java bean and enterprise java beans are most widely used java technology. Both technologies contribute towards component programming. GUI JavaBeans can be used in visual tools. Currently most of the Java IDE and applications are using GUI JavaBeans. For enterprise application we have to choose EJB. Based on the requirements we can either use Session Bean, Entity Bean or Message Driven Bean. Session and Entity beans can be used in normal scenarios where we have synchronous mode. For asynchronous messaging like Publish /Subscribe we should use message driven beans.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Enterprise JavaBeans extends the JavaBeans component model to handle the needs of transactional business applications.

JavaBeans is a component model for the visual construction of reusable components for the Java platform. Enterprise JavaBeans extends JavaBeans to middle-tier/server side business applications. The extensions that Enterprise JavaBeans adds to JavaBeans include support for transactions, state management, and deployment time attributes.

Although applications deploying Enterprise JavaBeans architecture are independent of the underlying communication protocol, the architechture of the Enterprise JavaBeans specifies how communication among components, maps into the underlying communication protocols, such as CORBA/IIOP.

2) Different types of Enterprise Beans are following :

- **Stateless Session Bean:** An instance of these non-persistent EJBs provides service without storing an interaction or conversation state between methods. Any instance can be used for any client.

- **Stateful Session Bean**: An instance of these non-persistent EJBs maintains state across methods and transactions. Each instance is associated with a particular client.\

- **Entity Bean:** An instance of these persistent EJBs represents an object view of the data, usually rows in a database. They have a primary key as a unique identifier. Entity bean persistence can be either container-managed or bean-managed.

- **Message:Driven Bean:** An instance of these EJBs is integrated with the Java Message Service (JMS) to provide the ability for message-driven beans to act as a standard JMS message consumer and perform asynchronous processing between the server and the JMS message producer.

3) Java Bean is a plain java class with member variables and getter setter methods. Java Beans are defined under JavaBeans specification as Java-Based software component model which includes features such as introspection, customisation, events, properties and persistence.

Enterprise JavaBeans or EJBs for short are Java-based software components that comply with Java's EJB specification. EJBs are delpoyed on the EJB container and execute in the EJB container. EJB is not that simple, it is used for building distributed applications.

Examples of EJB are Session Bean, Entity Bean and Message Driven Bean. EJB is used for server side programming whereas java bean is a client side programme. While Java Beans are meant only for development the EJB is developed and then deployed on EJB Container.

4) Entity Beans can have no create() methods. Entity Beans have no create() an method, when an entity bean is not used to store the data in the database. In this case, entity bean is used to retrieve the data from the database.

5) Callback methods are called for, by the container to notify the important events to the beans in its life cycle. The callback methods are defined in the javax.ejb.EntityBean interface.The callback methods example are ejbCreate(), ejbPassivate(), and ejbActivate().

6) An entity bean consists of 4 groups of methods:

- **Create methods**: To create a new instance of a CMP entity bean, and therefore, insert data into the database, the create() method on the bean's home interface must be invoked. They look like this: EntityBeanClass ejbCreateXXX(parameters), where EntityBeanClass is an Entity Bean you are trying to instantiate, ejbCreateXXX(parameters) methods are used for creating Entity Bean instances according to the parameters specified and to some programmer-defined conditions.

  A bean's home interface may declare zero or more create() methods, each of which must have corresponding ejbCreate() and ejbPostCreate() methods in the bean class. These creation methods are linked at run time, so that when a create() method is invoked on the home interface, the container delegates the invocation to the corresponding ejbCreate() and ejbPostCreate() methods on the bean class.

- **Finder methods**: The methods in the home interface that begin with "find" are called the find methods. These are used to query to the EJB server for specific entity beans, based on the name of the method and arguments passed. Unfortunately, there is no standard query language defined for find methods, so each vendor will implement the find method differently. In CMP entity beans, the find methods are not implemented with matching methods in the bean class; containers implement them when the bean is deployed in a vendor specific manner. The deployer will use vendor specific tools to tell the container how a particular find method should behave. Some vendors will use object-relational mapping tools to define the behaviour of a find method while others will simply require the deployer to enter the appropriate SQL command.

  There are two basic kinds of find methods: single-entity and multi-entity. Single-entity find methods return a remote reference to the one specific entity bean that matches the find request. If, no entity beans are found, the method throws an ObjectNotFoundException . Every entity bean must define the

single-entity find method with the method name findByPrimaryKey(), which takes the bean's primary key type as an argument.

The multi-entity find methods return a collection (Enumeration or Collection type) of entities that match the find request. If, no entities are found, the multi-entity find returns an empty collection.

- **Remove methods**: These methods (you may have up to 2 remove methods, or don't have them at all) allow the client to physically remove Entity beans by specifying either Handle or a Primary Key for the Entity Bean.

- **Home methods**: These methods are designed and implemented by a developer, and EJB specifications do not require them as such, except when, there is the need to throw a RemoteException in each home method.

7) Asynchronous notification is a known hole in the first versions of the EJB spec. The recommended solution to this is to use JMS (Java Messaging Services), which is now, available in J2EE-compliant servers. The other option, of course, is to use client-side threads and polling. This is not an ideal solution, but it's workable for many scenarios.

8) Entity Beans actually represents the data in a database. It is not that Entity Beans replaces JDBC API. There are two types of Entity Beans – Container Managed and Bean Mananged. In a Container Managed Entity Bean – Whenever, the instance of the bean is created, the container automatically retrieves the data from the DB/Persistance storage and assigns to the object variables in the bean for the user to manipulate or use them. For this, the developer needs to map the fields in the database to the variables in deployment descriptor files (which varies for each vendor). In the Bean Managed Entity Bean – the developer has to specifically make connection, retrive values, assign them to the objects in the ejbLoad() which will be called for, by the container when it instatiates a bean object. Similarly, in the ejbStore() the container saves the object values back to the persistance storage. ejbLoad and ejbStore are callback methods and can only be invoked by the container. Apart from this, when you are use Entity beans you do not need to worry about database transaction handling, database connection pooling etc. which are taken care of by the ejb container. But, in case of JDBC you have to explicitly take care of the above features. The great thing about the entity beans is that container managed is, that, whenever the connection fail during transaction processing, the database consistancy is mantained automatically. The container writes the data stored at persistant storage of the entity beans to the database again to provide the database consistancy. Whereas in jdbc api, developers need to maintain the consistency of the database manually.

9) The bean class defines create methods that match methods in the home interface and business methods that match methods in the remote interface. The bean class also implements a set of callback methods that allow the container to notify the bean of events in its life cycle. The callback methods are defined in the javax.ejb.EntityBean interface that is implemented by all entity beans.The EntityBean interface has the following definition. Notice that, the bean class implements these methods.

```
public interface javax.ejb.EntityBean {
public void setEntityContext();
public void unsetEntityContext();
    public void ejbLoad();
    public void ejbStore();
    public void ejbActivate();
    public void ejbPassivate();
```

```
    public void ejbRemove();
    }
```

The setEntityContext() method provides the bean with an interface to the container called the EntityContext. The EntityContext interface contains methods for obtaining information about the context under which the bean is operating at any particular moment. The EntityContext interface is used to access security information about the caller; to determine the status of the current transaction or to force a transaction rollback; or to get a reference to the bean itself, its home, or its primary key. The EntityContext is set only once in the life of an entity bean instance, so its reference should be put into one of the bean instance's fields if it will be needed later.

The unsetEntityContext() method is used at the end of the bean's life cycle before the instance is evicted from the memory to dereference the EntityContext and perform any last-minute clean-up.

The ejbLoad() and ejbStore() methods in CMP entities are invoked when the entity bean's state is being synchronised with the database. The ejbLoad() is invoked just after the container has refreshed the bean container-managed fields with its state from the database.

The ejbStore() method is invoked just before the container is about to write the bean container-managed fields to the database. These methods are used to modify data as it is being synchronised. This is common when the data stored in the database is different than the data used in the bean fields.

The ejbPassivate() and ejbActivate() methods are invoked on the bean by the container just before the bean is passivated and just after the bean is activated, respectively. Passivation in entity beans means that the bean instance is disassociated with its remote reference so that the container can evict it from the memory or reuse it. It's a resource conservation measure that the container employs to reduce the number of instances in the memory. A bean might be passivated if it hasn't been used for a while or as a normal operation performed by the container to maximise reuse of resources. Some containers will evict beans from the memory, while others will reuse instances for other more active remote references. The ejbPassivate() and ejbActivate() methods provide the bean with a notification as to when it's about to be passivated (disassociated with the remote reference) or activated (associated with a remote reference).

10) Session and Entity EJBs consist of 4 and 5 parts respectively:

   a) A remote interface (a client interacts with it),

   b) A home interface (used for creating objects and for declaring business methods),

   c) A bean object (an object, which actually performs business logic and EJB-specific operations).

   d) A deployment descriptor (an XML file containing all information required for maintaining the EJB) or a set of deployment descriptors (if you are using some container-specific features).

   e) A Primary Key class - that is only Entity bean specific.

11) A session bean is a non-persistent object that implements some business logic running on the server. One way to think of a session object is as a logical extension of the client program that runs on the server.

   Session beans are used to manage the interactions of entity and other session beans, access resources, and generally perform tasks on behalf of the client.

There are two basic kinds of session bean: Stateless and Stateful.

Stateless session beans are made up of business methods that behave like procedures; they operate only on the arguments passed to them when they are invoked. Stateless beans are called stateless because they are transient; they do not maintain business state between method invocations.Each invocation of a stateless business method is independent of any previous invocations. Because stateless session beans are stateless, they are easier for the EJB container to manage, so they tend to process requests faster and use less resources.

Stateful session beans encapsulate business logic and are state specific to a client. Stateful beans are called "tateful" because they do maintain business state between method invocations, held the in memory and not persistent. Unlike stateless session beans, clients do not share stateful beans. When a client creates a stateful bean, that bean instance is dedicated to the service of only that client. This makes it possible to maintain conversational state, which is business state that can be shared by methods in the same stateful bean.

1.10 FURTHER READINGS/REFERENCES

- Paco Gomez and Peter Zadronzy, *Professional Java 2 Enterprise Edition with BEA Weblogic Server,* WROX Press Ltd

- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,* New Riders Publishing

- Richard Monson-Haefel, *Enterprise JavaBeans (3rd Edition),* O'Reilly

- Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns, *Applying Enterprise JavaBeans™:Component-Based Development for the J2EE™ Platform,* Second Edition, Pearson Education

- Robert Englander, *Developing Java Beans,* O'Reilly Media

Reference websites:
- www.j2eeolympus.com
- www.phptr.com
- www.samspublishing.com
- www.oreilly.com
- www.roseindia.net
- www.caucho.com
- www.tutorialized.com
- www.stardeveloper.com

UNIT 2 ENTERPRISE JAVA BEANS: ARCHITECTURE

Structure   **Page Nos.**

2.0   INTRODUCTION

In the previous unit, we have learnt the basics of Enterprise java beans and different type of java beans. In this unit, we shall learn about the architecture of Enterprise java beans. "Enterprise JavaBeans" (EJBs) are distributed network aware components for developing secure, scalable, transactional and multi-user components in a J2EE environment. Enterprise JavaBeans (EJB) is suitable architecture for developing, deploying, and managing reliable enterprise applications in production environments. In this unit, we will study the benefits of using EJB architecture for enterprise applications. Enterprise application architectures have evolved through many phases. Such architectures inevitably evolve because, the underlying computer support and delivery systems have changed enormously and will continue to change in the future. With the growth of the Web and the Internet, more and more enterprise applications, including intranet and extranet applications, are now Web based. Enterprise application architectures have undergone an extensive evolution. The first generation of enterprise applications consisted of centralised mainframe applications. In the late 1980s and early 1990s, most new enterprise applications followed a two-tier, or client/server, architecture. Later, enterprise architecture evolved to a three-tier architecture and then to a Web-based architecture. The current evolutionary state is now represented by the J2EE application architecture. The J2EE architecture provides comprehensive support for two- and three-tier applications, as well as Web-based and Web services applications. Now, we will learn, about EJB architecture in detail.

2.1   OBJECTIVES

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

- list the different goals of Enterprise java beans in the business environment;
- list the benefits offered by EJB architecture to application developers and customers;
- differentiate the services offered by EJB architecture;
- list the various kinds of restrictions for the services offered by EJB;
- differentiate the between programming style and life cycle between stateful sessions;
- make a choice between Entity beans or stateful session beans, and
- install and run application like Jboss for EJB.

2.2   GOALS OF JAVA ENTERPRISE BEANS

The EJB Specifications try to meet several goals which are described as following:

- EJB is designed to make it easy for developers to create applications, freeing them from low-level system details of managing transactions, threads, load balancing, and so on. Application developers can concentrate on business logic and leave the details of managing the data processing to the framework. For specialised applications, though, it's to customise these lower-level services.

- The EJB Specifications define the major structures of the EJB framework, and then specifically define the contracts between them. The responsibilities of the client, the server, and the individual components are all clearly spelled out. A developer creating an Enterprise JavaBean component has a very different role from someone creating an EJB-compliant server, and the specification describes the responsibilities of each.

- EJB aims to be the standard way for client/server applications to be built in the Java language. Just as the original JavaBeans (or Delphi component etc.) from different vendors can be combined to produce a custom client, EJB server components from different vendors can be combined to produce a custom server. EJB components, being Java classes, will of course run in any EJB-compliant server without recompilation. This is a benefit that platform-specific solutions cannot offer.

- Finally, the EJB is compatible with and uses other Java APIs, can interoperate with non-Java applications, and is compatible with CORBA.

2.3   ARCHITECTURE OF AN EJB/ CLIENT SYSTEM

There are various architectures of EJB (J2EE) available which are used for various purposes. J2EE is a standard architecture specifically oriented to the development and deployment of enterprise Web-oriented applications that use Java programming language. ISVs and enterprises can use the J2EE architecture for only not the development and deployment of intranet applications, thus effectively replacing the two-tier and three-tier models, but also for the development of Internet applications, effectively replacing the cgi-bin-based approach. The J2EE architecture provides a flexible distribution and tiering model that allows enterprises to construct their applications in the most suitable manner.

Now, let us understand how an EJB client/server system operates and the underlying architecture of EJB, we need to understand the basic parts of an EJB system: *the EJB component, the EJB container, and the EJB object.*

The Enterprise JavaBeans Component

An Enterprise JavaBean is a component, just like a traditional JavaBean. Enterprise JavaBeans execute within an EJB container, which in turn executes within an EJB server. Any server that can host an EJB container and provide it with the necessary services can be an EJB server. (Hence, many existing servers are being extended to be EJB servers.) An EJB component is the type of EJB class most likely to be considered an "Enterprise JavaBean". It's a Java class, written by an EJB developer, that implements business logic. All the other classes in the EJB system either support client access to or provide services (like persistence, and so on) to EJB component classes.

The Enterprise JavaBeans Container

The EJB container is where the EJB component "lives". The EJB container provides services such as transaction and resource management, versioning, scalability, mobility,

persistence, and security to the EJB components it contains. Since the EJB container handles all these functions, the EJB component developer can concentrate on business rules, and leave database manipulation and other such fine details to the container. For example, if a single EJB component decides that the current transaction should be aborted, it simply tells its container (container is responsible for performing all rollbacks, or doing whatever is necessary to cancel a transaction in progress). Multiple EJB component instances typically exist inside a single EJB container.

The EJB Object and the Remote Interface

Client programs execute methods on remote EJBs by way of an EJB object. The EJB object implements the "remote interface" of the EJB component on the server. The remote interface represents the "business" methods of the EJB component. The remote interface does the actual, useful work of an EJB object, such as creating an order form or deferring a patient to a specialist. EJB objects and EJB components are separate classes, though from the outside (i.e., by looking at their interfaces), they look identical. This is because, they both implement the same interface (the EJB component's remote interface), but they do very different things. An EJB component runs on the server in an EJB container and implements the business logic. *The EJB object runs on the client and remotely executes the EJB component's methods.*

Let us consider an analogy to understand the concept of EJB. Assume your VCD player is an EJB component. The EJB object is then analogous to your remote control: both the VCD and the remote control have the same buttons on the front, but they perform different functions. By pushing the Rewind button on your VCD player's remote control is equivalent to pushing the Rewind button on the VCD player itself, even though it's the VCD player -- and not the remote -- that actually rewinds a tape.

Working of EJB

The actual implementation of an EJB object is created by a code generation tool that comes with the EJB container. The EJB object's interface is the EJB component's remote interface. The EJB object (created by the container and tools associated with the container) and the EJB component (created by the EJB developer) implement the same remote interface. To the client, an EJB object looks just like an object from the application domain -- an order form, for example. But the EJB object is just a stand-in for the actual EJB, running on the server inside an EJB container. When the client calls a method on an EJB object, the EJB object method communicates with the remote EJB container, requesting that the same method be called, on the appropriate (remote) EJB, with the same arguments, on the client's behalf. This is explained with the help of the *Figure1.* This is the core concept behind how an EJB client/server system works.
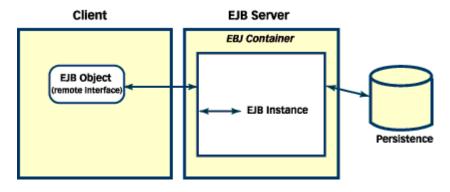

Figure 1: Enterprise JavaBeans in the Client and the Server

2.4   ADVANTAGES OF EJB ARCHITECTURE

Benefits to Application Developers:

The EJB component architecture is the backbone of the J2EE platform. The EJB architecture provides the following benefits to the application developer:

- **Simplicity:** It is easier to develop an enterprise application with the EJB architecture than without it. Since, EJB architecture helps the application developer access and use enterprise services with minimal effort and time, writing an enterprise bean is almost as simple as writing a Java class. The application developer does not have to be concerned with system-level issues, such as security, transactions, multithreading, security protocols, distributed programming, connection resource pooling, and so forth. As a result, the application developer can concentrate on the business logic for domain-specific applications.

- **Application Portability:** An EJB application can be deployed on any J2EE-compliant server. This means that the application developer can sell the application to any customers who use a J2EE-compliant server. This also means that enterprises are not locked into a particular application server vendor. Instead, they can choose the "best-of-breed" application server that meets their requirements.

- **Component Reusability:** An EJB application consists of enterprise bean components. Each enterprise bean is a reusable building block. There are two essential ways to reuse an enterprise bean:

  1) An enterprise bean not yet deployed can be reused at application development time by being included in several applications. The bean can be customised for each application without requiring changes, or even access, to its source code.

  2) Other applications can reuse an enterprise bean that is already deployed in a customer's operational environment, by making calls to its client-view interfaces. Multiple applications can make calls to the deployed bean.

  In addition, the business logic of enterprise beans can be reused through Java subclassing of the enterprise bean class.

- **Ability to Build Complex Applications:** The EJB architecture simplifies building complex enterprise applications. These EJB applications are built by a team of developers and evolve over time. The component-based EJB architecture is well suited to the development and maintenance of complex enterprise applications. With its clear definition of roles and well-defined interfaces, the EJB architecture promotes and supports team-based development and lessens the demands on individual developers.

- **Separation of Business Logic from Presentation Logic:** An enterprise bean typically encapsulates a business process or a business entity (an object representing enterprise business data), making it independent of the presentation logic. The business programmer need not worry about formatting the output; the Web page designer developing the Web page need be concerned only with the output data that will be passed to the Web page. In addition, this separation makes it possible to develop multiple presentation logic for the same business process or to change the presentation logic of a business process without needing to modify the code that implements the business process.

- **Easy Development of Web Services:** The Web services features of the EJB architecture provide an easy way for Java developers to develop and access Web services. Java developers do not need to bother about the complex details of Web services description formats and XML-based wire protocols but instead can program at the familiar level of enterprise bean components, Java interfaces and data types. The tools provided by the container manage the mapping to the Web services standards.

- **Deployment in many Operating Environments**— The goal of any software development company is to sell an application to many customers. Since, each customer has a unique operational environment, the application typically needs to be customised at deployment time to each operational environment, including different database schemas.

    1) The EJB architecture allows the bean developer to separate the common application business logic from the customisation logic performed at deployment.

    2) The EJB architecture allows an entity bean to be bound to different database schemas. This persistence binding is done at deployment time. The application developer can write a code that is not limited to a single type of database management system (DBMS) or database schema.

    3) The EJB architecture facilitates the deployment of an application by establishing deployment standards, such as those for data source lookup, other application dependencies, security configuration, and so forth. The standards enable the use of deployment tools. The standards and tools remove much of the possibility of miscommunication between the developer and the deployer.

- **Distributed Deployment:** The EJB architecture makes it possible for applications to be deployed in a distributed manner across multiple servers on a network. The bean developer does not have to be aware of the deployment topology when developing enterprise beans but rather writes the same code whether the client of an enterprise bean is on the same machine or a different one.

- **Application Interoperability:** The EJB architecture makes it easier to integrate applications from different vendors. The enterprise bean's client-view interface serves as a well-defined integration point between applications.

- **Integration with non-Java Systems:** The related J2EE APIs, such as the J2EE Connector specification and the Java Message Service (JMS) specification, and J2EE Web services technologies, such as the Java API for XML-based RPC (JAX-RPC), make it possible to integrate enterprise bean applications with various non-Java applications, such as ERP systems or mainframe applications, in a standard way.

- **Educational Resources and Development Tools:** Since, the EJB architecture is an industry wide standard, the EJB application developer benefits from a growing body of educational resources on how to build EJB applications. More importantly, powerful application development tools available from leading tool vendors simplify the development and maintenance of EJB applications.

Benefits to Customers

A customer's perspective on EJB architecture is different from that of the application developer. The EJB architecture provides the following benefits to the customer:

- **Choice of Server:** Because the EJB architecture is an industry wide standard and is part of the J2EE platform, customer organisations have a wide choice of J2EE-compliant servers. Customers can select a product that meets their needs in terms of scalability, integration capabilities with other systems, security protocols, price, and

so forth. Customers are not locked in to a specific vendor's product. Should their needs change, customers can easily redeploy an EJB application in a server from a different vendor.

- **Facilitation of Application Management:** Because the EJB architecture provides a standardised environment, server vendors have the required motivation to develop application management tools to enhance their products. As a result, sophisticated application management tools provided with the EJB container allow the customer's IT department to perform such functions as starting and stopping the application, allocating system resources to the application, and monitoring security violations, among others.

- **Integration with a Customer's Existing Applications and Data:** The EJB architecture and the other related J2EE APIs simplify and standardise the integration of EJB applications with any non-Java applications and systems at the customer operational environment. For example, a customer does not have to change an existing database schema to fit an application. Instead, an EJB application can be made to fit the existing database schema when it is deployed.

- **Integration with Enterprise Applications of Customers, Partners, and Suppliers:** The interoperability and Web services features of the EJB architecture allows EJB-based applications to be exposed as Web services to other enterprises. This means that enterprises have a single, consistent technology for developing intranet, Internet, and e-business applications.

- **Application Security:** The EJB architecture shifts most of the responsibility for an application's security from the application developer to the server vendor, system administrator, and the deployer. The people performing these roles are more qualified than the application developer to secure the application. This leads to better security of operational applications.

2.5   SERVICES OFFERED BY EJB

As we have learnt earlier, in J2EE all the components run inside their own containers. JSP, Servlets and JavaBeans and EJBs have their own web container. The container of EJB provides certain built-in services to EJBs, which is used by EJBs to perform different functions. The services that EJB container provides are:

- Component Pooling
- Resource Management
- Transaction Management
- Security
- Persistence
- Handling of multiple clients

i) Component Pooling

The EJB container handles the pooling of EJB components. If, there are no requests for a particular EJB then the container will probably contain zero or one instance of that component in the memory. If, the need arises then it will increase component instances to satisfy all incoming requests. Then again, if, the number of requests decrease, the container will decrease the component instances in the pool. The most important thing is that the client is absolutely unaware of this component pooling which the container handles.

ii) Resource Management

The container is also responsible for maintaining database connection pools. It provides us a standard way of obtaining and returning database connections. The container also manages EJB environment references and references to other EJBs. The container manages the following types of resources and makes them available to EJBs:

- JDBC 2.0 Data Sources
- JavaMail Sessions
- JMS Queues and Topics
- URL Resources
- Legacy Enterprise Systems via J2EE Connector Architecture

iii) Transaction Management

This is the most important functionality served by the container. A transaction is a single unit of work, composed of one or more steps. If, all the steps are successfully executed, then, the transaction is committed otherwise it is rolled back. There are different types of transactions and it is absolutely unimaginable not to use transactions in today's business environments.

iv) Security

The EJB container provides it's own authentication and authorisation control, allowing only specific clients to interact with the business process. Programmers are not required to create a security architecture of their own, they are provided with a built-in system, all they have to do is to use it.

v) Persistence

Persistence is defined as saving the data or state of the EJB on non J-C volatile media. The container, if, desired can also maintain persistent data of the application. The container is then responsible for retrieving and saving the data for programmers, while taking care of concurrent access from multiple clients and not corrupting the data.

vi) Handling of Multiple Clients

The EJB container handles multiple clients of different types. A JSP based thin client can interact with EJBs with same ease as that of GUI based thick client. The container is smart enough to allow even non-Java clients like COM based applications to interact with the EJB system.

2.6   RESTRICTIONS ON EJB

Enterprise Java Beans have several restrictions, which they must adhere to:

1. They cannot create or manage threads.
2. They cannot access threads using the java.io package.
3. They cannot operate directly with sockets.
4. They cannot load native libraries.
5. They cannot use the AWT to interact with the user.
6. They can only pass objects and values which are compatible with RMI/IIOP.
7. They must supply a public no argument constructor.
8. Methods cannot be static or final.
9. There are more minor restrictions.

But following can be done with EJB:

1. Subclass another class.

2. Interfaces can extend other interfaces, which are descendants of EJBObject or EJBHome.

3. Helper methods can pass any kind of parameters or return types within the EJB.

4. Helper methods can be any kind of visibility.

2.7 DIFFERENCE BETWEEN SESSION BEANS AND ENTITY BEANS

To understand whether a business process or business entity should be implemented as a stateful session bean or an entity bean, it is important to understand the life-cycle and programming differences between them. These differences pertain principally to object sharing, object state, transactions, and container failure and object recovery.
Object sharing pertains to entity objects. Only a single client can use a stateful session bean, but multiple clients can share an entity object among themselves.

The container typically maintains a stateful session bean's object state in the main memory, even across transactions, although the container may swap that state to secondary storage when deactivating the session bean. The object state of an entity bean is typically maintained in a database, although the container may cache the state in memory during a transaction or even across transactions. Other, possibly non-EJB-based, programs can access the state of an entity object that is externalised in the database. For example, a program can run an SQL query directly against the database storing the state of entity objects. In contrast, the state of a stateful session object is accessible only to the session object itself and the container.

The state of an entity object typically changes from within a transaction. Sinces, its state changes transactionally, the container can recover the state of an entity bean should the transaction fail. The container does not maintain the state of a session object transactionally. However, the bean developer may instruct the container to notify the stateful session objects of the transaction boundaries and transaction outcome. These notifications allow the session bean developer to synchronise manually the session object's state with the transactions. For example, the stateful session bean object that caches changed data in its instance variables may use the notification to write the cached data to a database before the transaction manager commits the transaction.

Session objects are not recoverable; that is, they are not guaranteed to survive a container failure and restart. If, a client has held a reference to a session object, that reference becomes invalid after a container failure. (Some containers implement session beans as recoverable objects, but this is not an EJB specification requirement.) An entity object, on the other hand, survives a failure and restart of its container. If, a client holds a reference to the entity object prior to the container failure, the client can continue to use this reference after the container restarts.

The *Table 1* depicts the significant differences in the life cycles of a stateful session bean and an entity bean.

Table 1: Entity Beans and Stateful Session Beans: Life-Cycle Differences

Functional Area	Stateful Session Bean	Entity Bean
Object state	Maintained by the container in the main memory across transactions. Swapped to secondary storage when deactivated.	Maintained in the database or other resource manager. Typically cached in the memory in a transaction.
Object sharing	A session object can be used by only one client.	An entity object can be shared by multiple clients. A client may pass an object reference to another client.
State externalisation	The container internally maintains the session object's state. The state is inaccessible to other programs.	The entity object's state is typically stored in a database. Other programs, such as an SQL query, can access the state in the database.
Transactions	The state of a session object can be synchronised with a transaction but is not recoverable.	The state of an entity object is typically changed transactionally and is recoverable.
Failure recovery	A session object is not guaranteed to survive failure and restart of its container. The references to session objects held by a client becomes invalid after the failure.	An entity object survives the failure and the restart of its container. A client can continue using the references to the entity objects after the container restarts.

2.8 CHOOSING ENTITY BEANS OR STATEFUL SESSION BEANS

Now, we will learn when and where we should use entity beans or stateful session beans. The architect of the application chooses how to map the business entities and processes to enterprise beans. No prescriptive rules dictate whether a stateful session bean or an entity bean should be used for a component: Different designers may map business entities and processes to enterprise beans differently.

We can also combine the use of session beans and entity beans to accomplish a business task. For example, we may have a session bean represent an ATM withdrawal that invokes an entity bean to represent the account.

The following guidelines outline the recommended mapping of business entities and processes to entity and session beans. The guidelines reflect the life-cycle differences between the session and entity objects.

- A bean developer typically implements a business entity as an entity bean.

- A bean developer typically implements a conversational business process as a stateful session bean. For example, developers implement the logic of most Web application sessions as session beans.

- A bean developer typically implements, as an entity bean a collaborative business process: a business process with multiple actors. The entity object's state represents the intermediate steps of a business process that consists of multiple steps. For example, an entity object's state may record the changing information—state—on a loan application as it moves through the steps of the loan-approval process. The object's state may record that the account representative entered the information on the loan application, the loan officer reviewed the application, and application approval is still waiting on a credit report.

- If, it is necessary for any reason to save the intermediate state of a business process in a database, a bean developer implements the business process as an entity bean. Often, the saved state itself can be considered a business entity. For example, many e-commerce Web applications use the concept of a shopping cart, which stores the

items that the customer has selected but not yet checked out. The state of the shopping cart can be considered to be the state of the customer shopping business process. If, it is desirable that the shopping process span extended time periods and multiple Web sessions, the bean developer should implement the shopping cart as an entity bean. In contrast, if the shopping process is limited to a single Web session, the bean developer can implement the shopping cart as a stateful session bean.

2.9   INSTALLING AND RUNNING APPLICATION SERVER FOR EJB

Depending on the requirements of the organisation or company, one may select an appropriate Application server. If organisation's business processes span hundreds or thousands of different computers and servers then, one can select good application servers like BEA Web Logic, IBM Web Sphere and Oracle 9i Application Server etc. which are licensed servers. But, on the other hand, if, the organisation is really small but still wants to use EJBs due to future scalability requirements then, there are quite a few EJB application servers from free and open source to the ones, which are fast, but with reasonably low license fees. Examples of application servers, which are popular in small organisations, include **JBoss** and **Orion** Application Server.

An application server is a conglomeration of software services that provide a runtime environment for any number of containers, as shown in the *Figure 2*. A typical J2EE application server, such as WebLogic, WebSphere, JBoss, and Sun's J2EE Reference Server, houses a multitude of containers. WebLogic, for example, supports an EJB container and a servlet container.
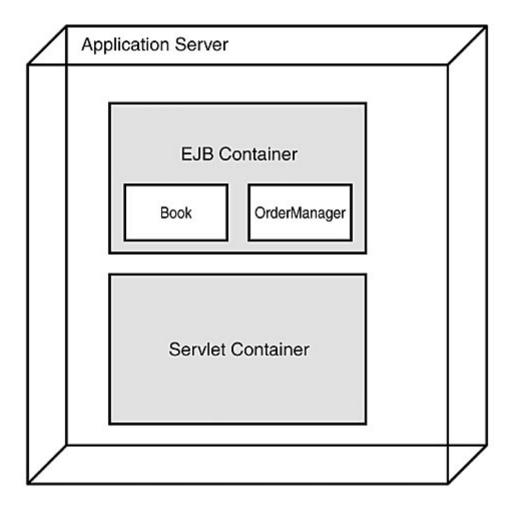

Figure 2: Architecture of EJB container

The EJB container provides basic services, including transactions, life-cycle management, and security, to the EJBs deployed into it. By shouldering much of this burdensome lower-level functionality, the EJB container significantly reduces the responsibilities of the EJBs deployed into it. Because EJBs no longer contain the code to provide these fundamental behaviours, EJB developers are free to concentrate on writing code that solves business problems instead of computer science problems.

Every application server vendor has its own way of deploying EJBs. They all share some common traits, however, that are illustrated in *Figure 3* and described here:

- An EJB's class (or sometimes its source code) files and its deployment descriptor are placed into an archive, which is typically a JAR file. Deployment descriptors are described in more depth in Part II, "Reference".

- A deployment tool of some sort creates a deployable archive file (typically, but not always, a JAR) from the contents of the archive created in Step 1.

- The deployable archive file is deployed into the EJB container by editing the container's configuration file or by running an administrative interface program.
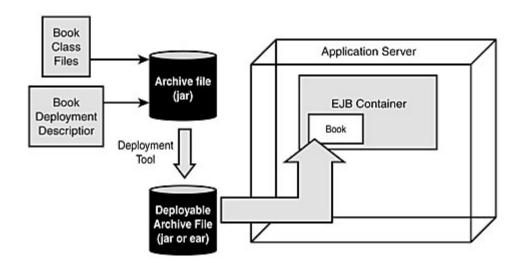

Figure 3: EJBs are typically bundled into archive files, processed by a deployment tool, and then deployed into the EJB container

There are many free application servers like Sun's J2EE Reference Application Server, which is available free at http://www.javasoft.com. Or Jboss, which may be downloaded JBoss from JBoss web site: www.jboss.org. Current stable version is 2.4.3. Download it from their web site. Once you have downloaded it, unzip the JBoss zip file into some directory e.g. C:\JBoss. The directory structure should be something like the following:

```
C:\JBoss
 admin
 bin
 client
 conf
  db
```

```
deploy

lib

log

tmp
```

Now, to start JBoss with default configuration go to JBoss/bin directory and run the following command at the DOS prompt :

```
C:\JBoss\bin>run
```

run.bat is a batch file which starts the JBoss Server. Once JBoss Server starts, you should see huge lines of text appearing on your command prompt screen. These lines show that JBoss Server is starting. Once JBoss startup is complete you should see a message like following one on your screen :

```
[Default] JBoss 2.4.3 Started in 0m:11s
```

Now, we have successfully installed and run JBoss on your system. To stop JBoss, simply press Ctrl + C on the command prompt and JBoss will stop, after displaying huge lines of text.

The client for our EJB will be a JSP page / Java Servelet running in a separate Tomcat Server. We have already learnt in an earlier Block 1, how to create and install Tomcat server for running JSP page or Servlet

Configuring and Running Tomcat :

Create a new folder under the main C:\ drive and name it "Projects". Now, create a new sub-folder in the C:\Projects folder and name it "TomcatJBoss". The directory structure should look like the following:

```
C:\Projects

    TomcatJBoss
```

Now, open conf/Server.xml file from within the Tomcat directory where you have installed it. By default this location will be :

```
C:\Program Files\Apache Tomcat 4.0\conf\server.xml
```

Somewhere in the middle where you can see multiple <Context> tags, add following lines between other <Context> tags :

```
<!-- Tomcat JBoss Context -->

<Context     path="/jboss"     docBase="C:\Projects\TomcatJBoss\"     debug="0"

reloadable= "true" />
```

Now, save Server.xml file. Go to Start -> Programs -> Apache Tomcat 4.0 -> Start Tomcat, to start Tomcat Server. If everything has been setup correctly, you should see the following message on your command prompt:

Starting service Tomcat-Standalone

Apache Tomcat/4.0.1

Note: Creating C:\Projects\TomcatJBoss folder and setting up new /jboss context in Tomcat is NOT required as far as accessing EJBs is concerned. We are doing it only to put our JSP client in a separate folder so as not to intermingle it with your other projects.

⬜ Check Your Progress 1

1) State True or False

_F
a) EJB can manage threads.   T ☐ F ☐

b) In EJB Methods can be static or final.   T ☐ F ☐

c) A transaction is a single work, composed of one or more steps.   T ☐ F ☐

d) The EJB container handles multiple clients of different types.   T ☐ F ☐

2) Describe the layered architecture of EJB and explain all its components briefly.

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

3) Explain the different benefits of EJB architecture to Application Developers and Customers.

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

4) How does the EJB container assist in transaction management and persistence?

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

5) Explain the different types of restrictions on EJB.

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

6) How does the Session Bean differ from the Entity Bean in terms of object sharing, object state and failure recovery?

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

7) What criteria should a developer keep in mind while choosing between a session bean and an entity bean?

…………………………………………………………………………………..…

…………………………………………………………………………………..…

…………………………………………………………………………………

2.10 SUMMARY

In this unit, we have learnt basic architecture of EJB and the various kind of benefits offered by it to application developers and customers. We have also highlighted services offered by EJB, restriction emposed on EJB. The difference between session beans and entitiy beans and choasing between entity beans and session beans. Finally we have discussed how to install and run application server for EJB.

2.11 SOLUTIONS/ ANSWERS

Check Your Progress 1

1)   True/ False

   a) False

   b) False

   c) True

   d) True

Explanatory Answers

2)   EJB is a layered architecture consisting of:

   - Enterprise bean component which contains methods that implements the business logic
   - EJB container
   - EJB server, which contains the EJB container
   - EJB object.

The Enterprise Java Beans Component

An Enterprise JavaBean is a component, just like a traditional JavaBean. Enterprise JavaBeans execute within an EJB container, which in turn executes within an EJB server. Any server that can host an EJB container and provide it with the necessary services can be an EJB server. EJB is a Java class, written by an EJB developer, that implements business logic. All the other classes in the EJB system either support client access to or provide services (like persistence, and so on) to EJB component classes.

The Enterprise Java Beans Container

The EJB container acts as an interface between an Enterprise bean and clients. The client communicates with the Enterprise bean through the remote and home interfaces provided by the customer. The EJB container is where the EJB component "lives." Since the EJB container handles all these functions, the EJB component developer can concentrate on business rules, and leave database manipulation and other such fine details to the container. The EJB container provides services such as security, transaction and resource management, versioning, scalability, mobility, persistence, and security to the EJB components it contains.

EJB Server

The EJB server provides some low level services like network connectivity to the container. In addition to this, it also provides the following services:

- Instance Passivation

- Instance pooling

- Database Connection Pooling

- Preached Instances

The EJB Object and the Remote Interface

Client programs execute methods on remote EJBs by way of an EJB object. The EJB object implements the "remote interface" of the EJB component on the server. The remote interface represents the "business" methods of the EJB component. The remote interface does the actual, useful work of an EJB object, such as creating an order form or referring a patient to a specialist.

EJB objects and EJB components are separate classes, though from the outside (i.e., by looking at their interfaces), they look identical. This is because, they both implement the same interface (the EJB component's remote interface), but they do very different things. An EJB component runs on the server in an EJB container and implements the business logic. *The EJB object runs on the client and remotely executes the EJB component's methods.*

3) The EJB component architecture is the backbone of the J2EE platform. The EJB architecture provides the following benefits to the application developer:

- **Simplicity:** It is easier to develop an enterprise application with EJB architecture than without it. Since, EJB architecture helps the application developer access and use enterprise services with minimal effort and time, writing an enterprise bean is almost as simple as writing a Java class.

- **Application Portability:** An EJB application can be deployed on any J2EE-compliant server. Application developer can choose the "best-of-breed" application server that meets their requirements.

- **Component Reusability:** An EJB application consists of enterprise bean components and each of the enterprise Bean is a reusable building block. Business logic of enterprise beans can be reused through Java sub classing of the enterprise bean class.

- **Ability to Build Complex Applications:** EJB architecture simplifies building complex enterprise applications. The component-based EJB architecture is well suited to the development and maintenance of complex enterprise applications.

- **Separation of Business Logic from Presentation Logic:** An enterprise bean typically encapsulates a business process or a business entity (an object representing enterprise business data), making it independent of the presentation logic.

- **Easy development of Web Services:** The Web services features of the EJB architecture provide an easy way for Java developers to develop and access Web services. The tools provided by the container manage the mapping to the Web services standards.

- **Deployment in many Operating Environments:** The EJB architecture allows the bean developer to separate common application business logic from the customisation logic performed at deployment to me. The EJB architecture allows an entity bean to be bound to different database schemas. The EJB architecture facilitates the deployment of an application by establishing deployment standards, such as those for data source lookup, other application dependencies, security configuration, and so forth.

- **Distributed Deployment:** The EJB architecture makes it possible for applications to be deployed in a distributed manner across multiple servers on a network.

- **Application Interoperability:** The EJB architecture makes it easier to integrate applications from different vendors. The enterprise bean's client-view interface serves as a well-defined integration point between applications.

- **Integration with non-Java systems:** The related J2EE APIs, such as the J2EE Connector specification and the Java Message Service (JMS) specification, and J2EE Web services technologies, such as the Java API for XML-based RPC (JAX-RPC), make it possible to integrate enterprise bean applications with various non-Java applications, such as ERP systems or mainframe applications, in a standard way.

- **Educational Resources and Development Tools:** Because EJB architecture is an industry wide standard, the EJB application developer benefits from a growing body of educational resources on how to build EJB applications.

Benefits to Customers

A customer's perspective on EJB architecture is different from that of the application developer. The EJB architecture provides the following benefits to the customer:

- **Choice of Server:** Because the EJB architecture is an industry wide standard and is part of the J2EE platform, customer organisations have a wide choice of J2EE-compliant servers. Customers can select a product that meets their needs in terms of scalability, integration capabilities with other systems, security protocols, price, and so forth.

- **Facilitation of Application Management:** Sophisticated application management tools provided with the EJB container allow the customer's IT department to perform such functions as starting and stopping the application, allocating system resources to the application, and monitoring security violations, among others.

- **Integration with a Customer's Existing Applications and Data:** EJB architecture and the other related J2EE APIs simplify and standardize the integration of EJB applications with any non-Java applications and systems at the customer operational environment.

- **Integration with Enterprise Applications of Customers, Partners, and Suppliers:** The interoperability and Web services features of the EJB architecture allow EJB-based applications to be exposed as Web services to other enterprises.

- **Application Security:** EJB architecture shifts most of the responsibility for an application's security from the application developer to the server vendor, system administrator, and the deployer.

4) EJB container offers various kinds of services. Transaction Management is the most important functionality served by the container. The container provides the support according to the transaction specified and we can specify following types of transaction support for the bean:

- The bean manages its own transaction

- The bean doesn't support any transaction.

- When the client invokes the bean's method and if, the client has a transaction in progress, then the bean runs within the client's transaction. Otherwise, the container starts a new transaction for the bean.

- The bean must always start a new transaction, even if a transaction is open.

- The bean requires the client to have a transaction open before the client invokes the bean's method.

Persistence is another important function, which is supported by the EJB container. Persistence is the permanent storage of state of an object in a non-volatile media. This allows an object to be accessed at any time, without having to recreate the object every time it is required. The container if desired can also maintain persistent data of the application.

5) Enterprise Java Beans have several restrictions, which they must adhere to:

   1. They cannot create or manage threads.
   2. They cannot access threads using the java.io package.
   3. They cannot operate directly with sockets.
   4. They cannot load native libraries.
   5. They cannot use the AWT to interact with the user.
   6. They can only pass objects and values, which are compatible with RMI/IIOP.
   7. They must supply a public no argument constructor.
   8. Methods cannot be static or final.
   9. There are more minor restrictions.

6) Session Beans and Entity beans differ in respect to object sharing, object state, transactions and container failure and object recovery.

   Object Sharing: A session object can be used by only one client whereas, an entity object can be shared by multiple clients. A client may pass an object reference to another client.

   Object State: Object state of stateful session bean is maintained by the container in the main memory across transactions and swapped to secondary storage when deactivated whereas, the object state of an entity bean is maintained in the database or other resource manager and typically cached in the memory in a transaction.

   Failure Recovery: A session object is not guaranteed to survive failure and restart of its container. The references to session objects held by a client become invalid after the failure whereas, an entity object survives the failure and the restart of its container. A client can continue using the references to the entity objects after the container restarts.

7) There could be various guidelines, which can assist application developer choose between a session bean and entity bean, which are described below:

   - A bean developer may typically implement a business entity as an entity bean.

   - A bean developer may typically implement a conversational business process as a stateful session bean. For example, developers implement the logic of most Web application sessions as session beans.

   - A bean developer typically implements as an entity bean a collaborative business process: a business process with multiple actors.

   - If, it is necessary for any reason to save the intermediate state of a business process in a database, a bean developer implements the business process as an entity bean. Often, the saved state itself can be considered a business entity.

2.12 FURTHER READINGS/REFERENCES

- Paco Gomez and Peter Zadronzy, *Professional Java 2 Enterprise Edition with BEA Weblogic Server,* WROX Press Ltd

- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,* New Riders Publishing

- Richard Monson-Haefel, *Enterprise JavaBeans (3rd Edition),* O'Reilly

- Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns, *Applying Enterprise JavaBeans™:Component-Based Development for the J2EE™ Platform,* Second Edition, Pearson Education

- Robert Englander, *Developing Java Beans,* O'Reilly Media

Reference websites:

- www.j2eeolympus.com
- www.phptr.com
- www.samspublishing.com
- www.oreilly.com
- www.roseindia.net
- www.caucho.com
- www.tutorialized.com
- www.stardeveloper.com

UNIT 3 EJB: DEPLOYING ENTERPRISE JAVA BEANS

3.0 INTRODUCTION

In the previous unit, we have learnt the basics of Enterprise java beans and different type of java beans. In this unit we shall learn how to create and deploy Enterprise java beans. "Enterprise JavaBeans" (EJBs) are distributed network aware components for developing secure, scalable, transactional and multi-user components in a J2EE environment. EJBs are a collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. As we have learnt, JSP Servlets have their own web container and run inside that container. Similarly, EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function. Now, we will learn how to create our first Enterprise JavaBean and we will then deploy this EJB on a production class, open source, and free EJB Server; JBoss.

3.1 OBJECTIVES

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

- provide an overview of XML;

- discuss how XML differs from HMTL;

- understand the SGML and its use;

- analyse the difference between SGML and XML;

- understand the basic goals of development of XML;

- explain the basic structure of XML document and its different components;

- understand what DTD is and how do prepare DTD for an XML document;

- learn what an XML parser is and its varieties, and

- differentiate between the different types of entities and their uses.

3.2   DEVELOPING THE FIRST SESSION EJB

We have already learnt in the previous unit about Session EJBs which are responsible for maintaining business logic or processing logic in our J2EE applications. Session beans are the simplest beans and are easy to develop so, first, we will develop Session beans. Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are :

- **Remote Interfaces:** Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

- **Home Interfaces:** Home interface is actually EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, the Home interface must contain at least one create() method.
  The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

The XML file will be named as "ejb-jar.xml" and will be used to configure the EJBs during deployment. So, in essence our EJB, FirstEJB, which we are going to create consists of the following files :

Ejb

> session

>> First.java

>> FirstHome.java

>> FirstEJB.java

META-INF

> ejb-jar.xml

"First.java" will be the Remote interface we talked about. "FirstHome.java" is our Home interface and "FirstEJB.java" is the actual EJB class file. The ejb-jar.xml deployment descriptor file goes in the META-INF folder.

Now, create a new folder under the C:\Projects folder which we had created earlier, and name it "EJB". Now, create a new sub-folder under C:\Projects\EJB folder and name it "FirstEJB". Create a new folder "src" for Java source files in the "FirstEJB' folder. The directory structure should look like the following :

```
C:\Projects
   TomcatJBoss
   EJB
     FirstEJB
       Src
```

Now, create directory structure according to the package ignou.mca.ejb.session in the "src" folder. If, you know how package structure is built, it shouldn't be a problem. Anyhow, the final directory structure should like the following :

```
C:\Projects
   TomcatJBoss
```

```
EJB
    FirstEJB
      src
        ignou
          mca
            ejb
              session
```

First.java :

Now, we shall create the First.java source file in ignou/mca/ejb/session folder. Type the following code in it:

```
/* First.java */

package ignou.mca.ejb.session;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface First extends EJBObject {

    public String getTime() throws RemoteException;
}
```

Now save this file as first.java

First line is the package statement which indicates that the first interface belongs to the ignou.mca.ejb.session package:

```
package com.stardeveloper.ejb.session;
```

The next two lines are import statements for importing the required classes from the packages.

import javax.ejb.EJBObject;

import java.rmi.RemoteException;

Then, comes the interface declaration line which depicts that this is an interface with name "First" which extends an existing interface javax.ejb.EJBObject.

```
Note:  Every Remote interface *must* always extend EJBObject interface. It is a
        requirement, not an option.
```

public interface First extends EJBObject {

Now, we have to declare methods in Remote interface which we want to be called for the client (which the client can access and call). For simplicity, we will only declare a single method, getTime(); which will return a String object containing current time.

```
public String getTime() throws RemoteException;
```

We should notice that there are no {} parenthesis in this method as has been declared in an interface. Remote interface method's must also throw RemoteException because EJBs are distributed components and during the call to an EJB, due to some network problem, exceptional events can arise, so all Remote interface method's must declare that they can throw RemoteException in Remote interfaces.

FirstHome.java :

Now, let us create the home interface for our FirstEJB. Home interface is used to create and get access to Remote interfaces. Create a new FirstHome.java source file in ignou/mca/ejb/session package. Type the following code in it:

```
/* FirstHome.java */
package com.stardeveloper.ejb.session;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface FirstHome extends EJBHome {

    public First create() throws CreateException, RemoteException;
}
```

Now, save this file as FirstHome.java.

First few lines are package and import statements. Next, we have declared our FirstHome interface which extends javax.ejb.EJBHome interface.

Note: All Home interfaces *must* extend EJBHome interface.

```
public interface FirstHome extends EJBHome {
```

Then, we declared a single create() method which returns an instance of First Remote interface. Notice that all methods in Home interfaces as well must also declare that they can throw RemoteException. One other exception that must be declared and thrown is CreateException.

```
public First create() throws CreateException, RemoteException;
```

We are done with creating Remote and Home interfaces for our FirstEJB. These two are the only things which our client will see, the client will remain absolutely blind as far as the actual implementation class, FirstEJB is concerned.

FirstEJB.java :

FirstEJB is going to be our main EJB class. Create a new FirstEJB.java source file in

ignou/mca/ejb/session folder. Type the following code in it:

```
/* FirstEJB.java */
package ignou.mca.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import java.util.Date;

public class FirstEJB implements SessionBean {

    public String getTime() {
        return "Time is : " + new Date().toString();
    }
    public void ejbCreate() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext context) {}
}
```

Now, save this file as FirstEJB.java.

The first few lines are again the package and import statements. Next, we have declared our FirstEJB class and made it implement javax.ejb.SessionBean interface.

Note: All Session bean implementation classes must implement SessionBean interface.

public class FirstEJB implements SessionBean

Our first method is getTime() which had been declared in our First Remote interface. We implement that method here in our FirstEJB class. It simply returns get date and time as can be seen below:

```
public String getTime() {

    return "Time is : " + new Date().toString();

}
```

Then comes 5 callback methods which are part of SessionBean interface and since we are implementing SessionBean interface, we have to provide empty implementations of these methods.

ejb-jar.xml :

Let's now create the EJB deployment descriptor file for our FirstEJB Session bean. Create a new ejb-jar.xml file in the FirstEJB/META-INF folder. Copy and paste the following text in it:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
   Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
```

```
      <description></description>
    <enterprise-beans>
      <session>
        <display-name>FirstEJB</display-name>
        <ejb-name>First</ejb-name>
        <home>ignou.mca.ejb.session.FirstHome</home>
        <remote>ignou.mca.ejb.session.First</remote>
        <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
    </enterprise-beans>

    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>First</ejb-name>
          <method-name>*</method-name>
        </method>
        <trans-attribute>Supports</trans-attribute>
      </container-transaction>

      <security-role>
        <description>Users</description>
        <role-name>users</role-name>
      </security-role>
    </assembly-descriptor>
</ejb-jar>
```

One or more EJBs are packaged inside a JAR (.jar) file. There should be only one ejb-jar.xml file in an EJB JAR file. So ejb-jar.xml contains deployment description for one or more than one EJBs. Now as we learned in an earlier unit, there are 3 types of EJBs so ejb-jar.xml should be able to contain deployment description for all 3 types of EJBs.

Our ejb-jar.xml file for FirstEJB contains deployment description for the only EJB we have developed; FirstEJB. Since it is a Session bean, it's deployment description is contained inside <session></session> tags.

```
<ejb-jar>
  <description></description>
  <enterprise-beans>
    <session></session>
  </enterprise-beans>
</ejb-jar>
```

Now, let us discuss different deployment descriptor tags inside the <session></session> tag. First is the <ejb-name> tag. The value of this tag should be name of EJB i.e. any name you think should point to your Session EJB. In our case it's value is "First". Then, comes <home>, <remote> and <ejb-class> tags which contain complete path to Home, Remote and EJB implementation classes. Then comes <session-type> tag whose value is either "Stateless" or "Stateful". In our case it is "Stateless" because our Session bean is stateless. Last tag is <transaction-type>, whose value can be either "Container" or "Bean". The Transactions for our FirstEJB will be managed by the container.

```
<ejb-jar>
  <description></description>
```

```
  <enterprise-beans>
    <session>
      <display-name>FirstEJB</display-name>
      <ejb-name>First</ejb-name>
      <home>ignou.mca.ejb.session.FirstHome</home>
      <remote>ignou.mca.ejb.session.First</remote>
      <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Then there is a <container-transaction> tag, which indicates that all methods of FirstEJB "support" transactions.

Compiling the EJB Java Source Files :

Now, our directory and file structure till now looks something like the following:

```
C:\Projects
   TomcatJBoss
   EJB
     FirstEJB
       src
         ignou
           mca
             ejb
               session
                   First.java
                   FirstHome.java
                   FirstEJB.java
        META-INF
          ejb-jar.xml
```

We can compile all the Java source file by using a command like the following on the command prompt:

```
C:\Projects\EJB\FirstEJB\src\ignou\mca\ejb\session>
        javac -verbose -classpath %CLASSPATH%;C:\JBoss\client\jboss-j2ee.jar
    -d C:\Projects\EJB\FirstEJB *.java
```

Note: The point to remember is to make sure jboss-j2ee.jar (which contains J2EE package classes) in the CLASSPATH, or you will get errors when trying to compile these classes.

If, you have installed JBoss Server in a separate directory then, substitute the path to jboss-j2ee.jar with the one present on your system. The point is to put jboss-j2ee.jar in the CLASSPATH for the javac, so that all EJB source files compile successfully.

3.3 PACKAGING EJB SOURCE FILES INTO A JAR FILE

Till now our directory and file structure should look something like the following:

```
C:\Projects
  TomcatJBoss
  EJB
    FirstEJB
      ignou
        mca
          ejb
            session
              First.class
              FirstHome.class
              FirstEJB.class
      META-INF
        ejb-jar.xml
      src
        ignou
          mca
            ejb
              session
                First.java
                FirstHome.java
                FirstEJB.java
```

Now, to package the class files and XML descriptor files together, we will use the concept of jar file, which act as a container. For creating the jar file, run the following command at the command prompt:

```
C:\Projects\EJB\FirstEJB>jar cvfM FirstEJB.jar com META-INF
```

After running this command, EJB JAR file will be created with the name FirstEJB.jar in the FirstEJB folder. After this Jboss configuration file is to be added.

Adding JBoss specific configuration file :

Till now our FirstEJB.jar file contains generic EJB files and deployment description. To run it on JBoss we will have to add one other file into the META-INF folder of this JAR file, called jboss.xml. This file contains the JNDI mapping of FirstEJB.
So, create a new jboss.xml file in the FirstEJB/META-INF folder where ejb-jar.xml file is present and type the following text in it:

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
        "http://www.jboss.org/j2ee/dtd/jboss.dtd">
<jboss>
        <enterprise-beans>
                <session>
                        <ejb-name>First</ejb-name>
                        <jndi-name>ejb/First</jndi-name>
                </session>
        </enterprise-beans>
</jboss>
```

Now, we will add this new jboss.xml file into our existing FirstEJB.jar file by running the
following command at the DOS prompt:

```
C:\Projects\EJB\FirstEJB>jar uvfM FirstEJB.jar META-INF
```

Now, we will deploy the FirstEJB.jar on the JBoss Server.

3.4   DEPLOYING JAR FILE ON JBOSS SERVER

Now, we will deploy the FirstEJB.jar file on the Jboss Server by copying the FirstEJB.jar
file and pasting it into the C:\JBoss\deploy folder. If JBoss is running, you should see text
messages appearing on the console that this EJB is being deployed and finally deployed
and started.

Creating the Client JSP Page:

Now we will create a new WEB-INF folder in C:\Projects\TomcatJBoss folder. Then
create a new web.xml file and type the following text in it :

```xml
<?xml version= "1.0" encoding= "ISO-8859-1"?>

<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2.3.dtd">

<web-app>
</web-app>
```

As we can see, this web.xml is almost empty and is not doing anything useful. We still
created it because Tomcat will throw an error if you try to access this /jboss context that
we had created earlier without creating a /WEB-INF/web.xml file.

FirstEJB.jsp JSP Client page :

We will now create a new JSP page in the C:\Projects\TomcatJBoss folder and save it as
"firstEJB.jsp". Now type the code in it :

```jsp
<%@ page import= "javax.naming.InitialContext,
          javax.naming.Context,
          java.util.Properties,
          ignou.mca.ejb.session.First,
          ignou.mca.ejb.session.FirstHome"%>
<%
   long t1 = System.currentTimeMillis();
   Properties props = new Properties();
     props.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");
     props.put(Context.PROVIDER_URL, "localhost:1099");

   Context ctx = new InitialContext(props);
   FirstHome home = (FirstHome)ctx.lookup("ejb/First");
   First bean = home.create();
   String time = bean.getTime();
   bean.remove();
```

```
   ctx.close();
   long t2 = System.currentTimeMillis();
%>
<html>
<head>
   <style>p { font-family:Verdana;font-size:12px; }</style>
</head>
<body>
<p>Message received from bean = "<%= time %>".<br>Time taken :
   <%= (t2 - t1) %> ms.</p>
</body>
</html>
```

Now, save this file as firstEJB.jsp as JSP page.

As we can see, the code to connect to an *external* JNDI/EJB Server is extremely simple. First, we have created a Properties object and put certain values for Context.INITIAL_CONTEXT_FACTORY and Context.PROVIDER_URL properties.

The value for Context.INITIAL_CONTEXT_FACTORY is the interface provided by JBoss and the value for Context.PROVIDER_URL is the location:port number where JBoss is running. Both these properties are required to connect to an external JNDI Server.

```
   Properties props = new Properties();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.PROVIDER_URL, "localhost:1099");
```

We then get hold of that external JNDI Context object by creating a new InitialContext() object, it's argument being the Properties object we had created earlier.

```
Context ctx = new InitialContext(props);
```

Next, we have used that external JNDI Context handle to lookup our FirstEJB running on JBoss. Notice that the argument to Context.lookup("ejb/First") is the same value we had put in the jboss.xml file to bind our FirstEJB to this name in the JNDI context. We have used that same value again to look for it. Once our lookup is successful, we cast it to our FirstHome Home interface.

```
   FirstHome home = (FirstHome)ctx.lookup("ejb/First");
```

We have then used create method of our FirstHome Home interface to get an instance of the Remote interface; First. We will now use the methods of our EJB's remote interface (First).

```
   First bean = home.create();
```

We then called the getTime() method we had created in our EJB to get the current time from JBoss Server and saved it in a temporary String object.

```
String time = bean.getTime();
```

52

Once we are done with our Session EJB, we use the remove method to tell the JBoss Server that we no longer need this bean instance. After this we close the external JNDI Context.

```
bean.remove();
ctx.close();
```

After this retrieved value from getTime() method displayed on the user screen.

3.5   RUNNING THE CLIENT/JSP PAGE

Before trying to run firstJSP.jsp page, we have to do one other thing. Copy following files from C:\JBoss\client folder and paste them in the C:\Projects\TomcatJBoss\WEB-INF\lib folder. It is a must, without it firstEJB.jsp page will not run.

```
connector.jar
deploy.jar
jaas.jar
jboss-client.jar
jboss-j2ee.jar
jbossmq-client.jar
jbosssx-client.jar
jndi.jar
jnp-client.jar
```

we will also copy the FirstEJB.jar file from C:\Projects\EJB\FirstEJB folder to the C:\Projects\TomcatJBoss\WEB-INF\lib folder. Without it the Tomcat will not be able to compile firstEJB.jsp JSP page.

We are now ready to run our firstEJB.jsp page

Running firstEJB.jsp JSP page :

Now, start JBoss Server if, it is not already running. Also start Tomcat Server. If it is already running, then stop it and restart it again. Open your browser and access the following page :

```
http://localhost:8080/jboss/firstEJB.jsp
```

Please substitute port number "8080" above with the port number where your Tomcat Server is running. By default it is "8080" you will see a result like the following :

Figure 1: firstEJB.jsp Client View

3.6   MESSAGE-DRIVEN BEAN

Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java™ API for XML Messaging (JAX-M). Message-driven beans asynchronously consume messages from a message destination, such as a JMS queue or topic.

Message-driven beans are components that receive incoming enterprise messages from a messaging provider. The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages. A message-driven bean's business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message.

A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client—acts as a message producer and sends a message that is delivered to a message destination. The container activates an instance of the correct type of message-driven bean from its pool of message-driven beans, and the bean instance consumes the message from the message destination. As a message-driven bean is stateless, any instance of the matching type of message-driven bean can process any message. Thus, message-driven beans are programmed in a similar manner as stateless session beans.

The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

3.7   IMPLEMENTING A MESSAGE-DRIVEN BEAN

Now, we will learn how to implement message driven beans. Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. As they do not expose a component or home interface, clients cannot directly access message-driven beans. Like session beans, message-driven beans may be used to drive workflow processes. However, the arrival of a particular message initiates the process.

Implementing a message-driven bean is fairly straightforward. A message-driven bean extends two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

We can provide an empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references. If the message-driven bean sends messages or receives synchronous communication from another destination, you use the ejbCreate method to look up the JMS connection factories and destinations and to create the JMS connection. The implementation of the ejbRemove method can also be left empty. However, if the ejbCreate method obtained any resources, such as a JMS connection, you should use the ejbRemove method to close those resources.

The methods of the message listener interface are the principal methods of interest to the developer. These methods contain the business logic that the bean executes upon receipt of a message. The EJB container invokes these methods defined on the message-driven bean class when a message arrives for the bean to service.

A developer decides how a message-driven bean should handle a particular message and codes this logic into the listener methods. For example, the message-driven bean might simply pass the message to another enterprise bean component via a synchronous method invocation, send the message to another message destination, or perform some business logic to handle the message itself and update a database.

A message-driven bean can be associated with configuration properties that are specific to the messaging system it uses. A developer can use the bean's XML deployment descriptor to include the property names and values that the container can use when connecting the bean with its messaging system.

3.8 JMS AND MESSAGE-DRIVEN BEANS

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. You can think of message-driven beans as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. When the destination is a queue, there is only one message producer, or sender, and one message consumer. When the destination is a topic, a message producer publishes messages to the topic, and any number of consumers may consume the topic's messages.

A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

Because the method does not include a throws clause, no application exceptions may be thrown during processing.

The EJB architecture defines several configuration properties for JMS-based message-driven beans. These properties allow the container to appropriately configure the bean and link it to the JMS message provider during deployment. These properties include the following:

- **DestinationType:** Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.

- **SubscriptionDurability:** Used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.

- **AcknowledgeMode:** When message-driven beans use bean-managed transactions, this property indicates to the container the manner in which the delivery of JMS messages is to be acknowledged. (When beans use container-managed transactions, the message is acknowledged when the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge. The Auto-acknowledge mode indicates that the container should acknowledge messages as soon as the onMessage method returns. The Dups-ok-acknowledge mode indicates that the

container may lazily acknowledge messages, which could cause duplicate messages to be delivered.

- **MessageSelector**— Allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

3.9   MESSAGE-DRIVEN BEAN AND TRANSACTIONS

As messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction. Keep in mind, however, that if the message-driven bean participates in a transaction, you must also be using container-managed transaction demarcation. The deployment descriptor transaction attribute, which for a message-driven bean can be either Required or NotSupported, determines whether the bean participates in a transaction.

When a message-driven bean's transaction attribute is set to Required, the message delivery from the message destination to the message-driven bean is part of the subsequent transactional work undertaken by the bean. By having the message-driven bean be part of a transaction, you ensure that message delivery takes place. If the subsequent transaction fails, the message delivery is rolled back along with the other transactional work. The message remains available in the message destination until picked up by another message-driven bean instance. Note that, the message sender and the message receiver, which is the message-driven bean, do not share the same transaction. Thus, the sender and the receiver communicate in a loosely coupled but reliable manner.

If the message-driven bean's transactional attribute is NotSupported, it consumes the message outside of any subsequent transactional work. Should that transaction not complete, the message is still considered consumed and will be lost.

It is also possible to use bean-managed transaction demarcation with a message-driven bean. With bean-managed transaction demarcation, however, the message delivery is not be part of the transaction, because the transaction starts within the onMessage method.

3.10 MESSAGE-DRIVEN BEAN USAGE

Bean developers should consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.

- To implement asynchronous messaging.

- To integrate applications in a loosely coupled but reliable manner.

- To have message delivery drive other events in the system workflow.

- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.11 EXAMPLE OF MESSAGE-DRIVEN BEAN

Now, we will learn how to write a message driven bean. Consider,PayrollMDB, a message-driven bean that follows the requirements of the EJB 2.1 architecture, consisting

of the PayrollMDB class and associated deployment descriptors. The following Code shows the complete code for the PayrollMDB implementation class:

```
Public class PayrollMDB implements MessageDrivenBean,
      MessageListener {

  private PayrollLocal payroll;

  public void setMessageDrivenContext(MessageDrivenContext mdc) {
    try {
      InitialContext ictx = new InitialContext();
      PayrollLocalHome payrollHome = (PayrollLocalHome)
          ictx.lookup("java:comp/env/ejb/PayrollEJB");
      payroll = payrollHome.create();
    } catch ( Exception ex ) {
      throw new EJBException("Unable to get Payroll bean", ex);
    }
  }

  public void ejbCreate() { }

  public void ejbRemove() { }

  public void onMessage(Message msg) {
    MapMessage map = (MapMessage)msg;
    try {
      int emplNumber = map.getInt("Employee");
      double deduction = map.getDouble("PayrollDeduction");

      payroll.setBenefitsDeduction(emplNumber, deduction);
    } catch ( Exception ex ) {
      throw new EJBException(ex);
    }
  }
}
```

The PayrollMDB class implements two interfaces: javax.ejb.MessageDrivenBean and javax.jms.MessageListener. The JMS MessageListener interface allows the bean to receive JMS messages with the onMessage method. The MessageDrivenBean interface defines a message-driven bean's life-cycle methods called for, by the EJB container.

Of three such life-cycle methods, only one is of interest to PayrollMDB— setMessageDrivenContext. The container calls for setMessageDrivenContext immediately after the PayrollMDB bean instance is created. PayrollMDB uses this method to obtain a local reference to the Payroll stateless session bean, first looking up the PayrollLocalHome local home object and then invoking its create method. The setMessageDrivenContext method then stores the local reference to the stateless bean in the instance variable payroll for later use in the onMessage method.

The ejbCreate and ejbRemove life-cycle methods are empty. They can be used for any initialisation and cleanup that the bean needs to do.

The real work of the PayrollMDB bean is done in the onMessage method. The container calls the onMessage method when a JMS message is received in the PayrollQueue queue. The msg parameter is a JMS message that contains the message sent by the Benefits Enrollment application or other enterprise application. The method typecasts the received message to a JMS MapMessage message type, which is a special JMS message type that

contains property-value pairs and is particularly useful when receiving messages sent by non-Java applications. The EJB container's JMS provider may convert a message of MapMessage type either from or to a messaging product–specific format.

Once the message is in the proper type or format, the onMessage method retrieves the message data: the employee number and payroll deduction amount, using the Employee and PayrollDeduction properties, respectively. The method then invokes the local business method setBenefitsDeduction on the Payroll stateless session bean method to perform the update of the employee's payroll information in PayrollDatabase.

The PayrollMDB bean's deployment descriptor declares its transaction attribute as Required, indicating that the container starts a transaction before invoking the onMessage method and to make the message delivery part of the transaction. This ensures that the Payroll stateless session bean performs its database update as part of the same transaction and that message delivery and database update are atomic. If, an exception occurs, the transaction is rolled back, and the message will be delivered again. By using the Required transaction attribute for the message-driven bean, the developer can be confident that the database update will eventually take place.

PayrollEJB Local Interfaces

In the PayrollMDB means we have noticed that it uses the local interfaces of the PayrollEJB stateless session bean. These interfaces provide the same functionality as the remote interfaces described earlier. Since, these interfaces are local, they can be since accessed only by local clients deployed in the same JVM as the PayrollEJB bean. As a result, the PayrollMDB bean can use these local interfaces because it is deployed together with the PayrollEJB bean in the payroll department's application server. The following is the code for the local interfaces :

```
public interface PayrollLocal extends EJBLocalObject {
    void setBenefitsDeduction(int emplNumber, double deduction)
        throws PayrollException;
    double getBenefitsDeduction(int emplNumber)
        throws PayrollException;
    double getSalary(int emplNumber)
        throws PayrollException;
    void setSalary(int emplNumber, double salary)
        throws PayrollException;
}

public interface PayrollLocalHome extends EJBLocalHome {
    PayrollLocal create() throws CreateException;

}
```

🗐 Check Your Progress 1

1) State True/ False

   a) By default Home interface must contain at least one create() method. T ☐ F ☐

   b) Message-driven beans always synchronously consume messages from a message destination, such as a JMS queue.   T ☐ F ☐

   c) The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface.   T ☐ F☐

d) A message-driven bean that consumes JMS messages is not always required to implement the javax.jms.MessageListener interface.   T ☐ F ☐

2) Explain two interfaces associated with EJB class files.

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

3) Explain the Message Driven bean and its advantages.

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

4) Explain the methods required to implement the Message Driven Bean.

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

5) How does Message Driven Bean consumes the messages from the JMS?

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

6) Explain the various configuration properties of JMS based Message Driven Beans offered by EJB architecture.

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

7) Explain the various circumstances under which Message Driven Bean can be used ?

……………………………………………………………………………..…

……………………………………………………………………………..…

………………………………………………………………………………

3.12 SUMMARY

In this unit we have learnt what comprises an Enterprise JavaBean and how to develop and deploy an EJB. We created an EJB, deployed it on JBoss Server and called it from a JSP page running on Tomcat Server in a separate process. EJBs are collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function.

Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are Remote Interfaces and home interface. Remote interface contains the methods that developer wants to expose to the clients. Home interface is actually EJB builder and should contain methods used to create Remote interfaces for the EJB. By default Home interface must contain at least one create() method. The actual implementation remains hidden from the clients.

Every Remote interface must always extend EJBObject interface. It is a requirement, not an option. Message driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging

system, such as the Java API for XML Messaging (JAX-M). Message-driven beans are components that receive incoming enterprise messages from a messaging provider. Message-driven beans contain business logic for handling received messages. A message-driven bean's business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message. The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus reducing the dependencies between separate components.

Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. Message-driven beans act as message listeners that consume messages from a JMS destination. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class.

Messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction.

Message driven can be used under the various circumstances like when messages to automatically delivered, to implement asynchronous messaging and to create message selectors.

3.13 SOLUTIONS/ ANSWERS

Check Your Progress 1

1)   True/ False

a) True
b) False
c) True
d) False

Explanatory Answers

2)   There are two interfaces associated with every EJB class file :

Remote Interfaces: Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

Home Interfaces: Home interface is actually the EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, Home interface must contain at least one create() method.

The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

3) Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java API for XML Messaging (JAX-M). The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages.

    A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client— acts as a message producer and sends a message that is delivered to a message destination.

    Advantages

    The advantage of message-driven beans is that they allow loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

4) The message-driven bean must implements two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

    There can be empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references.

5) Message-driven beans act as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

6) The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

7) The EJB architecture defines several configuration properties for JMS-based message-driven beans, which are described as the following:

    DestinationType: Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.

SubscriptionDurability: It is used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.

AcknowledgeMode: When message-driven beans use bean-managed transactions, this property indicates to the container the process of acknowledging the delivery of JMS messages. (When beans use container-managed transactions, the message is acknowledged once the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge.

MessageSelector: It allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

Bean developers may consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.
- To implement asynchronous messaging.
- To integrate applications in a loosely coupled but reliable manner.
- To have message delivery drive other events in the system workflow.
- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.14 FURTHER READINGS/REFERENCES

- Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible,* Hungry Minds, Inc.

- Paco Gomez and Peter Zadronzy, *Professional Java 2 Enterprise Edition with BEA Weblogic Server,* WROX Press Ltd

- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,* New Riders Publishing

- Richard Monson-Haefel, *Enterprise JavaBeans (3rd Edition),* O'Reilly

- Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns, *Applying Enterprise JavaBeans™:Component-Based Development for the J2EE™ Platform,* Second Edition, Pearson Education

- Robert Englander, *Developing Java Beans,* O'Reilly Media

Reference websites:

- www.javaworld.com
- www.j2eeolympus.com
- www.samspublishing.com
- www.oreilly.com
- www.stardeveloper.com
- www.roseindia.net
- www.e-docs.bea.com
- www.java.sun.com

UNIT 4   XML : EXTENSIBLE MARKUP LANGUAGE

Structure

4.0   INTRODUCTION

In the previous blocks, we have already learnt about Java Servlets, Enterprise Java Beans and Java Server pages. In this unit, we shall cover some basics aspects of XML, which stands for Extensible Markup language, and SGML. XML is a structured document containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of the role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure. SGML, stands for both a language and an ISO standard for describing information, embedded within a document. XML is a meta-language written in SGML that allows one to design a markup language, used for the *easy interchange of documents* on the World Wide Web. In this unit, we shall also learn to discuss the differences between XML, SGML, and DTD, which stands for Document Type Definition and the requirements of DTD for the XML document.

4.1   OBJECTIVES

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

- provide an overview of XML;
- distinguish between XML and HTML;
- define SGML and its use;
- discuss the difference between SGML and XML;
- understand the basic goals of the development of XML;
- define the basic structure of the XML document and its different components;
- define DTD and the method of preparing DTD for an XML document;
- understand what is XML parser and its varieties, and
- understand the different types of entities and their uses.

4.2 AN OVERVIEW OF XML

XML stands for Extensible Markup Language (often written as extensible Markup Language to justify the acronym). A markup language is a specification that adds new information to existing information while keeping two sets of information separate and XML is a set of rules for defining semantic tags that break a document into parts and identifies the different parts of the document. It is a meta-markup language that defines a syntax that is in turn used to define other domain-specific, semantic, structured markup languages. With XML we can store the information in a structured manner.

Comparison of HTML and XML

XML differs from HTML in many aspects. As, we already know, HTML is a markup language used for displaying information, while XML markup is used for describing data of virtually any type. In other words, we can say that HTML deals with how to present whereas XML deals with what to present. Actually, HTML is a markup language whereas XML is a markup language and a language for creating markup languages. HTML limits you to a fixed collection of tags and these tags are primarily used to describe how content will be displayed, such as, making text bold or italiced or headings etc., whereas with XML you can create new or any user defined tags. Hence, XML enables the creation of new markup languages to markup anything imaginable (such as Mathematical formulas, chemical formulas or reactions, music etc.)

Let us, understand the difference between HTML and XML with the help of an example. In HTML a song might be described using a definition title, definition data, an unordered list, and list items. But none of these elements actually have anything to do with music. The HTML might look something like this:

```
<HTML>
<body>
<dt>Indian Classical </dt>
<dd> by HariHaran , Ravi shankar  and Shubha Mudgal</dd>
<ul>
<li>Producer: Rajesh
<li>Publisher: T-Series Records
<li>Length: 6:20
<li>Written: 2002
<li>Artist: Village People
</ul>
<body>
</html>
```

In XML, the same data might be marked up like this:

```
<XML>
<SONG>
<TITLE>Indian Classical</TITLE>
<COMPOSER>Hariharan</COMPOSER>
<COMPOSER>Ravi Shankar</COMPOSER>
<COMPOSER>Shubha Mudgal</COMPOSER>
<PRODUCER>Rajesh</PRODUCER>
<PUBLISHER>T-Series</PUBLISHER>
```

```
<LENGTH>6:20</LENGTH>
<YEAR>2002</YEAR>
<ARTIST>Village People</ARTIST>
</SONG></XML>
```

Instead of generic tags like <dt> and <li>, this listing uses meaningful tags like
<SONG>, <TITLE>, <COMPOSER>, and <YEAR>. This has a number of
advantages, including the fact that its easier for a human to read the source code to
determine what the author intended.

4.3   AN OVERVIEW OF SGML

Now, we shall learn how SGML acts as a base for all the markup languages. SGML is
a system or meta-language for defining markup languages, organising and tagging
elements of a document. SGML was developed and standardised by the International
Organisation for Standards (ISO). SGML itself does not specify any particular
formatting; rather, it specifies the rules for tagging elements. These tags can then be
interpreted to format elements in different ways. HTML and XML are based on
SGML.

Authors mark up their documents by representing structural, presentational, and
semantic information alongside content. Each markup language defined in SGML is
known as SGML application. An SGML application is generally characterised by:

1)   An SGML declaration. The SGML declaration specifies which characters and
       delimiters may appear in the application. By means of an SGML declaration
       (XML also has one), the SGML application specifies which characters are to be
       interpreted as data and which characters are to be interpreted as markup. (They
       do not have to include the familiar < and > characters; in SGML they could just
       as easily be {and} instead).

2)   A document type definition (DTD). The DTD defines the syntax of markup
       constructs. It has rules for SGML/XML document just like there is grammar for
       English. The DTD may include additional definitions such as character entity
       references. We shall study later in this unit how we can create DTD for an XML
       document.

3)   A specification that describes the semantics to be ascribed to the markup. This
       specification also imposes syntax restrictions that cannot be expressed within
       the DTD.

4)   Document instances containing data (content) and markup. Each instance
       contains a reference to the DTD to be used to interpret it.
       Using the rules given in the SGML declaration and the results of the
       information analysis (which ultimately creates something that can easily be
       considered an information model), the SGML application developer identifies
       various types of documents—such as reports, brochures, technical manuals, and
       so on—and develops a DTD for each one. Using the chosen characters, the
       DTD identifies information objects (elements) and their properties (attributes).
       The DTD is the very core of an SGML application; how well it is made largely
       determines the success or failure of the whole activity. Using the information
       objects (elements), defined in the DTD, the actual information is then marked
       up using the tags identified for it in the application. If, the development of the
       DTD has been rushed, it might need continual improvement, modification, or
       correction. Each time the DTD is changed, the information that has been
       marked up with it might also need to be modified because it may be incorrect.
       Very quickly, the quantity of data that needs modification (now called legacy

data) can become a far more serious problem—one that is more costly and time-consuming than the problem that SGML was originally introduced to solve.

Why not SGML?

The SGML on the Web initiative existed a long time before XML was even considered. Somehow, though, it was never really successful. Basically, SGML is just too expensive and complicated for Web use on a large scale. It isn't that it can't be used its just that it won't be used. Using SGML requires too much of an investment in time, tools, and training.

Why do we need XML?

XML adds a list of features that make it far more suitable than either SGML or HTML for use on an increasingly complex and diverse Web:

- **Modularity:** Although HTML appears to have no DTD, there is an implied DTD hard-wired into Web browsers. SGML has a limitless number of DTDs, on the other hand, but there's only one for each type of document. XML enables us to leave out the DTD altogether or, using sophisticated resolution mechanisms, combine multiple fragments of either XML instances or separate DTDs into one compound instance.

- **Extensibility:** XML's powerful linking mechanisms allow you to link to the material without requiring the link target to be physically present in the object. This opens up exciting possibilities for linking together things like material to which you do not have write access, CD-ROMs, library catalogue, the results of database queries, or even non-document media such as sound fragments or parts of videos. Furthermore, it allows you to store the links separately from the objects they link. This makes long-term link maintenance a real possibility.

- **Distribution:** In addition to linking, XML introduces a far more sophisticated method of including link targets in the current instance. This opens the doors to a new world of composite documents—documents composed of fragments of other documents that are automatically (and transparently) assembled to form what is displayed at that particular moment. The content can be instantly tailored to the moment, to the media, and to the reader, and might have only a fleeting existence: a virtual information reality composed of virtual documents.

- **Internationality:** Both HTML and SGML rely heavily on ASCII, which makes using foreign characters very difficult. XML is based on Unicode and requires all XML software to support Unicode as well. Unicode enables XML to handle not just Western-accented characters, but also Asian languages.

- **Data orientation:** XML operates on data orientation rather than readability by humans. Although being humanly readable is one of XML's design goals, electronic commerce requires the data format to be readable by machines as well. XML makes this possible by defining a form of XML that can be more easily created by a machine, but it also adds tighter data control through the more recent XML schema.

4.4   DIFFERENCE BETWEEN SGML AND XML

Now, we shall study what are the basic differences between SGML and XML. SGML, the Standard Generalised Markup Language, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML.

SGML requires that structured documents reference a Document Type Definition (DTD) to be "valid", XML allows for "well-formed" data and can be delivered with and without a DTD. XML was designed so that SGML can be delivered, as XML, over the Web.

What does XML mean to SGML product vendors? On the technology front, SGML products should be able to read valid XML documents as they sit, as long as they are in 7-bit ASCII. To read internationalised XML documents, (for example in Japanese) SGML software will need modification to handle the ISO standard 10646 character set, and probably also a few industry-but-not-ISO standard encoding such as JIS and Big5. To write XML, SGML products will have to be modified to use the special XML syntax for empty elements.

On the business front, much depends on whether the Web browsers learn XML. If they do, SGML product vendors should brace for a sudden, dramatic demand for products and services from all the technology innovators who are, at the moment, striving to get their own extensions into HTML, and will (correctly) see XML as the way to make this happen. If the browsers remain tied to the fixed set of HTML tags, then XML will simply be an easy on-ramp to SGML, important probably more so because, the spec is short and simple than because of its technical characteristics. This will probably still generate an increase in market size, but not at the insane-seeming rate that would result from the browers' adoption of XML.

🗗 Check Your Progress 1

1) State True or False

   a) In HTML, we can define our own tags.   T ☐   F ☐

   b) XML deals with what to present on the web page.   T ☐   F ☐

   c) With XML one can create new markup languages like
      math markup language.   T ☐   F ☐

2) How does HTML differs from XML? Explain with the help of an example.

   …………………………………………………………………………...

   …………………………………………………………………………...

   …………………………………………………………………………...

3) What are the basic characterstics of SGML?

   …………………………………………………………………………...

   …………………………………………………………………………...

   …………………………………………………………………………...

4) What are the advantages of XML over HTML?

   …………………………………………………………………………...

   …………………………………………………………………………...

   …………………………………………………………………………...

5) Explain the basic differences between XML and SGML.

   …………………………………………………………………………...

   …………………………………………………………………………...

   …………………………………………………………………………...

4.5   XML DEVELOPMENT GOALS

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

The design and development goals for XML are:

- XML shall be straightforwardly usable over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents. In practice, this will only be possible when XML browsers are as robust and widely available as HTML browsers, but the principle remains intact.

- XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc.

- XML shall be compatible with SGML. Most of the people involved in the XML effort come from organisations that have a large, in some cases staggering, amount of material in SGML. XML was designed pragmatically, to be compatible with existing standards while solving the relatively new problem of sending richly structured documents over the web.

- It shall be easy to write programs, which process XML documents.

- The number of optional features in XML is to be kept to the absolute minimum, ideally zero. Optional features inevitably raise compatibility problems when users want to share documents and sometimes lead to confusion and frustration.

- XML documents should be human-legible and reasonably clear. If you don't have an XML browser and you've received a hunk of XML from somewhere, you ought to be able to look at it in your favourite text editor and actually figure out what the content means.

- The XML design should be prepared quickly. XML was needed immediately and was developed as quickly as possible. It must be possible to create XML documents in other ways: directly in a text editor, with simple shell and Perl scripts, etc

- The design of XML shall be formal and concise.

- XML documents shall be easy to create.

- Terseness in XML markup is of minimal importance. Several SGML language features were designed to minimise the amount of typing required to manually key in SGML documents. These features are not supported in XML. From an abstract point of view, these documents are indistinguishable from their more fully specified forms, but supporting these features adds a considerable burden to the SGML parser (or the person writing it, anyway). In addition, most modern editors offer better facilities to define shortcuts when entering text.

4.6   THE STRUCTURE OF THE XML DOCUMENT

In this section, we shall create a simple XML document and learn about the structure of the XML document. Example 4.1 is about the simplest XML document I can imagine, so start with it. This document can be typed in any convenient text editor, such as Notepad, vi, or emacs.

Example 4.1: Hello XML

```
<?xml version="1.0"?>
<FOO>
Hello XML!
</FOO>
```

Example-4.1 is not very complicated, but it is a good XML document. To be more precise, it is a well-formed XML document. ("Wellformed" is one of the terms, we shall study about it later)

Saving the XML file

After you have typed in *Example 4.1*, save it in a file called hello.xml or some other name. The three-letter extension .xml is fairly standard. However, do make sure that you save it in plain-text format, and not in the native format of a word processor such as WordPerfect or Microsoft Word.

Loading the XML File into a Web Browser

Now that you've created your first XML document, you're going to want to look at it. The file can be opened directly in a browser that supports XML such as Internet Explorer 5.0. *Figure 1*a shows the result.
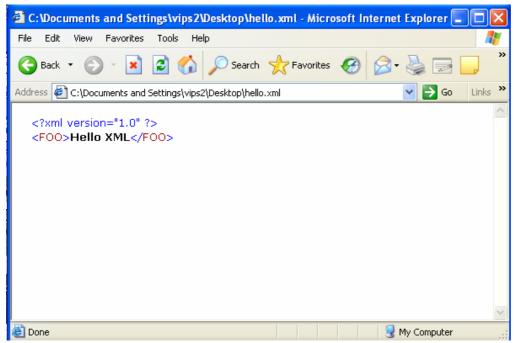

Figure 1: Output of XML Document

XML document may consists of the following parts:

a) The XML Prolog

XML file always starts with a Prolog as shown in the above example. An attribute is a name-value pair separated by an equals sign. Every XML document should begin with an XML declaration that specifies the version of XML in use. (Some XML documents omit this for reasons of backward compatibility, but you should include a version declaration unless you have a specific reason to leave it out). The minimal prolog contains a declaration that identifies the document as an XML document, like this:

<?xml version= "1.0"?>
The declaration may also contain additional information, like this:
<?xml version= "1.0" encoding= "ISO-8859-1" standalone= "yes"?>

The XML declaration may contain the following attributes:
Version Identifies the version of the XML markup language used in the data. This attribute is not optional.

Encoding

Identifies the character set used to encode the data. " ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.).

Standalone

This indicates whether or not this document references an external entity or an external data type specification (see below). If, there are no external references, then "yes" is appropriate.

The first line of the simple XML document in Listing 4.1 is the XML declaration:
<?xml version= "1.0"?>

<?xml version= "1.0" standalone= "no"?>

Identifying the version of XML ensures that future changes to the XML specification will not alter the semantics of this document. The standalone declaration simply makes explicit the fact that this document cannot "stand alone," and that it relies on an external DTD.

b) Elements and Attributes

Each Tag in a XML file can have Element and Attributes. Here's how a Typical Tag looks like,

<Email to= "admin@mydomain.com"
from= "user@mySite.com"
subject= "Introducing XML">
</Email>

In this Example, Email is known as an Element. This Element called E-mail has three attributes, to, from and subject.

The Following Rules need to be followed while declaring the XML Elements Names:

- Names can contain letters, numbers, and other characters
- Names must not start with a number or "_" (underscore)
- Names must not start with the letters xml (or XML or Xml).
- Names cannot contain spaces

Any name can be used, no words are reserved, but the idea is to make names descriptive. Names with an underscore separator are nice.
Examples: <author_name>, <published_date>.

Avoid "-" and "." in names. It could be a mess if your software tried to subtract name from first (author-name) or thought that "name" was a property of the object "author" (author.name).

Element names can be as long as you like, but don't exaggerate. Names should be short and simple, like this: <author_name> not like this: <name_of_the_author>.

XML documents often have a parallel database, where fieldnames are parallel to element names. A good rule is to use the naming rules of your databases.

The ":" should not be used in element names because it is reserved to be used for something called namespaces

In the above example-4.1, the next three lines after the prologue form a FOO element. Separately, <FOO> is a start tag; </FOO> is an end tag; and Hello XML! is the content of the FOO element. Divided another way, the start tag, end tag, and XML declaration are all markup. The text Hello XML! is character data.

c) Empty Tags:

In cases where you don't have to provide any sub tags, you can close the Tag, by providing a "/" to the Closing Tag. For example declaring
<Text></Text> is same a declaring   <Text />

d) Comments in XML File:

Comments in XML file are declared the same way as Comments in HTML File.

<Text>Welcome To XML Tutorial </Text>

<!-- This is a comment -->

<Subject />

e) Processing Instructions

An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

  <?target instructions?>

Where the target is the name of the application that is expected to do the processing, and instructions is a string of characters that embodies the information or commands for the application to process.

Well Formed Tags

One of the most important Features of a XML file is, it should be a Well Formed File. What it means that is all the Tags should have a closing tag. In a HTML file, for some tags like <br> we don't have to specify a closing tag called </br>. Whereas in a XML file, it is compulsory to have a closing tag. So we have to declare
</br>. These are what are known as Well Formed Tags.

We shall take another *Example 4.2* ignou.xml to understand all the components of the XML document.

Example 4.2

```
<?xml version= "1.0"?>
<?xml version= "1.0" encoding= "ISO-8859-1" standalone= "yes"?>
<!-- ignou.xml-- >
<!-- storing the student details in XML document-- >
<!DOCTYPE ignou SYSTEM ignou.dtd">
<student>
    < course type = "undergraduate">
    <name>
    <fname> Saurabh </fname>
    <lname> Shukla </lname>
    <city> New Delhi </city>
    <state> Delhi </state>
    <zip> 110035 </zip>
    <status id = "P">
</course>
< course type = "Post Graduate">
    <name>
        <fname> Sahil</fname>
```

```
        <lname> Shukla </lname>
    <city> Kanpur </city>
    <state> U.P.</state>
    <zip> 110034 </zip>
    <status id = "D">
</course>
<content> IGNOU is world class open university which is offering undergraduate
and postgraduate course.
</content>
</student>
```

In the above XML document, student is the root element and it contains all other elements (e.g. course, content). Lines preceeding the root element is the prolog which we have discussed above.

<!DOCTYPE ignou SYSTEM ignou.dtd">

The above given line specifies the document type definition for this xml document. Document files define the rules for the XML document. This tag contains three items: the name of the root element to which the DTD is applied, the SYSTEM flag (which denotes an external DTD), and the DTD's name and location. We shall study more about DTD in the next section.

4.7 USING DTD WITH XML DOCUMENT

DTD stands for **document type definition**. A DTD defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships to one another. In brief, we may say that DTD specifies, a set of rules for the structure of a document.

If present, the document type declaration must be the first, in the document after optional processing instructions and comments. The document type declaration identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all the contents of the document. Additional declarations may come from an external definition (a DTD) or be included directly in the document.

Creating DTD is just like creating a table in a database. In DTDs, you specify the structure of the data by declaring the element to denote the data. You can also specify whether providing a value for the element is mandatory or optional.

Declaring Elements in a DTD

After identifying the elements that can be used for storing structured data, they can be declared in a DTD. The XML document can be checked against the DTD. We can declare Element with the following syntax:

<!Element elementname(content-type or content-model)>

In the above syntax, elementname specifies the name of the element and content-type or content-model specifies whether the element contains textual data or other elements. For e.g. #PCDATA which specifies that element can store parsed character data(i.e. text). An element can be empty, unrestricted or container example. In the above mentioned example ignou.xml, the status is empty element whereas city is the unrestricted element.

Element Type	Description
Empty	Empty elements have no content and are marked up as <empty-elements>
Unrestricted	The opposite of an empty element is an unrestricted element, which can contain any element declared elsewhere in the DTD

In DTD various symbols are used to specify whether an element is mandatory or a optional or number of occurrences. The following is the list of symbols that can be used:

Symbol	Meaning	Example	Description
+	It indicates that there can be at least one or multiple occurrences of the element	Course+	There can multiple occurrences of course element.
*	It indicates that there can be either zero or any number of occurrences of the element	Content *	Any number of content element can be present.
?	It indicates that there can be either zero or exactly one occurrence.	Content ?	Content may not be present or present then only once
|	Or	City | state	City or state

We can declare attribute in DTD using the following syntax:

<!ATTLIST elementname att_name val_type [att_type] ["default"]>

att_name is the name of the attribute and val_type is the type of value which attribute can hold like CDATA defines that this attribute can contains a string.

We can also discuss whether attribute is mandatory or optional. We can do this with the help of att_type which can have the following values:

Attribute Type	Description
REQUIRED	If the attribute of an element is specified as # REQUIRED then, the value of that attribute must be specified if, it is not be specified then, the XML document will be invalid.
FIXED	If attribute of an element is specified as #FIXED then, the value of attribute cannot be changed in the XML document.
IMPLIED	If attribute of an element is specified as #IMPLIED then, attribute is optional i.e. this attribute need not be used every time the associated element is used.

Now, we are in the position to create the DTD named ignou.dtd, as shown in the *Example 4.2*

<?XML version= "1.0" rmd= "internal"?>

<!ELEMENT student(course +, content *)>

<!Element course (name, city, state, zip,status)>

<!ATTLIST course type CDATA #IMPLIED>

<!Element name (fname, lname)>

<!Element fname (#PCDATA)>

```
<!Element  lname (#PCDATA)>
<!Element  city (#PCDATA)>
<!Element  state (#PCDATA)>
<!Element  zip (#PCDATA)>
<!Element  content (#PCDATA)>
```

This example, references an external DTD, ignou.dtd, and includes element and attribute declarations for the course element. In this case, course is being given the semantics of a simple link from the XLL specification.

In order to determine if a document is valid, the XML processor must read the entire document type declaration (both internal and external). But for some applications, validity may not be required, and it may be sufficient for the processor to read only the internal declaration. In the example given above, if validity is unimportant and the only reason to read the doctype declaration is to identify the semantics of course, then, reading the external definition is not necessary.

4.8   XML PARSER

An XML parser (or XML processor) is the software that determines the content and structure of an XML document by combining XML document and DTD (if any present). *Figure 2* shows a simple relationship between XML documents, DTDs, parsers and applications. XML parser is the software that reads XML files and makes the information from those files available to applications and other programming languages. The XML parser is responsible for testing whether a document is well-formed and, if, given a DTD or XML schema, whether will also check for validity (i.e., it determines if the document follows the rules of the DTD or schema). Although, there are many XML parsers we shall discuss only Micorosoft's parser used by the Internet explorer and W3C's parser that AMAYA uses.
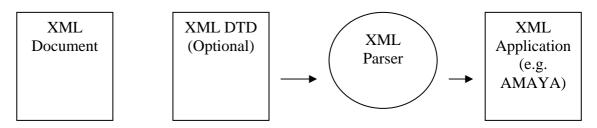
Figure 2: XML Documents and their Corresponding DTDs are Parsed and sent to Application.

XML Parser builds tree structures from XML documents. For example, an XML parser would build the tree structure shown in *Figure 2* for the previously mentioned example ignou.xml. If, this tree structure is created successfully without using a DTD, the XML document is considered **well-formed.** If, the tree stucture is created successfully and DTD is used, the XML document is considered valid. Hence, there can be two types of XML parsers : validating (i.e., enforces DTD rules) and non-validating (i.e., ignores DTD rules).

Figure 3: Tree Structure for ignou.xml

4.9   XML ENTITIES

Now, we shall learn about entity. Entities are normally external objects such as graphics files that are meant to be included in the document. Entity declarations [Section 4.3] allow you to associate a name with some other fragment of the document. That constr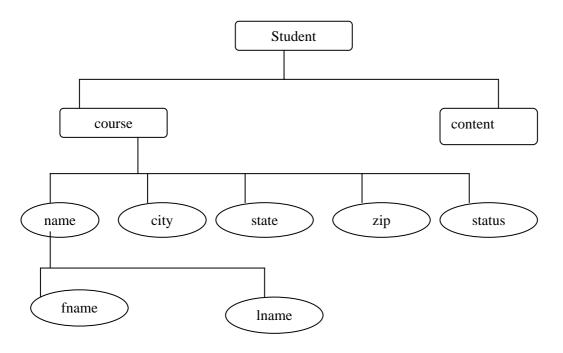uct can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data.There are three varities of entities in XML.

Here are a few typical entity declarations:

<!ENTITY BCA   "Bachelor of Computer Application.">

<!ENTITY course SYSTEM "/standard/course1.xml">

External Entity and Internal Entity : The first entity in the example given above is an internal entity because the replacement text is stored in the declaration. Using &BCA; anywhere in the document insert "Bachelor of computer Application." at that location. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document. You must declare an internal entity before you can use it. It can save you a lot of unnecessary typing.

The declaration of an internal entity has this form:

<!ENTITY name "replacement text">

Now, everytime the string &ampname; appears in your XML code, the XML processor will automatically replace it with the replacement text (which can be just as long as you like).

The XML specification predefines five internal entities:

* < produces the left angle bracket, <

* > produces the right angle bracket, >

* & produces the ampersand, &

- ' produces a single quote character (an apostrophe),
- " produces a double quote character,

External Entities: The second example id external entity. Using &course; will have – insert the contents of the file /standard/legalnotice.xml at that location in the document when it is processed. The XML processor will parse the content of that file as if its content were typed at the location of the entity reference. To reference external entities, you must have a DTD for your XML document.

External entities allow an XML document to refer to an external file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document.

Parameter Entities

Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration. The percent sign is also used in references to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded.

Check Your Progress 2

1) Explain the structure of an XML document.

   …………………………………………………………………………………...

   …………………………………………………………………………………..

   …………………………………………………………………………………..

2) Explain the different development goals of XML document.

   …………………………………………………………………………………...

   …………………………………………………………………………………..

   …………………………………………………………………………………...

3) What is DTD? Why do we use it? Explain the different components of DTD.

   …………………………………………………………………………………...

   …………………………………………………………………………………...

   …………………………………………………………………………………..

4) Differentiate between validating and non-validating parser.

   …………………………………………………………………………………...

   …………………………………………………………………………………...

   …………………..…………………………………………………………………

5) Explain the need/use of entities in XML document. Describe all three types of entities with the help of an example.

   …………………………………………………………………………………...

   …………………………………………………………………………………

   …………………………………………………………………………………...

6)   Where would you declare entities?

……………………………………………………………………...

…………………………………………………………………..

…………………………………………………………………..

7)   Why is an XML declaration needed?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

8)   Can entities in attribute values as well as in content be used?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

9)   Is the use of binary data permitted in a CDATA section?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

10)   How many elements should you have in a DTD?

…………………………………………………………………...……

…………………………………………………………….…………

…………………………………………………………………...11)   Is   it
necessary to validate XML documents?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

12)   How can we check whether the DTD is correct?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

13)   How can you put unparsed (non-XML) text in an external entity?

……………………………………………………………………...

……………………………………………………………………

……………………………………………………………………...

4.10 SUMMARY

XML, which stands for Extensible Markup Language, is defined as a structured
document containing structured information. Structured information contains both
content (words, pictures, etc.) and some indication of what role that content plays.
XML is a meta-language written in SGML that allows one to design a markup

language, used to allow for the easy interchange of documents on the World Wide Web. XML is a set of rules for defining semantic tags that break a document into parts and identifies the different parts of the document

XML differs from HTML in many aspects. As we know, HTML is a markup language and is used for displaying information, while XML markup is used for describing data of virtually any type. XML can be used to create other markup languages whereas HTML cannot be used for that purpose.

SGML acts as a base for all the markup languages. SGML is a system or meta-language for defining markup languages, organising and tagging elements of a document. SGML was developed and standardised by the International Organisation for Standards (ISO). SGML application is generally characterised by SGML declaration, Document Type Definition, specification and document instance.

XML is more suitable than SGML or HTML and is increasingly used for web application because of features like, modularity, extensibility distribution and data orientation. XML differs from SGML in some ways such as, SGML, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML.

XML was developed by an XML Working Group with few design and development goals. Some of them like XML shall be straightforwardly usable over the Internet, XML shall be compatible with SGML, the number of optional features in XML is to be kept to the absolute minimum, ideally zero. XML documents should be human-legible and reasonably clear etc. These two qualities are of utmost importance and have contributed to the success of XML.

XML document may consist of many parts like XML prolog, elements, attributes, empty tags, comments and processing instructions. XML document should begin with an XML declaration, which is known as an XML prolog that specifies the version of XML in use. There can be two types of elements in XML container elements and empty elements. An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data.

DTD stands for document type definition, it defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships with one another. The document type declaration identifies the root element of the document and may contain additional declarations. In DTDs, the structure of the data can be specified by declaring the element as denoting the data.

An XML parser (or XML processor) is the software that determines the content and structure of an XML document by combining XML document and DTD. The XML parser is responsible for testing whether a document is well-formed and, if given a DTD or XML schema, it also checks for the validity. XML Parser builds the tree structure from XML documents. If this tree structure is created successfully without using a DTD, the XML document is considered well-formed and if the tree structure is created successfully and DTD is used, the XML document is considered valid. Entities are normally external objects such as graphics files that are meant to be included in the document. Entity declarations allow you to associate a name with some other fragment of the document. There are three varities of entities in XML. Internal Entity, where the replacement text is stored in the declaration. Internal entities allow you to define shortcuts for frequently typed text. External entities allow

an XML document to refer to an external file. External entities contain either text or binary data. Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration

4.11 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) True/ False

   a) False

   b) True

   c) True

2) XML differs from HTML in many aspects. As we know, HTML is a markup language that is used for displaying information, while XML markup is used for describing data of virtually any type. HTML deals with How to present whereas XML deals with what to present. XML can be used for creating other markup languages whereas HTML cannot be used for creating other markup languages.

   In HTML a song might be described using a definition title, definition data, an unordered list, and list items. But none of these elements actually have anything to do with music. The HTML example:

```
<HTML>
<body>
<dt>Indian Classical </dt>
<dd> by HariHaran , Ravi shankar  and Shubha Mudgal</dd>
<ul>
<li>Producer: Rajesh
<li>Publisher: T-Series Records
<li>Length: 6:20
<li>Written: 2002
<li>Artist: Village People
</ul>
<body>
</html>
```

In XML the same data might be marked up like this:

```
<XML>
<SONG>
<TITLE>Indian Classical</TITLE>
<COMPOSER>Hariharan</COMPOSER>
<COMPOSER>Ravi Shankar</COMPOSER>
<COMPOSER>Shubha Mudgal</COMPOSER>
<PRODUCER>Rajesh</PRODUCER>
<PUBLISHER>T-Series</PUBLISHER>
<LENGTH>6:20</LENGTH>
<YEAR>2002</YEAR>
<ARTIST>Village People</ARTIST>
</SONG></XML>
```

Instead of generic tags like <dt> and <li>, this listing uses meaningful tags like <SONG>, <TITLE>, <COMPOSER>, and <YEAR>.

3) SGML stands for Standard Generalised Markup Language, SGML is a system or meta-language for defining markup languages, organising and tagging elements of a document. Each markup language defined in SGML is known an SGML application. An SGML application is generally characterised by the following:

   a) **An SGML Declaration:** The SGML declaration specifies which characters and delimiters may appear in the application. By means of an SGML declaration (XML also has one), the SGML application specifies which characters are to be interpreted as data and which characters are to be interpreted as markup.

   b) **A Document Type Definition (DTD):** The DTD defines the syntax of markup constructs. It has rules for SGML/XML document just as there are rules of grammar for English. The DTD may include additional definitions such as character entity references.

   c) **A Specification:** This describes the semantics to be ascribed to the markup. This specification also imposes syntax restrictions that cannot be expressed within the DTD.

   d) **Document Instances:** Contain data (content) and markup. Each instance contains a reference to the DTD to be used to interpret it.

4) The advantages of XML over HTML and SGML are the following:

   - *Modularity :* Although HTML appears to have no DTD, there is an implied DTD hard-wired into Web browsers. XML enables us to leave out the DTD altogether or, using sophisticated resolution mechanisms, combine multiple fragments of either XML instances or separate DTDs into one compound instance.

   - *Extensibility:* XML's powerful linking mechanisms allow you to link to material without requiring the link target to be physically present in the object. This helps in linking together things like material to which you do not have write access, CD-ROMs, library catalogue, the results of database queries etc.

   - *Distribution :*XML introduces a far more sophisticated method of including link targets in the current instance. This opens the doors to a new world of composite documents—documents composed of fragments of other documents that are automatically (and transparently) assembled to form what is displayed at that particular moment. The content can be instantly tailored to the moment, to the media, and to the reader.

   - *Internationality:* Both HTML and SGML rely heavily on ASCII, which makes using foreign characters very difficult. XML is based on Unicode and requires all XML software to support Unicode as well. Unicode enables XML to handle not just Western-accented characters, but also Asian languages.

   - *Data orientation:* XML operates on data orientation rather than readability by humans. Although being humanly readable is one of XML's design goals, electronic commerce requires the data format to be readable by machines as well.

5) XML differs from SGML in many respects. SGML, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML. SGML requires that structured documents reference a Document Type Definition (DTD) to be "valid", XML allows for "well-formed" data and can be delivered with and without a DTD.

Check Your Progress 2

1) XML document may consists of the following parts:

a) The XML Prolog

XML file always starts with a Prolog. Every XML document should begin with an XML declaration that specifies the version of XML in use. The minimal prolog contains a declaration that identifies the document as an XML document, like this:

<?xml version="1.0"?>
The declaration may also contain additional information, like this:
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

b) Elements and Attributes:

Each Tag in a XML file can have Element and Attributes. Here's what a Typical Tag looks like,

<Email to="admin@mydomain.com"
from="user@mySite.com"
subject="Introducing XML">
</Email>

In this Example, Email is called an Element. This Element called E-mail has three attributes: to, from and subject.

c) Empty Tags:

In cases where you don't have to provide any sub tags, you can close the Tag, by providing a "/" to the Closing Tag. For example declaring

<Text></Text> is same a declaring   <Text />

d) Comments in XML File:

Comments in XML file are declared the same way as Comments in HTML File.
<Text>Welcome To XML Tutorial </Text>
<!-- This is a comment -->
<Subject />

e) Processing Instructions

An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

  <?target instructions?>

Where the target is the name of the application that is expected to do the processing, and instructions is a string of characters that embodies the information or commands for the application to process.

2) XML was developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

    The design and development goals of XML are:

    - XML shall be straightforwardly usable over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents.

    - XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc.

    - XML shall be compatible with SGML. Most of the people involved in the XML effort come from organisations that have a large, in some cases staggering, amount of material in SGML.

    - It shall be easy to write programs, which process XML documents.

    - The number of optional features in XML is to be kept to the absolute minimum, ideally zero.

    - XML documents should be human-legible and reasonably clear.

    - The XML design should be prepared quickly. XML was needed immediately and was developed as quickly as possible.

    - The design of XML shall be formal and concise.

    - XML documents shall be easy to create.

    - Terseness in XML markup is of minimal importance.

3) DTD stands for document type definition. A DTD defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships with one another. The document type declaration identifies the root element of the document and may contain additional declarations. To determine whether document is valid or not, the XML processor must read the entire document type declaration.

    In DTDs, the structure of the data can be specified by declaring element to denote the data. DTD can be declared with the following syntax:

    <!Element elementname(content-type or content-model)>

    In the above syntax, elementname specifies the name of the element & content-type or content-model specifies whether the element contains textual data or other elements. . For e.g. #PCDATA which specifies that element can store parsed character data (i.e. text). DTD can describe two types of elements

    Empty : Empty elements have no content and are marked up as <empty-elements>

Unrestricted: The opposite of an empty element is an unrestricted element, which can contain any element declared elsewhere in the DTD.
DTD may contain various symbols. These varied symbols are used to specify an element and may be mandatory, or optional, or number. For example '+' is used indicate that there can be at least one or multiple occurrences of the element, '*' indicates that there can be either zero or any number of occurrences of the element, '?' indicates that there can be either zero or exactly one occurrence etc.
Attribute can be declared in DTD by the following syntax:
<!ATTLIST elementname att_name val_type [att_type] ["default"]>
att_name is the name of the attribute and val_type is the type of value which an attribute can hold such as CDATA. CDATA defines that this attribute can contains a string.
An Attribute type may be defined as REQUIRED, FIXED or IMPLIED.

4) XML parser, which enforces the DTD rules on the XML document, is known as the validating parser, whereas the XML document which ignores DTD rules, is known as the non-validating parser.

5) Entity declarations are used in XML document to associate a name with some other fragment of the document. It is also used to create shortcuts for the frequently typed text. That construct/text can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data. There are three varieties of entities in XML.

Internal Entity: In internal entity the replacement text is stored in the declaration. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document. The declaration of an internal entity has this form:

<!ENTITY name "replacement text">
for e.g,
<!ENTITY BCA   "Bachelor of Computer Application.">
Using &BCA; anywhere in the document insert "Bachelor of computer Application." at that location.

External Entities: External entities allow an XML document to refer to an external file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document. For e.g.,

<!ENTITY course SYSTEM "/standard/course1.xml">
Using &course; will have insert the contents of the file /standard/legalnotice.xml at that location in the document when it is processed.

Parameter Entities :
Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration. The percent sign is also used with reference to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded.

6)   You can declare entities inside either the internal subset or the external subset of the DTD. If , you have an external DTD, you will have to create a complete DTD. If, you need only the entities then, you can get away with an internal DTD subset.

Entity references in XML documents that have external DTD subsets are only replaced when the document is validated.

7) As such we do not need an XML declaration. XML has also been approved as a MIME type, which means that if you add the correct MIME header (xml/text or xml/application), a Web server can explicitly identify the data that follows as being an XML document, regardless of what the document itself says. MIME, (Multipurpose Internet Mail Extensions), is an Internet standard for the transmission of data of any type via electronic mail. It defines the way messages are formatted and constructed, can indicate the type and nature of the contents of a message, and preserve international character set information. MIME types are used by Web servers to identify the data contained in a response to a retrieval request.

The XML declaration is not mandatory for some practical reasons; SGML and HTML code can often be converted easily into perfect XML code (if it isn't already). If the XML declaration was compulsory, this wouldn't be possible.

8) You can use entity references in attribute values, but an entity cannot be the attribute value. There are strict rules on where entities can be used and when they are recognised. Sometimes they are only recognised when the XML document is validated.

9) Technically, there's nothing which stops you from using binary data in a CDATA section, even though it's really a character data section, particularly as, the XML processor doesn't consider the contents of a CDATA section to be part of the document's character data. However, this may cause increase in file size and all the transportation problems that may be implied. Ultimately, it would could create problems for the portability of the XML documents when there is a far more suitable feature of XML you can use for this purpose. Entities, allow you to declare a format and a helper application for processing a binary file (possibly displaying it) and associating it with an XML document by reference.

10) It all depends on your application. HTML has about 77 elements, and some of the industrial DTDs have several hundred. It isn't always the number of elements that determine the complexity of the DTD. By using container elements in the DTD (which add to the element count), authoring software is able to limit possible choices and automatically enter required elements. Working with one of the major industrial DTDs can actually be far easier than creating HTML. Because HTML offers you more free choice, you have to have a much better idea of the purpose of all the elements rather than just the ones you need.

11) No, but unless you are certain that your documents are valid you cannot predict what will happen at the receiving end. Although the XML specification lays down rules of conduct for XML processors and specifies what they must do when certain invalid content is parsed, the requirements are often quite loose. Validation certainly won't do any harm—though it might create some extra work.

12) Simply validate an XML document that uses the DTD. There aren't as many tools specifically intended for checking DTDs as there are for validating documents. However, when an XML document is validated against the DTD, the DTD is checked and errors in the DTD are reported.

13)   There are several ways. You could declare a TEXT notation, but this would not allow you to physically include the text in the XML document. (It would go to the helper application you designated in the notation declaration.) The best way would probably be to declare the file containing the text as an external text entity and put the text in that file in a CDATA section.

4.12 FURTHER READINGS/REFERENCES

- Brett McLaughlin & Justin Edelson, *Java and XML,* Third Edition, Oreilly

- Simon North, *Sams Teach yourself XML in 21 days,* Macmillan Computer Publishing

- Eric Westermann, *Learn XML in a weekend XML Bible,* Premier press

- Natanya Pitts-Moultis and Cheryl Kirk, *XML Black Book,* The Coriolis Group

References websites:

- www.w3schools.com
- www.xml.com
- www.java.sun.com/xml/tutorial_intro.html
- www.zvon.org
- www.javacommerce.com

UNIT 1 WEB SECURITY CONCEPTS

1.0 INTRODUCTION

Web Security may be defined as technological and managerial procedures applied to computer systems to ensure the availability, integrity, and confidentiality of information. It means that protection of integrity, availability and confidentiality of computer assets and services from associated threats and vulnerabilities.

The security of the web is divided into two categories (a) computer security, and (b) network security. In generic terms, computer security is the process of securing a single, standalone computer; while network security is the process of securing an entire network of computers.

(a)   **Computer Security:** Technology and managerial procedures applied to computer systems to ensure the availability, integrity, and confidentiality of the data managed by the computer.

(b)   **Network Security:** Protection of networks and their services from unauthorised modification destruction, or disclosure and provision of assurance that the network performs its critical functions correctly and that are nor harmful side effects.

The major points of weakness in a computer system are hardware, software, and data. However, other components of the computer system may be targeted. In this unit, we will focus on web security related topics.

1.1 OBJECTIVES

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

• achieve integrity, confidentiality and availability of information on the internet integrity and confidentiality can also be enforced on web services through the use of SSL(Secure Socket Layer); and

- ensure secure communication on the Internet for as e-mail,
  Internet faxing, and other data transfers.

1.2 WEB SERVICES AND ITS ADVANTAGES

According to the W3C a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface that is described in a machine-processable format such as WSDL. Other systems interact with the Web service in a manner prescribed by its interface using messages, which may be enclosed in a SOAP envelope. These messages are typically conveyed using HTTP, and normally comprise XML in conjunction with other Web-related standards. Software applications written in various programming languages and running on various platforms, can use web services to exchange data over computer networks like, the Internet, in a manner similar to inter-process communication on a single computer. This interoperability (i.e. between Java and Python, or Microsoft Windows and Linux applications) is due to the use of open standards.

Web services and its advantages

- Web services provide interoperability between various software applications running on disparate platforms/operating systems.

- Web services use open standards and protocols. Protocols and data formats are text-based where possible, which makes it easy for developers to undestand.

- By utilising HTTP, web services can work through many common firewall security measures without having to make changes to the firewall filtering rules.

- Web services allow software and services from different companies and locations to be combined easily to provide an integrated service.

- Web services allow the reuse of services and components within an infrastructure.

- Web services are loosely coupled thereby, facilitating a distributed approach to application integration.

So it is necessary to provide secure communication in web communication.

WS-Security

WS-Security (Web Services Security) is a communications protocol providing a means for applying security to Web Services Originally developed by IBM, Microsoft, and VeriSign, the protocol is now officially called WSS and developed via a committee in Oasis-Open.

The protocol contains specifications on how integrity and confidentiality can be enforced on Web Services messaging.

Three basic security concepts important to information on the Internet are confidentiality, integrity, and availability.

A Common Windows Security Problem

Unfortunately, many Microsoft Windows users are unaware of a common security leak in their network settings.

This is a common setup for network computers in Microsoft Windows:

- Client for Microsoft Networks
- File and Printer Sharing for Microsoft Networks
- NetBEUI Protocol
- Internet Protocol TCP/IP

If your setup allows NetBIOS over TCP/IP, you have a security problem:

- Your files can be shared all over the Internet
- Your logon-name, computer-name, and workgroup-name are visible to others.

If your setup allows File and Printer Sharing over TCP/IP, you also have a problem:

- Your files can be shared all over the Internet

Solving the Problem

For Windows 95/98/2000 users:

You can solve your security problem by disabling NetBIOS over TCP/IP:

You must also disable the TCP/IP Bindings to Client for Microsoft Networks and File and Printer Sharing.

If, you still want to share your Files and Printer over the network, you must use the NetBEUI protocol instead of the TCP/IP protocol. Make sure you have enabled it for your local network.

🗗 **Check Your Progress 1**

1) Compare and Contrast Computer and Network Security.

    ……………………………………………………………………………………
    ……………………………………………………………………………………
    ……………………………………………………………………………………

2) Explain IP protocol Suit.

    ……………………………………………………………………………………
    ……………………………………………………………………………………
    ……………………………………………………………………………………

3) What is web security?Explain with suitable examples.

    ……………………………………………………………………………………...
    ……………………………………………………………………………………
    ……………………………………………………………………………………

1.3   WEB SECURITY CONCEPTS

In this section, we will describe briefly four concepts related to web security.

Three basic security concepts important to information on the Internet are confidentiality, integrity, and availability. Concepts relating to the people who use that information are authentication, authorisation, and nonrepudiation. Integrity and confidentiality can also be enforced on Web Services through the use of Transport Layer Security (TLS). Both SSL and TSL (Transport Layer Security) are the same. The dependencies among these concepts (also called objects) is shown in *Figure 1*.

1.3.1   Integrity

Integrity has two facets:

Data Integrity: This property, that data has not been altered in an unauthorised manner while in storage, during processing or while in transit. Another aspect of data integrity is the assurance that data can only be accessed and altered by those authorised to do so. Often such integrity is ensured by use of a number referred to as a Message Integrity Code or Message Authentication Code. These are abbreviated as MIC and MAC respectively.

System Integrity: This quality that a system has when performing the intended function in an unimpaired manner, free from unauthorised manipulation.
Integrity is commonly an organisations most important security objective, after availability. Integrity is particularly important for critical safety and financial data used for activities such as electronic funds transfers, air traffic control, and financial accounting.

1.3.2   Confidentiality

Confidentiality is the requirement that private or confidential information should not to be disclosed to unauthorised individuals. Confidentiality protection applies to data in storage, during processing, and while in transit.

For many organisations, confidentiality is frequently behind availability and integrity in terms of importance. For some types of information, confidentiality is a very important attribute. Examples include research data, medical and insurance records, new product specifications, and corporate investment strategies. In some locations, there may be a legal obligation to protect the privacy of individuals.

This is particularly true for banks and loan companies; debt collectors; businesses that extend credit to their customers or issue credit cards; hospitals, doctors' offices, and medical testing laboratories; individuals or agencies that offer services such as psychological counseling or drug treatment; and agencies that collect taxes.

1.3.3   Availability

Availability is a requirement intended to assure that systems work promptly and service is not denied to authorised users. This objective protects against:

- Intentional or accidental attempts to either:
  - perform unauthorised deletion of data or
  - otherwise cause a denial of service or data.

- Attempts to use system or data for unauthorised purposes.
  Availability is frequently an organisations foremost security objective. To make information available to those who need it and who can be trusted with it, organisations use authentication and authorisation.

Authentication

Authentication is proving that a user is whom s/he claims to be. That proof may involve something the user knows (such as a password), something the user has (such as a "smartcard"), or something about the user that proves the person's identity (such as a fingerprint).

Authorisation

Authorisation is the act of determining whether a particular user (or computer system) has the right to carry out a certain activity, such as reading a file or running a program. Authentication and authorisation go hand in hand. Users must be authenticated before carrying out the activity they are authorised to perform.

Accountability (to be individual level)

Accountability is the requirement that actions of an entity may be traced uniquely to that entity. Accountability is often an organisational policy requirement and directly supports repudiation, deterrence, fault isolation, intrusion detection and prevention, and after action recovery and legal action.

Assurance (that the other four objectives have been adequately met)

Assurance is the basis for confidence that the security measures, both technical and operational, work as intended to protect the system and the information it processes. Assurance is essential, without it other objectives cannot be met.

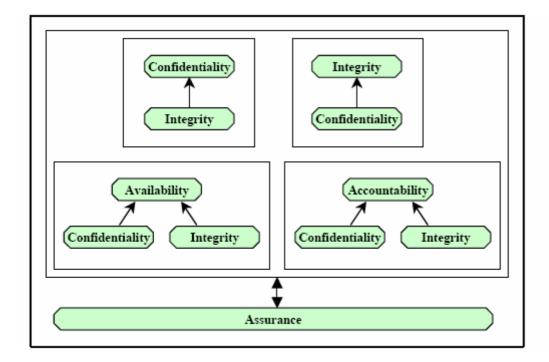The above mentioned security objects, dependencies are shown in *Figure 1*.

Figure 1: Security Objects Dependencies

⌕ **Check Your Progress 2**

1)   List the basic security concepts.

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

2)   What do you understand by information assurance?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

3)   Compare and contrast data integrity and system integrity.

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

1.3.4   Secure Socket Layer (SSL)/Transport Layer Security(TLS)

Secure Socket Layer (SSL) and Transport Layer Security (TLS), its successor, are cryptographic protocols which provide secure communication on the Internet for as e-mail, internet faxing, and other data transfers.

Description

SSL provides endpoint authentication and communication privacy over the Internet using cryptography. In typical use, only the server is authenticated (i.e. its identity is ensured) while the client remains unauthenticated; mutual authentication requires public key infrastructure (PKI) deployment to clients. The protocols allow client/server applications to communicate in a way designed to prevent eavesdropping, tampering, and message forgery.

Tampering may relate to:

- **Tampering (Sports):** The practice, often illegal, of professional sports teams negotiating with athletes of other teams.
- **Tamper-evident:** A device or process that makes unauthorised access to a protected object easily detected.
- **Tamper proofing:** A methodology used to hinder, deter or detect unauthorized access to a device or circumvention of a security system.

Message Forgery

In cryptography, message forgery is the sending of a message to deceive the recipient of whom the real sender is. A common example is sending a spam e-mail from an address belonging to someone else

SSL involves three basic phases:

1) Peer negotiation for algorithm support,

2) Public key encryption-based key exchange and certificate-based authentication, and

3) Symmetric cipher-based traffic encryption.

During the first phase, the client and server negotiation uses cryptographic algorithms. Current implementations support the following choices:

- For public-key cryptography: RSA, Diffie-Hellman, DSA or Fortezza;

- For symmetric ciphers: RC2, RC4, IDEA, DES, Triple DES or AES;

- For one-way hash functions: MD5 or SHA.

SSL working

The SSL protocol exchanges records. Each record can be optionally compressed, encrypted and packed with a Message Authentication Code (MAC). Each record has a "content type" field that specifies which upper level protocol is being used.
When the connection begins, the record level encapsulates another protocol, the handshake protocol. The client then sends and receives several handshake structures:

- It sends a *ClientHello* message specifying the list of cipher suites, compression methods and the highest protocol version it supports. It also sends random bytes which will be used later.

- Then it receives a *ServerHello*, in which the server chooses the connection parameters from the choices offered by the client earlier.

- When the connection parameters are known, client and server exchange certificates (depending on the selected public key cipher). These certificates are currently X.509, but there is also a draft specifying the use of OpenPGP based certificates.

- The server can request a certificate from the client, so that the connection can be mutually authenticated.

- Client and server negotiate a common secret called "aster secret", possibly using the result of a Diffie-Hellman exchange, or simply encrypting a secret with a public key that is decrypted with the peer's private key. All other key data is derived from this "master secret" (and the client- and server-generated random values), which is passed through a carefully designed "Pseudo Random Function".

TLS/SSL have a variety of security measures:

- Numbering all the records and using the sequence number in the MACs.

- Using a message digest enhanced with a key.

- Protection against several known attacks (including man in the middle attacks), like those involving a downgrade of the protocol to previous (less secure) versions, or weaker cipher suites.

- The message that ends the handshake ("Finished") sends a hash of all the exchanged data seen by both parties.

- The pseudo random function splits the input data in 2 halves and processes them with different hashing algorithms (MD5 and SHA), then XORs them together. This way it protects itself in the event that one of these algorithms is found to be vulnerable.

Public key cryptography is a form of cryptography which generally allows users to communicate securely without having prior access to a shared secret key. This is done by using a pair of cryptographic keys, designated as public key and private key, which are related mathematically.

The term asymmetric key cryptography is a synonym for public key cryptography though a somewhat misleading one. There are asymmetric key encryption algorithms that do not have the public key-private key property noted above. For these algorithms, both keys must be kept secret, that is both are private keys.

In public key cryptography, the private key is kept secret, while the public key may be widely distributed. In a sense, one key "locks" a lock; while the other is required to unlock it. It should not be possible to deduce the private key of a pair, given the public key, and in high quality algorithms no such technique is known.

There are many forms of public key cryptography, including:

- *Public key encryption* keeping a message secret from anyone that does not possess a specific private key.

- *Public key digital signature* allowing anyone to verify that a message was created with a specific private key.

- *Key agreement* generally, allowing two parties that may not initially share a secret key to agree on one.

1.4 HTTP AUTHENTICATION

A web client can authenticate a user to a web server using one of the following mechanisms:

- HTTP Basic Authentication

- HTTP Digest Authentication

- Form Based Authentication

- HTTPS Client Authentication

1.4.1 HTTP Basic Authentication

HTTP Basic Authentication, which is based on a username and password, is the authentication mechanism defined in the HTTP/1.0 specification. A web server requests a web client to authenticate the user. As a part of the request, the web server passes the realm (a string) in which the user is to be authenticated. The realm string of Basic

Authentication does not have to reflect any particular security policy domain (confusingly also referred to as a realm). The web client obtains the username and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm.

Basic Authentication is not a secure authentication as user passwords are sent in simple base64 ENCODING (not ENCRYPTED !), and there is no provision for target server authentication. Additional protection mechanism can be applied to mitigate

these concerns: a secure transport mechanism (HTTPS), or security at the network level (such as the IPSEC protocol or VPN strategies) can be deployed. This is shown in the following role-based authentication.

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>User Auth</web-resource-name>
                        <url-pattern>/auth/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>manager</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                <auth-method>BASIC</auth-method>
                <realm-name>User Auth</realm-name>
        </login-config>

        <security-role>
                <role-name>admin</role-name>
        </security-role>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
</web-app>
```

1.4.2   HTTP Digest Authentication

Similar to HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However, the authentication is performed by transmitting the password in an ENCRYPTED form, which is much MORE SECURE than the simple base64 encoding used by Basic Authentication, e.g., HTTPS Client Authentication. As Digest Authentication is not currently in widespread use, servlet containers are encouraged but NOT REQUIRED to support it.

The advantage of this method is that the cleartext password is protected in transmission, it cannot be determined from the digest that is submitted by the client to the server. Digested password authentication supports the concept of digesting user passwords. This causes the stored version of the passwords to be encoded in a form that is not easily reversible, but that the web server can still utilise for authentication.

From a user perspective, digest authentication acts almost identically to basic authentication in that it triggers a login dialogue.

The difference between basic and digest authentication is that on the network connection between the browser and the server, the password are encrypted, even on a non-SSL connection. In the server, the password can be stored in clear text or encrypted text, which is true for all login methods, and is independent of the choice that the application deployer makes.

Digested password is authentication based on the concept of hash or digest. In this stored version, the passwords is encoded in a form that is not easily reversible and this is used for authentication. Digest authentication acts almost identically to basic authentication in that it triggers a login dialogue. The difference between basic and digest authentication is that on the network connection between the browser and the server, the password is encrypted, even on a non-SSL connection. In the server, the password can be stored in clear text or encrypted text, which is true for all login methods and is independent of the application deployment.

1.4.3 Form Based Authentication

The look and feel of the 'login screen' cannot be varied using the web browser's built-in authentication mechanisms. This form based authentication mechanism allows a developer to CONTROL the look and feel of the login screens.

The web application deployment descriptor, contains entries for a login form and error page. The login form must contain fields for entering a username and a password. These fields must be named j_username and j_password, respectively.

When a user attempts to access a protected web resource, the container checks the user's authentication. If the user is authenticated and possesses authority to access the resource, the requested web resource is activated and a reference to it is returned. If the user is not authenticated, all of the following steps occur:

1) The login form associated with the security constraint is sent to the client and the URL path triggering the authentication stored by the container.

2) The user is asked to fill out the form, including the username and password fields.

3) The client posts the form back to the server.

4) The container attempts to authenticate the user using the information from the form.

5) If authentication fails, the error page is returned using either a forward or a redirect, and the status code of the response is set to 200.

6) If authentication succeeds, the authenticated user's principal is checked to see if it is in an authorised role for accessing the resource.

7) If the user is authorised, the client is redirected to the resource using the stored URL path.

The error page sent to a user that is not authenticated contains information about the failure.

Form Based Authentication has the same lack of security as Basic Authentication since the user password is transmitted as a plain text and the target server is not authenticated. Again additional protection can alleviate some of these concerns: a secure transport mechanism (HTTPS), or security at the network level (such as the IPSEC protocol or VPN strategies) are applied in some deployment scenarios.

Form based login and URL based session tracking can be problematic to implement. Form based login should be used only when, sessions are being maintained by cookies or by SSL session information.

1.4.4   HTTPS Client Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a Public Key Certificate (PKC). Currently, PKCs are useful in e-commerce applications and also for a single-sign-on from within the browser. Servlet containers that are not J2EE technology compliant are not required to support the HTTPS protocol.

Client-certificate authentication is a more secure method of authentication than either BASIC or FORM authentication. It uses HTTP over SSL, in which the server and, optionally, the client authenticate one another with Public Key Certificates. Secure Sockets Layer (SSL) provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. It is issued by a trusted organisation, which is known as a certificate authority (CA), and provides identification for the bearer. If, you specify client-certificate authentication, the Web server will authenticate the client using the client's X.509 certificate, a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure (PKI). Prior to running an application that uses SSL, you must configure SSL support on the server and set up the public key certificate.

⏚ Check Your Progress 3

1)   Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and given a scenario, select an appropriate type.
…………..………………………………………………………………..
………………..………………………………………………………..
…………………………..……………………………………………..

1.5   SUMMARY

To Achieve integrity, confidentiality and availability of Information on the internet is the goal of web security integrity. Confidentiality can also be enforced on web services through the use of SSL. Integrity is The property that data has not been altered in an unauthorised manner while in storage, during processing or while in transit. Confidentiality is the requirement that private or confidential information is not to be disclosed to unauthorised individuals. Confidentiality protection applies to data in storage, during processing, and while in transit. Availability is a requirement intended to assure that systems work promptly and that service is not denied to authorised users.

1.6   SOLUTIONS/ANSWERS

Check Your Progress 1

1)   Computer Security: Technology and managerial procedures applied to computer systems to ensure the availability, integrity, and confidentiality of the data managed by the computer. Whereas, Network Security is protection of networks and their services from unauthorized modification destruction, or disclosure and provision of assurance that the network performs its critical functions correctly and there are not harmful side effects.

2)   IP protocol suit: The different types of protocols used in different layers are Physical, data link, network, transport, session, presentation and applications layer.

3)   Web Security can be defined as technological and managerial procedures applied to computer systems to ensure the availability, integrity, and confidentiality of the information. It means that protection of Integrity, Availability & confidentiality of computer assets and services from associated threats and vulnerabilities. Explain with suitable example HTTPS, SSL, IPSec etc.

Check Your Progress 2

1)   The basic security concepts are:
     Integrity, authenticity, confidentiality, authorisation, availability, and assurance.

2)   Information assurance is the basis for confidence that the security measures, both technical and operational, work as intended to protect the system and the information it processes.

3)   **Data Integrity:** The property that data has not been altered in an unauthorised manner while in storage, during processing or while in transit.

     System Integrity: The quality that a system has, when, performing the intended function in an unimpaired manner, free from unauthorised manipulation.

Check Your Progress 3

1)   Hint: Compare and Authentication type with suitable example in different scenario depending upon the application type finance, stock exchange, simple message exchange between two persons, remote logging, client server authentication etc.

1.7   FURTHER READINGS/REFERENCES

*   Stalling William, *Cryptography and Network Security, Principles and Practice*, 2000, SE, PE.

*   Daview D. and Price W., *Security for Computer Networks,* New York:Wiley, 1989.

*   Chalie Kaufman, Radia Perlman, Mike Speciner, *Network Security,* Pearson Education.

*   B. Schnier, *Applied Cryptography*, John Wiley and Sons

- Steve Burnett & Stephen Paine, *RSA Security's Official Guide to Practice*, SE, PE.

- Dieter Gollmann, *Computer Security*, John Wiley & Sons.

Reference websites:

- *World Wide Web Security FAQ:*
  http://www.w3.org/Security/Faq/www-security-faq.html

- *Web Security*: http://www.w3schools.com/site/site_security.asp

- *Authentication Authorisation and Access Control:*
  http://httpd.apache.org/docs/1.3/howto/auth.html

- *Basic Authentication Scheme:*
  http://en.wikipedia.org/wiki/Basic_authentication_scheme

- *OpenSSL Project:* http:/www.openssl.org

- *Request for Comments 2617 :* http://www.ietf.org/rfc/rfc2617.txt

- *Sun Microsystems Enterprise JavaBeans Specification:*
  http://java.sun.com/products/ejb/docs.html.

- *Javabeans Program Listings*: http:/e-docs.bea.com

UNIT 2 SECURITY IMPLEMENTATION

1.0 INTRODUCTION

In unit 1 we have discussed web services and its advantages, web security concepts, http authentication etc. This unit the following paragraphs and sections describes security implementation, security&servlet, form based custom authentication, and SSL authentication.

Intranet users are commonly required to use, a separate password to authenticate themselves to each and every server they need to access in the course of their work. Multiple passwords are an ongoing headache for both users and system administrators. Users have difficulty keeping track of different passwords, tend to choose poor ones, and then write them down in obvious places. Administrators must keep track of a separate password database on each server and deal with potential security problems related to the fact that passwords are sent over the network routinely and frequently.

2.1 OBJECTIVES

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

- the threats to computer security;

- what causes these threats;

- various security techniques;

- implementing Security Using Servlets, and

- implementing Security Using EJB's.

2.2   SECURITY IMPLEMENTATION

In this section, we will look at security implementation issues.

2.2.1   Security Considerations

System architecture vulnerabilities can result in the violation of the system's security policy. Some of these are described below:

Covert Channel: It is a way for an entity to receive information in an unauthorised manner. It is an information flow that is not controlled by a security mechanism. It is an unintended communication, violating the system's security policy, between two of more users/subjects sharing a common resource.

This transfer can be of two types:

(1) Covert Storage Channel: When a process writes data to a storage location and another process directly or indirectly reads it. This situation occurs when the processes are at different security levels, and therefore are not supposed to be sharing sensitive data.

(2) Covert Timing Channel: One process relays information to another by modulating its use of system resources. There is not much a user can do to countermeasure these channels. But for Trojan that uses HTTP protocol, intrusion detection and auditing may detect a covert channel. Buffer overflow or Parameter checking or **"smashing the stack":** Failure to check the size of input streams specified by the parameters. For example, buffer overflow attack exploits this vulnerability in some operating systems and programs. This happens when programs do not check the length of data that is inputted into a program and then processed by the CPU. The various countermeasures are: (a) proper programming and good coding practices, (b) Host IDS, (c) File system permission and encryption, (d) Strict access control, and (e) Auditing.

Maintenance Hook or Trap Door or Back Doors: These are instructions within the software that only the developers know about and can invoke. This is a mechanism designed to bypass the system's security protections. The various countermeasures are:

(a) Code reviews and unit integration testing.

(b) Host intrusion detection system.

(c) Using file permissions to protect configuration files and sensitive information from being modified.

(d) Strict access control.

(e) File system encryption, and

(f) Auditing.

Timing Issues or Asynchronous attack or Time of Check to Time of Use attack:

This deals with the timing difference of the sequences of steps a system takes to complete a task. This attack exploits the difference in the time that security controls were applied and the time the authorised service was used. A time-of-check versus time-of-use attack, also called race conditions, could replace autoexec.bat.

The various countermeasures for such type of attacks are:

(a) Host intrusion detection system.

(b) File system permissions and encryption.

(c) Strict access control mechanism, and

(e) Auditing.

2.2.2 Recovery Procedures

When a trusted system fails, it is very important that the failure does not compromise the security policy requirements. The recovery procedures also should not give any opportunity for violation of the system's security policy. The system restart must be in a secure mode. Startup should be in the maintenance mode that permits access the only privileged users from privileged terminals.

Fault-tolerant System: In this system, the computer or network continues to function even when a component fails. In this the system has the capability of detecting the fault and correcting the fault as well.

Failsafe System: In this system, the program execution is terminated and the system is protected from being compromised when a system (hardware or software) failure occurs is detected.

Failsoft or resilient: When a system failure occurs and is detected, selected non-critical processing is terminated. The system continues to function in a degraded mode.

Failover : This refers to switching to a duplicate "hot" backup component in real time when a hardware or software failure occurs.

Cold Start: This is required when a system failure occurs and the recovery procedures cannot return the system to a known, reliable, secure state. The maintenance mode of the system is usually employed to bring data consistency through external intervention.

🗐 Check Your Progress 1

1)   What are the different system architecture vulnerabilities?

       …………………………………………….…………………………………...
       …………………………………………….…………………………………
       …………………………………………….…………………………………

2)   What are the counter measures for these system architecture vulnerabilities?

       …………………………………………….…………………………………...
       …………………………………………….…………………………………
       …………………………………………….…………………………………

3)   What are the different procedures of recovery?

       …………………………………………….…………………………………...
       …………………………………………….…………………………………
       …………………………………………….…………………………………

2.3   SECURITY AND SERVLET

In this section, we will look at the security issues related to Java and its environment.

Java Security

In Java Security, there is a package, java.security.acl, that contains several classes that you can use to establish a security system in Java. These classes enable your development team to specify different access capabilities for users and user groups. The concept is fairly straightforward. A user or user group is granted permission to functionality by adding that user or group to an access control list.

For example, consider a java.security.Principal called testUser as shown below:

Principal testUser = new PrincipalImpl ("testUser");

Now, you can create a Permission object to represent the capability of reading from a file.

Permission fileRead = new PermissionImpl ("readFile");

Once, you have created the user and the user's permission, you can create the access control list entry. Its important to note that the security APIs require that the owner of the access list be passed in order to ensure that this is truly the developer's desired action. It is essential that this owner object be protected carefully.

Acl accessList = new AclImpl (owner, "exampleAcl"); In its final form, the access list will contain a bunch of access list entries.

You can create these as follows:

AclEntry aclEntry = new AclEntryImpl (testUser);

aclEntry.addPermission(fileRead);

accessList.addEntry(owner, aclEntry);

The preceding lines create a new AclEntry object for the testUser, add the fileRead permission to that entry, and then add the entry to the access control list. You can now check the user permissions quite easily, as follows:

boolean isReadFileAuthorised = accessList.checkPermission(testUser, readFile);

🗐 Check Your Progress 2

1)   Explain security provided by java?

…………………………………………………………………………...

…………………………………………………………………………

…………………………………………………………………………

2.4 FORM BASED CUSTOM AUTHENTICATION

In this section, we will examine how to use form to authenticate clients.

2.4.1 Use of Forms to Authenticate Clients

A common way for servlet-based systems to perform authentication is to use the session to store information indicating that a user has logged into the system. In this scheme, the authentication logic uses the HttpSession object maintained by the servlet engine in the Web server.

A base servlet with knowledge of authentication is helpful in this case. Using the service method of the BaseServlet, the extending servlets can reuse the security Check functionality.

The service method is shown in the following sample code snippet:

```
  Public void service(HttpServletRequest request, HttpServletResponse
response)   throws IOException, ServletException
   {


    // check if a session has already been created for this user don't create a new session
     HttpSession session = request.getSession( false);
     String requestedPage = request.getParameter(Constants.REQUEST);
     if ( session != null)
     {
       // retrieve authentication parameter from the session
       Boolean isAuthenticated = (Boolean)
       session.getValue(Constants.AUTHENTICATION);
       // Check if the user is not authenticated
       if ( !isAuthenticated.booleanValue() )
       {
         // process the unauthenticated request
         unauthenticatedUser(response, requestedPage);
       }


     }
     else // session does not exist
     {
      // therefore the user is not authenticated process the unauthenticated request
       unauthenticatedUser(response, requestedPage);
      }
    }
```

The BaseServlet attempts to retrieve the session from the servlet engine. On retrieval, the servlet verifies that the user has been granted access to the system. Should either of these checks fail, the servlet redirects the browser to the login screen. On the login screen, the user is prompted to give a username and password. Note that, the data passed from the browser to the Web server is unencrypted unless you use Secure Socket Layer (SSL).

The LoginServlet uses the username/password combination to query the database to ensure that this user does indeed have access to the system. If, the check fails to return a record for that user, the login screen is redisplayed. If, the check is successful, the following code stores the user authentication information inside a session variable.

```
// create a session
session = request.getSession(true);


// convert the boolean to a Boolean
 Boolean booleanIsAuthenticated = new Boolean ( isAuthenticated);


// store the boolean value to the session
session.putValue(Constants.AUTHENTICATION,  booleanIsAuthenticated);
```

This example assumes that any user who successfully authenticates to the system has access to the pages displayed prior to login.

Providing Security through EJB's in Java

In the EJB's deployment descriptor, the following code identifies the access control

entries associated with the bean:

(accessControlEntries

DEFAULT [administrators basicUsers]

theRestrictedMethod [administrators]

); end accessControlEntries

Administrators have access to the bean by default and constitute the only user group that has access to theRestrictedMethod. Once you've authorised that the administrators have access to the bean, you now need to create properties detailing which users are in the administrators group.

For this, the weblogic.properties file must include the following lines:

weblogic.password.SanjayAdministrator=Sanjay

weblogic.security.group.administrators=SanjayAdministrator

weblogic.password.User1Basic=User1

weblogic.security.group.basicUsers=User1Basic

The above method established the users who have access to the bean and have restricted certain specific methods. This limits the potential for malicious attacks on your Web server to reach the business logic stored in the beans.

The final step in this EJB authorisation is to establish the client connection to the bean. The client must specify the username/password combination properly in order to have access to the restricted bean or methods. For example client communication can be as follows:

```
try{

Properties myProperties = new Properties();

myProperties.put( Context.INITIAL_CONTEXT_FACTORY,

" weblogic.jndi.T3InitialContextFactory");

myProperties.put(Context.PROVIDER_URL, "t3://localhost:7001");

myProperties.put(Context.SECURITY_PRINCIPAL, "Sanjay");

myProperties .put(Context.SECURITY_CREDENTIALS, "san");

ic = new InitialContext(myProperties);

}

catch (Exception e) { ... }
```

Since, you've passed the SanjayAdministrator user to the InitialContext, you'll have access to any method to which the administrators group has been granted access. If, your application makes connections to external beans or your beans are used by external applications, you should implement this security authorisation.

2.4.2 Use Java's Multiple-Layer Security Implementation

The following examples are to demonstrate a complete system approach to the security problem.

- In the first example, the form-based authentication scheme was implemented. It checked only to ensure that the user was listed in the database of authenticated users. A user listed in the database was granted access to all functionality within the system without further definition.

- In the second example, the EJB authorised the user attempting to execute restricted methods on the bean and this protects the bean from unauthorised access, but does not protect the Web application.

- The third example, was that of the Java Security Access Control package is given to explains how to use a simple API to verify that a user has access to a certain functionality within the system. Using these three examples. It is possible to create a simple authentication scheme that limits the user's access to web-based components of a system, including back-office systems.

Delegate Security to the Java Access Control Model

The first step is to create delegate classes to wrap the security functionality contained in the Java Access Control Model classes. By wrapping the method calls and interfaces, the developer can ensure that the majority of the code in the system can function independently of the security implementation. In addition, through the delegation pattern, the remainder of the code can perform security functionality without obtaining specific knowledge of the inner workings of security model.

The first main component of this example is the User. The code that implements the interface can delegate calls to the java.security.Principal interface.

For example, to retrieve a user's telephone number, implement a method called getPhoneNumber(). Another approach to obtaining this user data involves the use of XML.Convert data stored in the database into an XMLDocument from which data could be accessed by walking the tree.

The second main component is interface. The classes that implement this interface use the implementation of the java.security.acl.Permission interface to execute their functionality. In a Web-based system, there is a need to identify both the name of the action and the URL related to that action.

The last major component is the WebSecurityManager object and this is responsible for performing the duties related to user management, features management, and the access control lists that establish the relationships between users and desired features.

WebSecurityManager can be implemented in many ways including, but not limited to, a Java bean used by JSP, a servlet, an EJB, or a CORBA/RMI service. The choice is with system's designer. In this simple example, the WebSecurityManager is assumed to run in the same JVM as the servlets/JSP.

In the following example it is assumed that: first, the information relating the users and their permissions is stored in a relational database; second, this database is already populated.

This example builds off of, the framework detailed in the prior example of servlet-based user authentication. The service method is as under:

```
Public void service (HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
  {

// check  if a session has already been created for this user don't create a new session.
    HttpSession session = request.getSession( false);

    String sRequestedFeature = request.getParameter(Constants.FEATURE);

    if ( session != null)
    {
      // retrieve User object
      User currentUser = (User) session.getValue(Constants.USER);

      Feature featureRequested = null;
      try {
        // get the page from Web Security Manager
        featureRequested = WebSecurityManager.getFeature(
sRequestedFeature);
      } catch ( WebSecurityManagerException smE)
      {
        smE.printStackTrace();
```

```
                }

        if ( WebSecurityManager.isUserAuthenticated( currentUser,
featureRequested) )
        {
          // get page from feature
          String sRequestedPage = featureRequested.getFeaturePath();


          // redirect to the requested page
          response.sendRedirect( Constants.LOGIN2 + sRequestedPage);
        } else {
          // redirect to the error page
          response.sendRedirect( Constants.ERROR + sRequestedFeature);
        }
      } else {
        // redirect to the login servlet (passing parameter)
        response.sendRedirect( Constants.LOGIN2 + sRequestedFeature);
      }
    }
```

In this code, the user is authenticated against the access control list using the requested feature name. The user object is retrieved from the session. The feature object corresponding to the request parameter is retrieved from the SecurityManager object. The SecurityManager then checks the feature against the access control list that was created on the user login through the implementation of the access control list interface.

Upon login, the username/password combination is compared to the data stored in the database. If successful, the User object will be created and stored to the session. The features related to the user in the database are created and added to an access control list entry for the user. This entry is then added to the master access control list for the application. From then on, the application can delegate the responsibility of securing the application to the Java Access Control Model classes.

Here's a code showing how the features are added to the access control list for a given user.

```
  private static void addAclEntry(User user, Hashtable hFeatures)
    throws WebSecurityManagerException
  {
    // create a new ACL entry for this user
    AclEntry newAclEntry = new AclEntryImpl( user);
    // initialize some temporary variables
    String sFeatureCode = null;
    Feature feature = null;
```

```
        Enumeration enumKeys = hFeatures.keys();
        String keyName = null;


        while ( enumKeys.hasMoreElements() )
        {
          // Get the key name from the enumeration
          keyName = (String) enumKeys.nextElement();


          // retrieve the feature from the hashtable
          feature = (Feature) hFeatures.get(keyName);


          // add the permission to the aclEntry
          newAclEntry.addPermission( feature );
        }
        try {
          // add the aclEntry to the ACL for the _securityOwner
          _aclExample.addEntry(_securityOwner, newAclEntry);
        } catch (NotOwnerException noE)
        {
          throw new WebSecurityManagerException("In addAclEntry", noE);
        }
      }
```

The addAclEntry method is passed a User object and an array of Feature objects.
Using these objects, it creates an AclEntry and then adds it to the Acl used by the
application. It is precisely this Acl that is used by the BaseServlet2 to authenticate the
user to the system.

Conclusion

Securing a Web system is a major requirement this section has put forth a security
scheme that leverages the code developed by Sun Microsystems to secure objects in
Java. Although this simple approach uses an access control list to regulate user access
to protected features, you can expand it based on your requirements.

⬚ Check Your Progress 3

1)   Explain form based custom authentication methods used by servlets and EJB.
      ……………………………………………………………………………………...
      ……………………………………………………………………………………
      ……………………………………………………………………………………
2)   What are the advantages of using Java's multiple-layer security
      implementation?
      ……………………………………………………………………………………...
      ……………………………………………………………………………………
      ……………………………………………………………………………………

2.5   RETRIEVING SSL AUTHENTICATION

The following section discuss the issues relating to SSL authentication.

2.5.1   SSL Authentication

SSL uses certificates for authentication — these are digitally signed documents which bind the public key to the identity of the private key owner. Authentication happens at connection time, and is independent of the application or the application protocol.

Certificates are used to authenticate clients to servers, and servers to clients; the mechanism used is essentially the same in both cases. However, the server certificate is mandatory — that is, the server must send its certificate to the client — but the client certificate is optional: some clients may not support client certificates; other may not have certificates installed. Servers can decide whether to require client authentication for a connection.

A certificate contains

- Two distinguished names, which uniquely identify the issuer (the certificate authority that issued the certificate) and the subject (the individual or organisation to whom the certificate was issued). The distinguished names contain several optional components:
  - o   Common name
  - o   Organisational unit
  - o   Organisation
  - o   Locality
  - o   State or Province
  - o   Country
- A digital signature. The signature is created by the certificate authority using the public-key encryption technique:
  - i)   A secure hashing algorithm is used to create a digest of the certificate's contents.
  - ii)   The digest is encrypted with the certificate authority's private key.
       The digital signature assures the receiver that no changes have been made to the certificate since it was issued:

    a) The signature is decrypted with the certificate authority's public key.
    b) A new digest of the certificate's contents is made, and compared with the decrypted signature. Any discrepancy indicates that the certificate may have been altered.

- The subject's domain name. The receiver compares this with the actual sender of the certificate.
- The subject's public key.

SSL Encryption

The SSL protocol operates between the application layer and the TCP/IP layer. This allows it to encrypt the data stream itself, which can then be transmitted securely, using any of the application layer protocols.

Two encryption techniques are used:

- Public key encryption is used to encrypt and decrypt certificates during the SSL handshake.
- A mutually agreed symmetric encryption technique, such as DES (data encryption standard), or triple DES, is used in the data transfer following the handshake.

The SSL Handshake

The SSL handshake is an exchange of information that takes place between the client and the server when a connection is established. It is during the handshake that client and server negotiate the encryption algorithms that they will use, and authenticate one another. The main features of the SSL handshake are:

- The client and server exchange information about the SSL version number and the cipher suites that they both support.

- The server sends its certificate and other information to the client. Some of the information is encrypted with the server's private key. If , the client can successfully decrypt the information with the server's public key, it is assured of the server's identity.

- If, client authentication is required, the client sends its certificate and other information to the server. Some of the information is encrypted with the client's private key. If, the server can successfully decrypt the information with the client's public key, it is assured of the client's identity.

- The client and server exchange random information which each generates and which is used to establish session keys: these are symmetric keys which are used to encrypt and decrypt information during the SSL session. The keys are also used to verify the integrity of the data.

🗗 Check Your Progress 4

1) What do you understand by SSL Authentication?
   …………………………………………………………………………...
   …………………………………………………………………………
   …………………………………………………………………………

2.5.2   Using SSL Authentication in Java Clients

2.5.2.1 JSSE (Java Secure Socket Extesnsion) and Web Logic Server

JSSE is a set of packages that support and implement the SSL and TLS v1 protocols, making those capabilities available. BEA WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between Web Logic Server clients and servers, Java clients, Web browsers, and other servers. Web Logic Server's Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients. Other JSSE implementations can be used for their client-side code outside the server as well.

The following restrictions apply when using SSL in WebLogic server-side applications:

- The use of third-party JSSE implementations to develop WebLogic server applications is not supported. The SSL implementation that WebLogic Server

uses is static to the server configuration and is not replaceable by user applications. You cannot plug different JSSE implementations into WebLogic Server to have it use those implementations for SSL.

• The WebLogic implementation of JSSE does support JCE Cryptographic Service Providers (CSPs), however, due to the inconsistent provider support for JCE, BEA cannot guarantee that untested providers will work out of the box. BEA has tested WebLogic Server with the following providers:

i) The default JCE provider (SunJCE provider) that is included with JDK

ii) The nCipher JCE provider.

WebLogic Server uses the HTTPS port for SSL. Only SSL can be used on that port. SSL encrypts the data transmitted between the client and WebLogic Server so that the username and password do not flow in clear text.

2.5.2.2 Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass on credentials to the WebLogic Server. A Java client establishes a connection with WebLogic Server by getting a JNDI InitialContext. The Java client then, uses the InitialContext to look up the resources it needs in the WebLogic Server JNDI tree.

To specify a user and the user's credentials, set the JNDI properties listed in the Table A.

Table A : JNDI Properties

JNDI Properties	Meaning
INITIAL_CONTEXT_FACTORY	Provides an entry point into the WebLogic Server environment. The class weblogic.jndi.WLInitialContextFactory is the JNDI SPI for WebLogic Server.
PROVIDER_URL	Specifies the host and port of the WebLogic Server that provides the name service.
SECURITY_PRINCIPAL	Specifies the identity of the user when that user authenticates the required information to the default (active) security realm.

These properties are stored in a hash table that is passed to the InitialContext constructor. Notice the use of t3s, which is a WebLogic Server proprietary version of SSL. t3s uses encryption to protect the connection and communication between WebLogic Server and the Java client.

The following Example demonstrates how to use one-way SSL certificate authentication in a Java client.

Example1.0: Example of One-Way SSL Authentication Using JNDI
...
```
Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3s://weblogic:7002");
```

```
env.put(Context.SECURITY_PRINCIPAL, "javaclient");
env.put(Context.SECURITY_CREDENTIALS, "javaclientpassword");
ctx = new InitialContext(env);
```

2.5.2.3 SSL Certificate Authentication Development Environment

SSL Authentication APIs

To implement Java clients that use SSL authentication on WebLogic Server, you use a combination of Java SDK application programming interfaces (APIs) and WebLogic APIs.

Table B lists and describes the Java SDK APIs packages used to implement certificate authentication. The information in Table B is taken from the Java SDK API documentation and annotated to add WebLogic Server specific information. For more information on the Java SDK APIs, Table C lists and describes the WebLogic APIs used to implement certificate authentication.

Table B: Java SDK Certificate APIs Java SDK Certificate APIs

Java SDK Certificate APIs Java SDK Certificate APIs	Description
javax.crypto	This package provides the classes and interfaces for cryptographic operations. The cryptographic operations defined in this package include encryption, key generation and key agreement, and Message Authentication Code (MAC) generation.
	Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects.
	Many classes provided in this package are provider-based (see the java.security.Provider class). The class itself defines a programming interface to which applications may be written. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers may take advantage of any number of provider-based implementations without having to add or rewrite the code.
javax.net	This package provides classes for networking applications. These classes include factories for creating sockets. Using socket factories you can encapsulate socket creation and configuration behaviour.
javax.net.SSL	While the classes and interfaces in this package are supported by WebLogic Server, BEA recommends that you use the weblogic.security.SSL package when you use SSL with WebLogic Server.
java.security.cert	This package provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. It contains support for X.509 v3 certificates and X.509 v2 CRLs.

java.security.KeyStore	This class represents an in-memory collection of keys and certificates. It is used to manage two types of keystore entries:
	• Key Entry : This type of keystore entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorised access.
	Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key.
	Private keys and certificate chains are used by a given entity for self-authentication. Applications for this authentication include software distribution organisations which sign JAR files as part of releasing and/or licensing software.
	• Trusted Certificate Entry : This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate.
	This type of entry can be used to authenticate other parties.
java.security.PrivateKey	A private key. This interface contains no methods or constants. It merely serves to group (and provide type safety for) all private key interfaces.
	Note: The specialised private key interfaces extend this interface. For example, see the DSAPrivateKey interface in java.security.interfaces.
java.security.Provider	This class represents a "Cryptographic Service Provider" for the Java Security API, where a provider implements some or all parts of Java Security, including:
	• Algorithms (such, as DSA, RSA, MD5 or SHA-1).
	• Key generation, conversion, and management facilities (such, as for algorithm-specific keys).
	Each provider has a name and a version number, and is configured in each runtime it is installed in.
	To supply implementations of cryptographic services, a team of developers or a third-party vendor writes the implementation code and creates a subclass of the Provider class.
javax.servlet.http. HttpServletRequest	This interface extends the ServletRequest interface to provide request information for HTTP servlets.
	The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).

| javax.servlet.http.
HttpServletResponse	This interface extends the ServletResponse interface to provide HTTP–specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies. The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).
javax.servlet.
ServletOutputStream	This class provides an output stream for sending binary data to the client. A ServletOutputStream object is normally retrieved via the ServletResponse.getOutputStream() method. This is an abstract class that the servlet container implements. Subclasses of this class must implement the java.io.OutputStream.write(int) method.
javax.servlet.
ServletResponse	This class defines an object to assist a servlet in sending a response to the client. The servlet container creates a ServletResponse object and passes it as an argument to the servlet's service methods (doGet, doPost, and so on).

Table C: Web Logic Certificate APIs WebLogic Certificate APIs

| web Logic Certificate APIs
WebLogic Certificate APIs	Description
weblogic.net.http.
HttpsURLConnection	This class is used to represent a Hyper-Text Transfer Protocol with SSL (HTTPS) connection to a remote object. This class is used to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.
weblogic.security.SSL.
HostnameVerifierJSSE	This interface provides a callback mechanism, so that implementers of this interface can supply a policy for handling the case where the host that's being connected to and the server name from the certificate SubjectDN must match. To specify an instance of this interface to be used by the server, set the class for the custom host name verifier in the Client Attributes fields that are located on the Advanced Options panel under the Keystore and SSL tab for the server (for example, myserver).
weblogic.security.SSL.
TrustManagerJSSE	This interface permits the user to override certain validation errors in the peer's certificate chain and allows the handshake to continue. This interface also permits the user to perform additional validation on the peer certificate chain and interrupt the handshake if need be.
weblogic.security.SSL.
SSLContext	This class holds all the state information shared across all sockets created under that context.
weblogic.security.SSL.
SSLSocketFactory	This class delegates requests to create SSL sockets to the SSLSocketFactory.

SSL Client Application Components

At a minimum, an SSL client application comprises the following components:

- *Java client*

  A Java client performs these functions:

  - Initialises an SSLContext with client identity, a HostnameVerifierJSSE, a TrustManagerJSSE, and a HandshakeCompletedListener.

  - Creates a keystore and retrieves the private key and certificate chain.

  - Uses an SSLSocketFactory, and

  - Uses HTTPS to connect to a JSP served by an instance of WebLogic Server.

- *HostnameVerifier*

  The HostnameVerifier implements the weblogic.security.SSL.HostnameVerifierJSSE interface. It provides a callback mechanism so that implementers of this interface can supply a policy for handling the case where the host that is being connected to and the server name from the certificate Subject Distinguished Name (SubjectDN) must match.

- *HandshakeCompletedListener*

  The HandshakeCompletedListener implements the javax.net.ssl.HandshakeCompletedListener interface. It defines how the SSL client receives notifications about the completion of an SSL handshake on a given SSL connection. It also defines the number of times an SSL handshake takes place on a given SSL connection.

- *TrustManager*

  The TrustManager implements the weblogic.security.SSL.TrustManagerJSSE interface. It builds a certificate path to a trusted root and returns true if it can be validated and is trusted for client SSL authentication.

- *build script (build.xml)*

  This script compiles all the files required for the application and deploys them to the WebLogic Server applications directories.

2.5.2.4 Writing Applications that Use SSL

This section covers the following topics:

- ➢ Communicating Securely From WebLogic Server to Other WebLogic Servers

- ➢ Writing SSL Clients

- ➢ Using Two-Way SSL Authentication

- ➢ Two-Way SSL Authentication with JNDI

- ➢ Using a Custom Host Name Verifier

- ➢ Using a Trust Manager
- ➢ Using an SSLContext
- ➢ Using an SSL Server Socket Factory
- ➢ Using URLs to Make Outbound SSL Connections

Communicating Securely From WebLogic Server to Other WebLogic Servers

You can use a URL object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. The weblogic.net.http.HttpsURLConnection class provides a way to specify the security context information for a client, including the digital certificate and private key of the client.

The weblogic.net.http.HttpsURLConnection class provides methods for determining the negotiated cipher suite, getting/setting a host name verifier, getting the server's certificate chain, and getting/setting an SSLSocketFactory in order to create new SSL sockets.

Writing SSL Clients

This section describes, by way of example, how to write various types of SSL clients. Examples of the following types of SSL clients are provided:

- SSLClient

- SSLSocketClient

- SSLClientServlet

Below, Example 1 shows a sample SSLClient, the relevant explanation is embeddede inside the code for easy understanding of the same.

Example 1: SSL Client Sample Code

```
package examples.security.sslclient;

import java.io.File;
import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.Hashtable;
import java.security.Provider;
import javax.naming.NamingException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.ServletOutputStream;
import weblogic.net.http.*;
import weblogic.jndi.Environment;
/** SSLClient is a short example of how to use the SSL library of
 * WebLogic to make outgoing SSL connections. It shows both how to
 * do this from a stand-alone application as well as from within
 * WebLogic (in a Servlet).
 *
 */
```

```java
public class SSLClient {
  public void SSLClient() {}
  public static void main (String [] argv)
    throws IOException {
    if ((!((argv.length == 4) || (argv.length == 5))) ||
      (!(argv[0].equals("wls")))
      ) {
      System.out.println("example: java SSLClient wls
              server2.weblogic.com 80 443 /examplesWebApp/SnoopServlet.jsp");
      System.exit(-1);
    }
    try {
      System.out.println("----");
      if (argv.length == 5) {
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], argv[4], System.out);
      } else {  // for null query, default page returned...
        if (argv[0].equals("wls"))
          wlsURLConnect(argv[1], argv[2], argv[3], null, System.out);
      }
      System.out.println("----");
    } catch (Exception e) {
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
  }
  private static void printOut(String outstr, OutputStream stream) {
    if (stream instanceof PrintStream) {
      ((PrintStream)stream).print(outstr);
      return;
    } else if (stream instanceof ServletOutputStream) {
      try {
        ((ServletOutputStream)stream).print(outstr);
        return;
      } catch (IOException ioe) {
        System.out.println(" IOException: "+ioe.getMessage());
      }
    }
    System.out.print(outstr);
  }
  private static void printSecurityProviders(OutputStream stream) {
    StringBuffer outstr = new StringBuffer();
    outstr.append(" JDK Protocol Handlers and Security Providers:\n");
    outstr.append("   java.protocol.handler.pkgs - ");
    outstr.append(System.getProperties().getProperty(
                          "java.protocol.handler.pkgs"));
    outstr.append("\n");
    Provider[] provs = java.security.Security.getProviders();
    for (int i=0; i<provs.length; i++)
      outstr.append("   provider[" + i + "] - " + provs[i].getName() +
              " - " + provs[i].getInfo() + "\n");
    outstr.append("\n");
    printOut(outstr.toString(), stream);
  }
  private static void tryConnection(java.net.HttpURLConnection connection,
                          OutputStream stream)
    throws IOException {
```

```
    connection.connect();
    String responseStr = "\t\t" +
                connection.getResponseCode() + " -- " +
                connection.getResponseMessage() + "\n\t\t" +
                connection.getContent().getClass().getName() + "\n";
    connection.disconnect();
    printOut(responseStr, stream);
}
/*
 * This method contains an example of how to use the URL and
 *  URLConnection objects to create a new SSL connection, using
 *  WebLogic SSL client classes.
 */
public static void wlsURLConnect(String host, String port,
                    String sport, String query,
                    OutputStream out) {
  try {
    if (query == null)
      query = "/examplesWebApp/index.jsp";
    // The following protocol registration is taken care of in the
    //  normal startup sequence of WebLogic. It can be turned off
    //  using the console SSL panel.
    //
    // we duplicate it here as a proof of concept in a stand alone
    //  java application. Using the URL object for a new connection
    //  inside of WebLogic would work as expected.
    java.util.Properties p = System.getProperties();
    String s = p.getProperty("java.protocol.handler.pkgs");
    if (s == null) {
      s = "weblogic.net";
    } else if (s.indexOf("weblogic.net") == -1) {
      s += "|weblogic.net";
    }
    p.put("java.protocol.handler.pkgs", s);
    System.setProperties(p);
    printSecurityProviders(out);
    // end of protocol registration
    printOut(" Trying a new HTTP connection using WLS client classes -
        \n\thttp://" + host + ":" + port + query + "\n", out);
    URL wlsUrl = null;
    try {
      wlsUrl = new URL("http", host, Integer.valueOf(port).intValue(), query);
      weblogic.net.http.HttpURLConnection connection =
        new weblogic.net.http.HttpURLConnection(wlsUrl);
      tryConnection(connection, out);
    } catch (Exception e) {
      printOut(e.getMessage(), out);
      e.printStackTrace();
      printSecurityProviders(System.out);
      System.out.println("----");
    }
    printOut(" Trying a new HTTPS connection using WLS client classes -
        \n\thttps://" + host + ":" + sport + query + "\n", out);
    wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(), query);
    weblogic.net.http.HttpsURLConnection sconnection =
      new weblogic.net.http.HttpsURLConnection(wlsUrl);
    // only when you have configured a two-way SSL connection, i.e.
    // Client Certs Requested and Enforced is selected in Two Way Client Cert
    // Behavior field in the Server Attributes
```

```
    // that are located on the Advanced Options pane under Keystore & SSL
    // tab on the server, the following private key and the client cert chain
    // is used.
    File ClientKeyFile  = new File ("clientkey.pem");
    File ClientCertsFile  = new File ("client2certs.pem");
    if (!ClientKeyFile.exists() || !ClientCertsFile.exists())
{
  System.out.println("Error : clientkey.pem/client2certs.pem
              is not present in this directory.");
  System.out.println("To create it run - ant createmycerts.");
  System.exit(0);
}
    InputStream [] ins = new InputStream[2];
    ins[0] = new FileInputStream("client2certs.pem");
    ins[1] = new FileInputStream("clientkey.pem");
    String pwd = "clientkey";
    sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
    tryConnection(sconnection, out);
  } catch (Exception ioe) {
    printOut(ioe.getMessage(), out);
    ioe.printStackTrace();
  }
 }
}
```

Example 2: SSLSocketClient Sample Code

The SSLSocketClient sample demonstrates how to use SSL sockets to go directly to
the secure port to connect to a JSP served by an instance of WebLogic Server and
display the results of that connection. It shows how to implement the following
functions:

- Initializing an SSLContext with client identity, a HostnameVerifierJSSE, and a
  TrustManagerJSSE

- Creating a keystore and retrieving the private key and certificate chain

- Using an SSLSocketFactory

- Using HTTPS to connect to a JSP served by WebLogic Server

- Implementing the javax.net.ssl.HandshakeCompletedListener interface

- Creating a dummy implementation of the
  weblogic.security.SSL.HostnameVerifierJSSE class to verify that the server the
  example connects to is running on the desired host

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import weblogic.security.SSL.HostnameVerifierJSSE;
```

```java
import weblogic.security.SSL.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSession;
import weblogic.security.SSL.SSLSocketFactory;
import weblogic.security.SSL.TrustManagerJSSE;
/**
 * A Java client demonstrates connecting to a JSP served by WebLogic Server
 * using the secure port and displays the results of the connection.
 */
public class SSLSocketClient {
 public void SSLSocketClient() { }
 public static void main (String [] argv)
   throws IOException {
  if ((argv.length < 2) || (argv.length > 3)) {
    System.out.println("usage:   java SSLSocketClient host sslport
                        <HostnameVerifierJSSE>");
    System.out.println("example: java SSLSocketClient server2.weblogic.com 443
                        MyHVClassName");
    System.exit(-1);
   }
   try {
    System.out.println("\nhttps://" + argv[0] + ":" + argv[1]);
    System.out.println(" Creating the SSLContext");
    SSLContext sslCtx = SSLContext.getInstance("https");
    File KeyStoreFile  = new File ("mykeystore");
    if (!KeyStoreFile.exists())
{
  System.out.println("Keystore Error : mykeystore is not present in this
                  directory.");
 System.out.println("To create it run - ant createmykeystore.");
 System.exit(0);
}
    System.out.println(" Initializing the SSLContext with client\n" +
                " identity (certificates and private key),\n" +
                "  HostnameVerifierJSSE, AND NulledTrustManager");
   // Open the keystore, retrieve the private key, and certificate chain
    KeyStore ks = KeyStore.getInstance("jks");
    ks.load(new FileInputStream("mykeystore"), null);
    PrivateKey key = (PrivateKey)ks.getKey("mykey", "testkey".toCharArray());
    Certificate [] certChain = ks.getCertificateChain("mykey");
    sslCtx.loadLocalIdentity(certChain, key);
    HostnameVerifierJSSE hVerifier = null;
    if (argv.length < 3)
      hVerifier = new NulledHostnameVerifier();
    else
      hVerifier = (HostnameVerifierJSSE) Class.forName(argv[2]).newInstance();
    sslCtx.setHostnameVerifierJSSE(hVerifier);
    TrustManagerJSSE tManager = new NulledTrustManager();
    sslCtx.setTrustManagerJSSE(tManager);
    System.out.println(" Creating new SSLSocketFactory with SSLContext");
    SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();
    System.out.println(" Creating and opening new SSLSocket with
                SSLSocketFactory");
   // using createSocket(String hostname, int port)
    SSLSocket sslSock = (SSLSocket) sslSF.createSocket(argv[0],
                                    new Integer(argv[1]).intValue());
    System.out.println(" SSLSocket created");
    sslSock.addHandshakeCompletedListener(new MyListener());
    OutputStream out = sslSock.getOutputStream();
```

```
  // Send a simple HTTP request
  String req = "GET /examplesWebApp/ShowDate.jsp HTTP/1.0\r\n\r\n";
  out.write(req.getBytes());
  // Retrieve the InputStream and read the HTTP result, displaying
  // it on the console
  InputStream   in = sslSock.getInputStream();
  byte buf[] = new byte[1024];
  try
  {
    while (true)
    {
      int amt = in.read(buf);
      if (amt == -1) break;
      System.out.write(buf, 0, amt);
    }
  }
  catch (IOException e)
  {
    return;
  }
  sslSock.close();
  System.out.println(" SSLSocket closed");
  } catch (Exception e) {
    e.printStackTrace();
  }
 }
}
```

Example 3: SSLClientServlet Sample Code

The SSL ClientServlet Sample code is given below. For easy understanding details
are provided inside the code.

```
package examples.security.sslclient;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * SSLClientServlet is a simple servlet wrapper of
 * examples.security.sslclient.SSLClient.
 *
 */
public class SSLClientServlet extends HttpServlet {
 public void service(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   response.setContentType("text/html");
   response.setHeader("Pragma", "no-cache"); // HTTP 1.0
   response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
   ServletOutputStream out = response.getOutputStream();
   out.println("<br><h2>ssl client test</h2><br><hr>");
   String[] target = request.getParameterValues("url");
   try {
```

```
    out.println ("<h3>wls ssl client classes</h3><br>");
    out.println ("java SSLClient wls localhost 7001 7002
              /examplesWebApp/SnoopServlet.jsp<br>");
    out.println("<pre>");
    SSLClient.wlsURLConnect("localhost", "7001", "7002",
                  "/examplesWebApp/SnoopServlet.jsp", out);
    out.println("</pre><br><hr><br>");
  } catch (IOException ioe) {
    out.println ("<br><pre> "+ioe.getMessage () +"</pre>");
    ioe.printStackTrace ();
  }
 }
}
```

Using Two-Way SSL Authentication

When using certificate authentication, WebLogic Server sends a digital certificate to the requesting client. The client examines the digital certificate to ensure that it is authentic, has not expired, and matches the WebLogic Server instance that presented it.

With two-way SSL authentication (a form of mutual authentication), the requesting client also presents a digital certificate to WebLogic Server. When the instance of WebLogic Server is configured for two-way SSL authentication, requesting clients are required to present digital certificates from a specified set of certificate authorities. WebLogic Server accepts only digital certificates that are signed by root certificates from the specified trusted certificate authorities.

The following sections describe the different ways two-way SSL authentication can be implemented in WebLogic Server.

- Two-way SSL Authentication with JNDI
- Using Two-way SSL Authentication Between WebLogic Server Instances
- Using Two-way SSL Authentication with Servlets

Two-Way SSL Authentication with JNDI

While using JNDI for two-way SSL authentication in a Java client, use the setSSLClientCertificate() method of the WebLogic JNDI Environment class. This method sets a private key and chain of X.509 digital certificates for client authentication.

For passing digital certificates to JNDI, create an array of InputStreams opened on files containing DER-encoded digital certificates and set the array in the JNDI hash table. The first element in the array, must contain an InputStream opened on the Java client's private key file. The second element, must contain an InputStream opened on the Java client's digital certificate file. (This file contains the public key for the Java client.) Additional elements, may contain the digital certificates of the root certificate authority and the signer of any digital certificates in a certificate chain. A certificate chain allows WebLogic Server to authenticate the digital certificate of the Java client if that digital certificate was not directly issued by a certificate authority registered for the Java client in the WebLogic Server keystore file.

You can use the weblogic.security.PEMInputStream class to read digital certificates stored in Privacy Enhanced Mail (PEM) files. This class provides a filter that decodes the base 64-encoded DER certificate into a PEM file.

Example 4: Example of a Two-Way SSL Authentication Client That Uses JNDI

This example demonstrates how to use two-way SSL authentication in a Java client.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.jndi.Environment;
import weblogic.security.PEMInputStream;
import java.io.InputStream;
import java.io.FileInputStream;
Public class SSLJNDIClient
{
 public static void main (String [] args) throws Exception
 {
  Context context = null;
  try {
   Environment env = new Environment ();
   // set connection parameters
   env.setProviderUrl("t3s://localhost:7002");
   // The next two set methodes are optional if you are using
   // a UserNameMapper interface.
   env.setSecurityPrincipal("system");
   env.setSecurityCredentials("weblogic");
   InputStream key = new FileInputStream("certs/demokey.pem");
   InputStream cert = new FileInputStream("certs/democert.pem");
   // wrap input streams if key/cert are in pem files
   key = new PEMInputStream(key);
   cert = new PEMInputStream(cert);
   env.setSSLClientCertificate (new InputStream [] {key, cert});
   env.setInitialContextFactory(Environment.
         DEFAULT_INITIAL_CONTEXT_FACTORY);
   context = env.getInitialContext ();
   Object myEJB = (Object) context. lookup ("myEJB");
  }
  finally {
   if (context != null) context.close ();
  }
 }
}
```

The code in Example 4 generates a call to the WebLogic Identity Assertion provider that implements the weblogic.security.providers.authentication.UserNameMapper interface. The class that implements the UserNameMapper interface returns a user object if the digital certificate is valid. WebLogic Server stores this authenticated user object on the Java client's thread in WebLogic Server and uses it for subsequent authorisation requests when the thread attempts to use WebLogic resources protected by the default (active) security realm.

Example 5: Establishing a Secure Connection to Another WebLogic Server Instance

You can use two-way SSL authentication in server-to-server communication in which one WebLogic Server instance is acting as the client of another WebLogic Server instance. Using two-way SSL authentication in server-to-server communication enables you to have dependable, highly-secure connections, even without the more common client/server environment.

This example shows an example of how to establish a secure connection from a servlet running in one instance of WebLogic Server to a second WebLogic Server instance called server2.weblogic.com.

```
FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream("demokey.pem");
   f[1]= new FileInputStream("democert.pem");
   f[2]= new FileInputStream("ca.pem");
Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");

e.setInitialContextFactory
          ("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

In Example 5, the WebLogic JNDI Environment class creates a hash table to store the following parameters:

- setProviderURL—specifies the URL of the WebLogic Server instance acting as the SSL server. The WebLogic Server instance acting as SSL client calls this method. The URL specifies the t3s protocol which is a WebLogic Server proprietary protocol built on the SSL protocol. The SSL protocol protects the connection and communication between the two WebLogic Servers instances.

- setSSLClientCertificate—specifies a certificate chain to be used for the SSL connection. You use this method to specify an input stream array that consists of a private key (which is the first input stream in the array) and a chain of X.509 certificates (which make up the remaining input streams in the array). Each certificate in the chain of certificates is the issuer of the certificate preceding it in the chain.

- setSSLServerName—specifies the name of the WebLogic Server instance acting as the SSL server. When the SSL server presents its digital certificate to the server acting as the SSL client, the name specified using the setSSLServerName method is compared to the common name field in the digital certificate. In order for hostname verification to succeed, the names must match. This parameter is used to prevent man-in-the-middle attacks.

- setSSLRootCAFingerprint—specifies digital codes that represent a set of trusted certificate authorities. The root certificate in the certificate chain received from the WebLogic Server instance acting as the SSL server has to match one of the fingerprints specified with this method to be trusted. This parameter is used to prevent man-in-the-middle attacks.

Using Two-Way SSL Authentication with Servlets

To authenticate Java clients in a servlet (or any other server-side Java class), you must check whether the client presented a digital certificate and if so, whether the certificate was issued by a trusted certificate authority. The servlet developer is responsible for asking whether the Java client has a valid digital certificate. When developing servlets with the WebLogic Servlet API, you must access information about the SSL connection through the getAttribute () method of the HTTPServletRequest object.

The following attributes are supported in WebLogic Server servlets:

- javax.servlet.request.X509Certificate
  java.security.cert.X509Certificate []—returns an array of the X.509
  certificate.

- javax.servlet.request.cipher_suite—returns a string representing the cipher
  suite used by HTTPS.

- javax.servlet.request.key_size— returns an integer (0, 40, 56, 128, 168)
  representing the bit size of the symmetric (bulk encryption) key algorithm.

- weblogic.servlet.request.SSLSession
  javax.net.ssl.SSLSession—returns the SSL session object that contains the
  cipher suite and the dates on which the object was created and last used.

You have access to the user information defined in the digital certificates. When you
get the javax.servlet.request.X509Certificate attribute, it is an array of the
java.security.cert X.509 certificate. You simply cast the array to that and examine the
certificates.

A digital certificate includes information, such as the following:

- The name of the subject (holder, owner) and other identification information
  required to verify the unique identity of the subject, such as the uniform
  resource locator (URL) of the Web server using the digital certificate, or an
  individual user's e-mail address.

- The subject's public key.

- The name of the certificate authority that issued the digital certificate.

- A serial number.

- The validity period (or lifetime) of the digital certificate (as defined by a start
  date and an end date).

Example 6: Host Name Verifier Sample Code

A host name verifier validates that the host to which an SSL connection is made is the
intended or authorised party. A host name verifier is useful when a WebLogic client
or a WebLogic Server instance is acting as an SSL client to another application server.
It helps prevent man-in-the-middle attacks.

The following program verify the host name.
Package examples.security.sslclient;
/**
 * HostnameVerifierJSSE provides a callback mechanism so that
 * implementers of this interface can supply a policy for handling
 * the case where the host that's being connected to and the server
 * name from the certificate SubjectDN must match.
 *
 * This is a null version of that class to show the WebLogic SSL
 * client classes without major problems. For example, in this case,
 * the client code connects to a server at 'localhost' but the
 * democertificate's SubjectDN CommonName is 'bea.com' and the
 * default WebLogic HostnameVerifierJSSE does a String.equals () on

```
 * those two hostnames.
 *
 */
Public class NulledHostnameVerifier implements
             weblogic.security.SSL.HostnameVerifierJSSE {
 public boolean verify(String urlHostname, String certHostname)
 {
   return true;
 }
}
```

Example 7: TrustManager Code Example

The weblogic.security.SSL.TrustManagerJSSE interface allows you to override
validation errors in a peer's digital certificate and continue the SSL handshake. You
can also use the interface to discontinue an SSL handshake by performing additional
validation on a server's digital certificate chain.

The following is a example TrustManager

Package examples.security.sslclient;

```
import weblogic.security.SSL.TrustManagerJSSE;
import javax.security.cert.X509Certificate;
Public class NulledTrustManagerJSSE implements TrustManagerJSSE {
 public boolean certificateCallback(X509Certificate[] o, int validateErr) {
   System.out.println(" --- Do Not Use In Production ---\n" + "  By using this " +
             "NulledTrustManager, the trust in the server's identity "+
             "is completely lost.\n -----------------------------");
   for (int i=0; i<o.length; i++)
     System.out.println(" certificate " + i + " -- " + o[i].toString());
   return true;
 }
}
```

The SSLSocketClient example uses the custom trust manager shown above. The
SSLSocketClient shows how to set up a new SSL connection by using an SSL context
with the trust manager.

Using a Handshake Completed Listener

Example 8: HandshakeCompletedListener Code Example

The *javax.net.ssl.HandshakeCompletedListener* interface defines how the SSL client
receives notifications about the completion of an SSL protocol handshake on a given
SSL connection. It also defines the number of times an SSL handshake takes place on
a given SSL connection. Example 8 shows a HandshakeCompletedListener interface
code example. A sample coding is given below:

Package examples.security.sslclient;

```
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Hashtable;
import javax.net.ssl.HandshakeCompletedListener;
import javax.net.ssl.SSLSession;
```

```
 public class MyListener implements HandshakeCompletedListener
  {
 public void handshakeCompleted(javax.net.ssl.
                         HandshakeCompletedEvent event)
   {
    SSLSession session = event.getSession();
    System.out.println("Handshake Completed with peer " +
               session.getPeerHost());
    System.out.println("   cipher: " + session.getCipherSuite());
    javax.security.cert.X509Certificate[] certs = null;
    try
    {
     certs = session.getPeerCertificateChain();
    }
    catch (javax.net.ssl.SSLPeerUnverifiedException puv)
    {
     certs = null;
    }
    if  (certs != null)
    {
     System.out.println("   peer certificates:");
     for (int z=0; z<certs.length; z++) System.out.
          println("certs["+z+"]: " + certs[z]);
    }
    else
    {
     System.out.println("No peer certificates presented");
    }
   }
  }
```

Using an SSLContext

Example 9: SSL Context Code Example

import weblogic.security.SSL.SSLContext;
SSLcontext sslctx = SSLContext.getInstance ("https")
The SSLContext class is used to programmatically configure SSL and retain SSL
session information. For example, all sockets that are created by socket factories
provided by the SSLContext class can agree on session state by using the handshake
protocol associated with the SSL context. Each instance can be configured with the
keys, certificate chains, and trusted root certificate authorities that it needs to perform
authentication. These sessions are cached so that other sockets created under the same
SSL context can potentially reuse them later. For more information on session caching
see SSL Session Behavior in *Managing WebLogic Security*. To associate an instance
of a trust manager class with its SSL context, use the
weblogic.security.SSL.SSLContext.setTrustManagerJSSE () method

Using an SSL Server Socket Factory

Example 10: SSLServerSocketFactory Code Example

Instances of the SSLServerSocketFactory class create and return SSL sockets. This
class extends javax.net.SocketFactory. A sample code is given below:

import weblogic.security.SSL.SSLSocketFactory;
 SSLSocketFactory sslSF = (SSLSocketFactory) sslCtx.getSocketFactoryJSSE();

**Example 11: One-Way SSL Authentication URL Outbound SSL Connection
Class**

You can use a URL object to make an outbound SSL connection from a WebLogic
Server instance acting as a client to another WebLogic Server instance. WebLogic
Server supports both one-way and two-way SSL authentication for outbound SSL
connections.

That Uses Java Classes Only

```java
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.io.IOException;
Public class simpleURL
{
  public static void main (String [] argv)
  {
   if (argv.length != 1)
    {
     System.out.println("Please provide a URL to connect to");
     System.exit(-1);
    }
   setupHandler();
   connectToURL(argv[0]);
  }
  private static void setupHandler()
  {
   java.util.Properties p = System.getProperties();
   String s = p.getProperty("java.protocol.handler.pkgs");
   if (s == null)
     s = "weblogic.net";
   else if (s.indexOf("weblogic.net") == -1)
     s += "|weblogic.net";
   p.put("java.protocol.handler.pkgs", s);
   System.setProperties(p);
  }
  private static void connectToURL(String theURLSpec)
  {
   try
    {
     URL theURL = new URL(theURLSpec);
     URLConnection urlConnection = theURL.openConnection();
     HttpURLConnection connection = null;
     if (!(urlConnection instanceof HttpURLConnection))
      {
       System.out.println("The URL is not using HTTP/HTTPS: " +
                   theURLSpec);
       return;
      }
     connection = (HttpURLConnection) urlConnection;
     connection.connect();
     String responseStr = "\t\t" +
         connection.getResponseCode() + " -- " +
         connection.getResponseMessage() + "\n\t\t" +
           connection.getContent().getClass().getName() + "\n";
     connection.disconnect();
     System.out.println(responseStr);
    }
   catch (IOException ioe)
```

```
    {
      System.out.println("Failure processing URL: " + theURLSpec);
      ioe.printStackTrace();
    }
  }
}
```

WebLogic Two-Way SSL Authentication URL Outbound SSL Connection

Example 12: WebLogic Two-Way SSL Authentication URL Outbound SSL
Connection Code Example

For two-way SSL authentication, the weblogic.net.http.HttpsURLConnection class
provides a way to specify the security context information for a client, including the
digital certificate and private key of the client. Instances of this class represent an
HTTPS connection to a remote object.

```
wlsUrl = new URL ("https", host, Integer.valueOf(sport).intValue(),
          query);
weblogic.net.http.HttpsURLConnection sconnection =
      new weblogic.net.http.HttpsURLConnection(wlsUrl);
InputStream [] ins = new InputStream[2];
    ins[0] = new FileInputStream("client2certs.pem");
    ins[1] = new FileInputStream("clientkey.pem");
    String pwd = "clientkey";
    sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
```

⯐ Check Your Progress 5

1)   Compare and Contrast SSL Authentication in Java Clients.

……………………………………………………………………………...

…………………………………………………………………………………

…………………………………………………………………………………

2)   JSSE and Web Logic Server.

……………………………………………………………………………...

…………………………………………………………………………………

…………………………………………………………………………………

3)   What is JNDI Authentication?Explian with suitable example.

……………………………………………………………………………...

…………………………………………………………………………………

…………………………………………………………………………………

2.6 SUMMARY

Software authentication enables a user to authenticate once and gain access to the
resources of multiple software systems. Securing a Web system is a major
requirement for the development team.

This unit has put forth a security scheme that leverages the code developed by Sun
Microsystems to secure objects in Java. Although this simple approach uses an access
control list to regulate user access to protected features, you can expand it based on

the requirements of your user community to support additional feature-level variations or user information.

2.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) System Architecture Vulnerabilities can result in violations of security policy. This may please flaw in system design, poor security parameters, open ports, poor management, access control vulnerabilities etc. This include covert channel analysis, maintenance hook or back door or trap doors, buffer overflow etc.

2) Code reviews and unit integration testing, Host intrusion detection system, Use file permissions to protect configuration files and sensitive information from being modified, Strict access control, File system encryption, Auditing.

3) Fault-tolerant System, Failsafe system, failsoft or resilient, Failover, and Cold Start.

Check Your Progress 2

1) Discuss java.security.acl class and its features with suitable example. There is a package, java.security.acl, that contains several classes that you can use to establish a security system in Java. These classes enable your development team to specify different access capabilities for users and user groups. The concept is fairly straightforward. A user or user group is granted permission to functionality by adding that user or group to an access control list.

Check Your Progress 3

1) Discuss the example or similar example as shown in 2.4.2.

2) Implementation without knowledge of inner working of the Operating System, platform independence, enhancing security with multiple security features, In addition, a J2EE server without much customisation may support the EJB mapping that was described earlier in the article. Java also provides some other additional methods of security ranging from digital signatures to the JAAS specification that can be used to protect the class files against unauthorized access.

Check Your Progress 4

1) Discuss SSL Authentication that takes place between the Application layer and the TCP/IP layer. The SSL protocol operates between the application layer and the TCP/IP layer. This allows it to encrypt the data stream itself, which can then be transmitted securely, using any of the application layer protocols. In this both symmetric and asymmetric encryption is used.

Check Your Progress 5

1) Hint: Discuss JSSE set of packages that support and implement the SSL. Discuss

2) Web Logic Server's Java Secure Socket Extension (JSSE) implementation can be used by WebLogic clients. Other JSSE implementations can be used for their client-side code outside the server as well.

3) Java clients use the Java Naming and Directory Interface (JNDI) to pass
    credentials to WebLogic Server. A Java client establishes a connection with
    WebLogic Server by getting a JNDI `InitialContext`. The Java client then
    uses the `InitialContext` to look up the resources it needs in the WebLogic
    Server JNDI tree. Please discuss with suitable example.

2.8   FURTHER READINGS/REFERENCES

*   Stalling William, *Cryptography and Network Security, Principles and Practice*,
    2000, SE, PE.

*   Daview D. and Price W., *Security for Computer Networks,* New York:Wiley,
    1989.

*   Chalie Kaufman, Radia Perlman, Mike Speciner, *Network Security,*
    Pearson Education.

*   B. Schnier, *Applied Cryptography*, John Wiley and Sons

*   Steve Burnett & Stephen Paine, *RSA Security's Official Guide to Practice*, SE,
    PE.

*   Dieter Gollmann, *Computer Security*, John Wiley & Sons.

Reference websites:

*   *World Wide Web Security FAQ:*
    http://www.w3.org/Security/Faq/www-security-faq.html

*   *Web Security*: http://www.w3schools.com/site/site_security.asp

*   *Authentication Authorisation and Access Control:*
    http://httpd.apache.org/docs/1.3/howto/auth.html

*   *Basic Authentication Scheme:*
    http://en.wikipedia.org/wiki/Basic_authentication_scheme

*   *OpenSSL Project:* http:/www.openssl.org

*   *Request for Comments 2617 :* http://www.ietf.org/rfc/rfc2617.txt

*   *Sun Microsystems Enterprise JavaBeans Specification:*
    http://java.sun.com/products/ejb/docs.html.

*   *Javabeans Program Listings*: http:/e-docs.bea.com

UNIT 3 CASE STUDY

3.0 INTRODUCTION

The XYZ Bank has embarked on a significant strategic programme, which aims to replace major parts of the Bank's technology. The existing infrastructure has been designed as 'stove-pipes' for each delivery channel, which is restricting the flexibility of on-going developments. The Bank has now reached the position, where it cannot sustain further developments, which are required in support of the Bank's business strategy particularly with regards to the growth of, and interaction with, customers, products and channels. Most of the current applications are running on legacy applications.

The XYZ bank is looking for a solution, which will establish a platform that will allow longer-term technology implications. The scope of the solution covers the existing business functionality that is available within the Internet Banking System, and the bank's teller application that is available within the Branch Network. The scope of the project is restricted to 'migrate the existing functionality' onto a new infrastructure and architecture, and does not allow the introduction of any new business functionality.

The key business drivers are:

- To deliver an effective platform for multi-channel access to Personal Banking information processing services that will support effective integration of the Bank.

- To deliver an infrastructure that has a lifespan and lifecycle in line with the Bank's strategic requirements and that, which reflects the strategic nature of the investment.

- To provide improved capacity for any future developments of browser-based, Personal banking information processing services.

- To provide an environment that facilitates the introduction of the Euro currency in terms of identifying the particular currency that is tendered.

- To make use of acknowledged industry standards, technologies and best practices.

- To have regard for the overall cost of ownership of the solution, covering both development costs and ongoing lifetime costs.

- There is no scope for significant core systems (mainframe) re-engineering work to be undertaken during the lifecycle of the project.

3.1   SOLUTION OVERVIEW

The solution provides an 'Integration Layer' that enables multi-channel access from the Bank's browser-based delivery channels, both internal and external, to its main Personal Banking information processing services, primarily running on the Banks existing mainframe environment. It supports account applications and account transaction services for customer and internal staff, accessed over the Internet or via the bank's intranet. The solution is designed to run on a completely new set of mid-tier devices, using the IBM WebSphere product stack running on IBM's AIX operating system, and is based entirely on J2EE development technologies.

Key Issues

- The need to provide a robust, flexible and future proof (ten years) mechanism for internal and external multi-channel access to Bank Account Applications and Transactions.

- Addressing this issue would be the foundation of the architecture and design of the solution.

- The software architecture has been designed to support this, but may be limited by incompatibilities in the infrastructure, information and functional architectures.

- The need to adopt new working methods and skills in support of component-based iterative development.

- The team have been trained and have received external consultancy in the use of new methods, techniques and tools.

- The need to deliver effective documentation for the project and the use of sub systems and components that may also be used in other solutions.

- Currently there is little design documentation apart from the source code and configuration scripts, although this has begun to be addressed.

- The need to re-use as much as possible of the existing systems.

- Whilst the multi-channel integration layer is based on new technology, the majority of the business function processing and information storage remains on the mainframe. In addition, both customers and Bank staff can access and use the solution via their existing workstations and network infrastructure.

Key Risks

- The previous technology has no future development path and has reducing levels of support.

- To be a addressed by extending the support contracts with the current provider to cover the expected time until the future iterations enable complete replacement of the old solution.

- The proposed set of products and configurations is relatively complex and untested.

- The complete set of skills needed to deliver the solution is in short supply within the industry.

3.2   SOLUTION ARCHITECTURE VIEW

In essence, the solution provides an 'Integration Layer' that enables multi-channel access from the Bank's delivery channels, both internal and external, to the main Personal Banking information processing services, primarily running on the existing Bank mainframe environment. It supports account applications and account transaction services to customers, as well as providing a mechanism for customers and internal staff to communicate in a secure manner.

The high-level software architecture (see *Figure 1*) is based on the bank's Software Layering Model, with the solution being constructed in a modular fashion and implementing discrete responsibilities for Presentation, Business Process, EAI and Business Function.
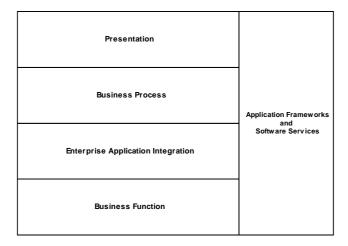
Figure 1: Software layering diagram

The Presentation layer only supports browser-based channels, via the delivery of HTML generated by Java Server Pages.

The term 'Integration Layer' loosely refers to the Business Process and EAI layers, which are based entirely on J2EE development technologies. Without exception, all the business processes that have been developed are 'single-shot' tasks and there has been no requirement for any long-running processes that would have necessitated some form of business process engine (workflow).

Most of the Business Function layer already existed on the mainframe, though it has been re-developed (where economically viable) to provide increased modularity and has been made available via a generic CICS transaction interface that calls specific business functions.

3.3   PRESENTATION LAYER

Figure 2 identifies the significant software components and indicates how these map to the high-level software packages (shown in grey) that are described within the bank's Software Layering Model.
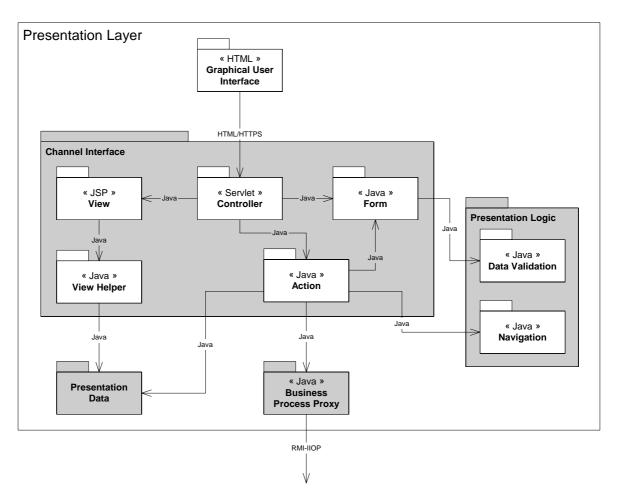
Figure 2: Presentation layer software context diagram

The Struts framework (from the open source Apache Software Foundation) should be used as the basis for the presentation layer. Struts utilises open standard technologies and encourages software architectures based on the Model-View-Controller (MVC) Model 2 design pattern. The framework was selected since it is one of the most widely adopted web presentation frameworks available off-the-shelf and if significantly reduces the development effort required on the project. As this is an open source framework there is a risk that a lack of a formal support contract from a supplier could lead to production issues if any significant bugs are found in the Struts code. This risk is considered acceptable since, the source code is available for update if required and the Java skills needed are prevalent on the project and will remain in the production support teams.

Controller

Controller components are responsible for receiving requests from the browser, deciding what business logic function is to be performed, and then delegating responsibility for producing the next stage of the user interface to an appropriate View component.

Within Struts, the primary component of the Controller is a command design pattern implemented as a servlet of class *ActionServlet*. This servlet is configured by defining a set of *ActionMappings*, which in turn define paths that are matched against the incoming requests and usually specifies the fully qualified class name of an Action component.

Form

Form components represent presentation layer state at a session or request level (not at a persistent level).

Within Struts, the Form components are implemented as *ActionForm* beans.

View

View components are responsible for rendering the particular user interface layout, acquiring data from the user and translating any user actions into events that can be handled by the Controller.

Within Struts, the View components are implemented as Java Server Pages (JPSs). In addition to using the Struts framework, the Tiles framework (bundled as a set of extensions to the Struts code) is also used. The Tiles build on the "include" feature provided by the Java Server Pages specification to provide a full-featured, robust framework for assembling presentation pages from component parts. Each part ("Tile") can be re-used as often as needed throughout the application, which reduces the amount of mark-up that needs to be maintained and makes it easier to change the look and feel of the application.

View Helper

View Helper components encapsulate access to the business logic and data that is accessed and/or manipulated by the views. This facilitates component reuse and allows multiple views to leverage a common 'helper' to retrieve and adapt similar business data for presentation in multiple ways.

A discrete View Helper has not been implemented as such, however a number of custom tags and utility classes have been developed to support the storage and retrieval of 'customer' data from the HTTP session object.

Data Validation

Data Validation components are responsible for simple field-level validation of user input data, for example type checking and cross-validation of mandatory fields. These components are not responsible for any business related validation, which is handled accordingly within the lower layers.

Validation of any input data is achieved via the use of *ActionForm* beans, though any common validation logic is grouped within the Data Validation package.

Navigation

The Navigation package provides the logic to determine the most appropriate view to be displayed to the user.

Action

Action components act as a facade between the presentation layer and the underlying business logic.

Within Struts, this is achieved using Action classes that invoke specific business processes via the use of Web Service requests through the Business Process Proxy.

Presentation Data

Presentation Data components are responsible for maintaining any state that facilitates the successful operation of the presentation layer logic. For example, the Controller and View Manager must have some way of knowing where the user is within a process, as well as have somewhere to store transient information such as the state of the user session.

Specifically, user session state is maintained within the HTTP session object provided by the appropriate J2EE container. This approach is both mature and proven and conforms to one of the standard session handling techniques as detailed within the J2EE specification.

Session tracking is achieved via the use of non-persistent HTTP cookies, which is a mature, standard for J2EE session tracking.

Business Process Proxy

Business Process Proxy components should act as a local representation of the Business Process Gateway.

The Business Process Gateway itself will be implemented as a command design pattern using an EJB and the Business Process Proxy as Java classes that encapsulate the lookup and instantiation of the gateway, such that the gateway can be remotely located from the proxy.

3.4   BUSINESS PROCESS LAYER

Figure 3 identifies the significant software components and indicates how these map to the high-level software packages that are described within the bank's Software Layering Model.
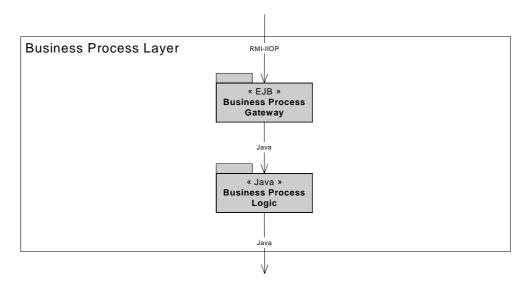
Figure 3: Business process layer software context diagram

Business Process Gateway

Business Process Gateway components will act as a wrapper to the underlying business process logic and translate simple requests into appropriate calls to execute specific business logic. The 'gateway' ensures that a generic interface is offered to clients and insulates clients from the complexities of the specific Business Process Logic interface.

The Business Process Gateway will be implemented as a command design pattern, using a single EJB with a single 'execute' method. The EJB utilises a configuration file that maps each specific request to the appropriate business process logic class and method for execution.

Business Process Logic

Business Process Logic components will implement the flow of work associated with requests for business logic execution, which could be anything from a simple single task to

a complex combination of activities and tasks invoked over long periods and possibly involving many different resources.

There is no requirement to manage process or activity state across multiple tasks (workflow). N.B. An example, exception to this is, the 'logon' activity that invokes multiple tasks, albeit without the need to manage state beyond the boundary of the activity.

The Business Process Logic components will be implemented as Java classes that will inherit from the *AbstractBusinessProcess* class and will implement the *BusinessProcessGatewayInterface*. The classes will map directly to the modelled use cases, with any user interaction (to capture input data for a task) being handled by the presentation layer and the task only being invoked once all the required input data is present. There is then no further interaction with the task until the resulting data is returned to the presentation layer once the task has been completed.

3.5 ENTERPRISE APPLICATION INTEGRATION LAYER

Figure 4 identifies the significant of software components and indicates how these map to the high-level software packages that are described within the bank's Software Layering Model.
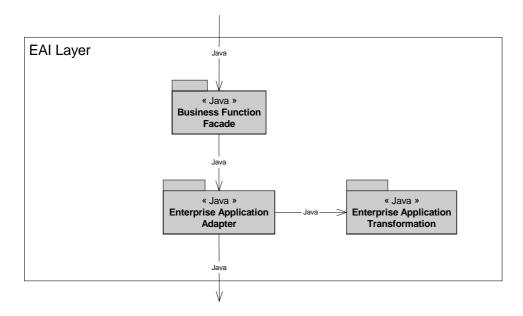

Figure 4: EAI layer software context diagram

Business Function Façade

Business Function Façade components will provide a unified interface to the set of business function interfaces that will be implemented in the underlying enterprise applications, thus enforcing a weak coupling between the façade's clients and the underlying applications.

The façade will be implemented as a single generic interface in the form of a Java class (*Busi*) with a single 'send' method that accepts three parameters, namely the relevant data, the associated context and a string that identifies the particular function to be invoked. Java reflection and configuration files will be then used to determine the particular external resource to communicate with, the format of the message and the expected return types. This implementation, though quite complex, is highly configurable and ensures a weak coupling between the business process layer and the underlying applications.

Enterprise Application Adapter

Enterprise Application Adapter components will provide transport mechanisms to forward requests for business function onto the underlying enterprise applications that implement the requested functions. This decouples the Business Function Façade from the physical implementation of the Business Function, which may be local or remote.

The Enterprise Application Adapter components will be implemented as stateless session EJBs, with a combination of XML configuration files, JNDI, and Java reflection being used to determine and instantiate the particular external resource to communicate with.

Four specific 'adapters' should have been implemented, namely:

- *HostAdapter* to communicate with the mainframe-based retail banking system via the Java Message Service (JMS)
- *DatabaseAdapter* to communicate with the mid-tier DB2 system to via JDBC
- *JavaMailAdapter* to communicate with Message Transfer Agents (MTAs) that support the Simple Message Transfer Protocol (SMTP), via the JavaMail interface
- *DummyHostAdapter* to communicate with a local 'test harness' that mimics the mainframe-based retail banking system when it is not available

Enterprise Application Transformation

Enterprise Application Transformation components will be responsible for transforming messages from one format to another, including translating objects from one development language to another where applicable. Typically such transformations will occur between the EAI layer's native format (Java) and the format supported by the particular underlying enterprise application. These transformations are usually bi-directional, occurring once in either direction.

A comprehensive transformation mechanism has been implemented, based on configurable Templates, to support the following:

- Transforming the requests for business function into the specific message formats expected by the mainframe-based retail banking system and other external resources.

- Transforming specific field types from one format to another, e.g. a Java date field to a Cobol date field.

- Encoding and decoding sensitive data before and after transmission, e.g. the customer PIN and SPI data.

3.6 HIGH LEVEL FUNCTIONAL ARCHITECTURE

The Solution's functional architecture (see *Figure 5)* has been defined to provide a potential set of groupings that could be of benefit in the future, because much of the business function processing is already implemented within the Bank's mainframe that, has been re-used with limited changes. As a result this iteration of the project has not created namespaces to reflect this potential set of groupings.

However, the following section does identify this grouping and then uses it to group the specific business process tasks and business function operations implemented to show how we would have been able to start structuring meaningful business units of work to understand and change the systems base easily.
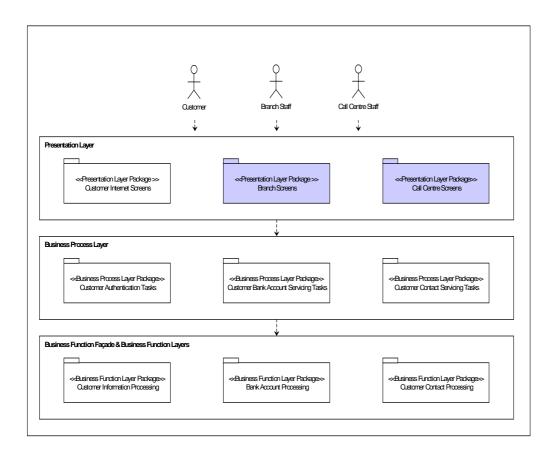
Figure 5:The Solution's Functional Architecture diagram

3.7   PRESENTATION LAYER PACKAGES

The solution was originally focused on delivering screens that could support multiple channels and multiple users. This genericity was quickly relaxed for the different classes of users (customer, branch and call centre) because of the potential benefits of interface optimisation. But the intention remains to re-use as much as possible across each of these user groups.

Within the customer Internet screens package there is additional variation that has been introduced by the need to provide different look and feel for branch and internet banking. This iteration has therefore, focused on the customer Internet screens package and its initial support for internet banking processing.

The presentation layer is logically partitioned into three separate packages of screens:

Customer Internet Screens Package

This package provides the screen interfaces needed for customer Internet access to the Bank Account and Customer Contact Servicing business process.

Branch Screens

This package will provide the screen interfaces needed for branch staff access to the Bank Account and Customer Contact Servicing business process.

Call Centre Screens

This package will provide the screen interfaces needed for call centre staff access to the Bank Account and Customer Contact Servicing business process. The call centre package may be addressed within the project or a separate Call Centre Integration project depending on their development over the coming months and years. A small part of this package has been addressed within this iteration of the project that has delivered the Operator Application for managing secure e-mails.

3.8   BUSINESS PROCESS LAYER PACKAGES

A major objective of the project is to begin to share business processes across different interaction channels and communities of users, where appropriate. The business processes have been implemented as single shot tasks with no processing provided for the creation and state maintenance of long running processes as there was no requirement to manage process or activity state across multiple tasks. Future extension of the definition and management of these processes may require a more expansive approach to process definition and management to be developed.

The set of tasks undertaken, within each business process layer package, are described below. Within this iteration the business process tasks were designed to replicate much of the existing functionality (available using the old technology) rather than to add new capabilities. The business process layer is logically partitioned into three separate packages of business process tasks:

Customer Authentication Tasks

The customer authentication task package implements the following tasks:
* Customer Logon and View Accounts
* Customer Logon, View Accounts and Secure Messages
* Customer Logoff

Customer Bank Account Servicing Tasks

The customer bank account servicing task package implements the following tasks:
* Customer Bank Account Servicing Requests
* Manage Bill Payments
* Manage Funds Transfer
* Manage Customer Requests
* Manage Standing Orders
* Manage Statements

Customer Contact Servicing Tasks

The customer contact servicing task package implements the following tasks:
* Manage Secure Messages

3.9   BUSINESS FUNCTION LAYER PACKAGES

Within the project, the aim was to reuse business function processing on the mainframe providing façade processing (to access the mainframe processing) in the middle tier and adding new business functions in the middle tier for processing. Within this section, the business function operations packages represent both the business function façades and the

business function implementations. The business function layer is logically partitioned into three separate packages of business function operations:

Customer Information Processing

The customer information-processing package implements the following business function operations:

- Customer Enquiry
- Customer Processing

Bank Account Processing

The bank account processing package implements the following business function operations:

- Account Enquiry
- Account Processing
- Customer Request Enquiry
- Customer Request Processing
- Funds Transfer Enquiry
- Funds Transfer Processing
- Bill Payments Enquiry
- Bill Payments Processing
- Statement Service
- Standing Order Enquiry
- Standing Order Processing

3.10 SPECIFIC SOLUTION USABILITY RELATED ELEMENTS

The main elements of the solution that support usability are the following:

- screen designs,
- task dialogue design,
- responsiveness and controls built into the mainframe implementation of business function, and
- conformance to the DDA.

The first three elements described above were developed in response to the specific behavioural and performance needs identified by the use cases that described the user interaction with the system. Each use case involves the interaction with a system interface (such as a screen), the invocation of a business task and the execution of a business function. The solution has been designed to enable each use case to provide usable interaction for the solution users.

Learnability

The solution needs to be easy to learn such that an external user can effectively use the different channels available when they first use the system. The solution should be designed to be easy to learn. It is intuitive to the degree that someone using it for the first time can find his/her way around the system with little or no assistance. Help screens are provided to assist the users.

Likeability

The different Bank channels of interaction with customers pride themselves on enhancing the customer's view of the Bank and its commitment to people and ethics. The solution would have to be designed to be "attractive" to customers and promote a positive view of the Bank.

Productivity

Customer and staff time is important and the solution needs to assist improvements in productivity. The solution should be designed to minimise the time and effort needed to carry out the tasks enabled by the system. This was somewhat compromised by the decision to move all screen processing to the server to meet other requirements. However, the creation of an effective user interface was managed by early prototyping with the system users to agree on the best solution for the interface requirements.

3.11 SUMMARY

The solution is prescribed along modular lines. It provides both vertical and horizontal scalability. The software should be developed as components with clear responsibilities servicing each of the application architecture layers. The components and objects within these layers are carefully designed following the principles of loose coupling, cohesion and clear management of pre and post conditions.

Components managing the interfaces between each application architecture layer should provide clearly defined APIs to enable the flexible combination of functionality offered by each layer, and localisation of the impact of change to those components. This provides the basis for managed extendibility of the solution.