Q.1 a. What is DHTML? What are the features of DHTML? Explain the Working of DHTML.

Answer:

DHTML stands for Dynamic HTML. The first thing that we need to clear about DHTML is that it is neither a language like HTML, JavaScript etc. nor a web standard. It is just a combination of HTML, JavaScript and CSS. It just uses these languages features to build dynamic web pages. DHTML is a feature of Netscape Communicator 4.0, and Microsoft Internet Explorer 4.0 and 5.0 and is entirely a "client-side" technology.

Features of DHTML:

- Simplest feature is making the page dynamic.
- Can be used to create animations, games, applications, provide new ways of navigating through web sites.
- DHTML use low-bandwidth effect which enhance web page functionality.
- Dynamic building of web pages is simple as no plug-in is required.
- Facilitates the usage of events, methods and properties and code reuse.

DHTML as it is combination of HTML as well as JavaScript and it uses the HTML to render the text on the browser. It also uses DOM dynamic Object Model which renders its every element as an Object that can be changed by setting properties and methods. This is done with the help of Scripting. DHTML is part of the general computing trend of late '90s. This is a trend different from structured programming, with a focus on actions and toward object-based programming, where the objects can be compared to nouns in our language.

DHTML allows scripting languages to change variable in a web page's definition language, which in turn affects the look and function of otherwise "static" HTML page content, after the page has been fully loaded and during the viewing process.

b. Explain the role of SMTP, IMAP and POP3.

Answer:

SMTP Simple Mail Transfer Protocol :-- allows two mail servers to communicate using a simple language, and provides a step-by-step protocol for exchanging information.

IMAP Internet Mail Access Protocol:- It is client/server mail protocol. SMTP delivers mail to a central location, where the user can either log in and read it directly or use a client/server mail protocol to read it remotely. IMAP is keep mail on a remote server and let the user interact with it there.

POP3 Post Office Protocol:- POP3 is forward a user's mail to a single machine, where the user can go offline and read it, if necessary. IPOP3 is mainly used when connection is slow.

c. **What do you understand about XML Namespaces and Linking? How do you define SAX in XML?**

Answer:
XML Namespaces:
XML give as facility to make our own tags. Suppose if we want add element of two XML sheets their may be conflicts. But if we made our own tags than it can solve the problem of data conflicts, because different tags may contain the same type of data.

XML Linking:
Using XML linking we can create the connection among the XML data means they can communicate each other.
XML Linking(XML Link, XML Pointer & XML Base)
Where,
XML Link is used to insert link into XML Sheets.
XML Pointer is used to link the address of some specific data in XML sheets.
XML Base is used to defining a default reference to external XML resources.
SAX is stands for Simple API for XML. It is a lexical and event-driven interface. Implementation of SAX is fast and efficient but we can't extract information from XML at random. It is good for use when some types of information ewe handle in same way always without knowing that where it execute in document.

How to use SAX in XML:

```
# Use to get a list of known parsers
my $parsers = XML::SAX->parsers();
# Use to add/update a parser
XML::SAX->add_parser(q(XML::SAX::PurePerl));
# Use to remove parser
XML::SAX->remove_parser(q(XML::SAX::Foodelberry));
# Use to save parsers
XML::SAX->save_parsers()
```

d. **How is an Ajax code used across different browsers?**

Answer:
Internet Explorer uses an ActiveXObject while other browsers uses the built-in JavaScript object called XMLHttpRequest.
Example:

```
<html>
<body>
<script type="text/javascript">
function ajaxFunction()
{
  var xmlHttp;
try
{
// Firefox, Opera 8.0+, Safari
xmlHttp=new XMLHttpRequest();
}
catch (e)
{
// Internet Explorer
try
{
xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
}
catch (e)
{
try
{
xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
}
catch (e)
{
alert("Your browser doesn't supported by Ajax!");
return false;
}
}
}
}
</script>
```

e. How is the JMS connection handled by the application server?

Answer:
o use the Struts Spring plugin, add the ContextLoaderPlugIn to your Struts config file (usually struts-config.xml):

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
<set-property property="contextConfigLocation"
        value="/WEB-INF/applicationContext.xml"/>
</plug-in>
```

The "contextConfigLocation" property is the location of the Spring beans configuration file.

For each action that uses Spring, you need to define the action mapping to use org.springframework.web.struts.DelegatingActionProxy and declare a matching (action "path" == bean "name") Spring bean for the actual Struts action. This is an example of an action that requires an instance of UserDatabase:

```
<action path="/logon"
```

```
            type="org.springframework.web.struts.DelegatingActionProxy">
<forward name="success"          path="/logon.jsp"/>
</action>
```

The corresponding Spring bean configuration:

```
<bean id="userDatabase"
class="org.apache.struts.webapp.example.memory.MemoryUserDatabase" destroy-
method="close" />

<bean name="/logon" class="org.apache.struts.webapp.example.LogonAction">
<property name="userDatabase"><ref bean="userDatabase" /></property>
</bean>
```

For more information on the Spring configuration file format, see the Spring beans DTD.

The Struts action org.apache.struts.webapp.example.LogonAction will automatically receive a reference to UserDatabase without any work on its part or references to Spring by adding a standard JavaBean setter:

```
   private UserDatabase database = null;

   public void setUserDatabase(UserDatabase database) {
      this.database = database;
   }
```

f. How does a typical client perform the communication?

Answer:

1. Use JNDI to locate administrative objects.
2. Locate a single ConnectionFactory object.
3. Locate one or more Destination objects.
4. Use the ConnectionFactory to create a JMS Connection.
5. Use the Connection to create one or more Session(s).
6. Use a Session and the Destinations to create the MessageProducers and MessageConsumers needed.
7. Perform your communication.

g. Explain the lifecycle of JSP.

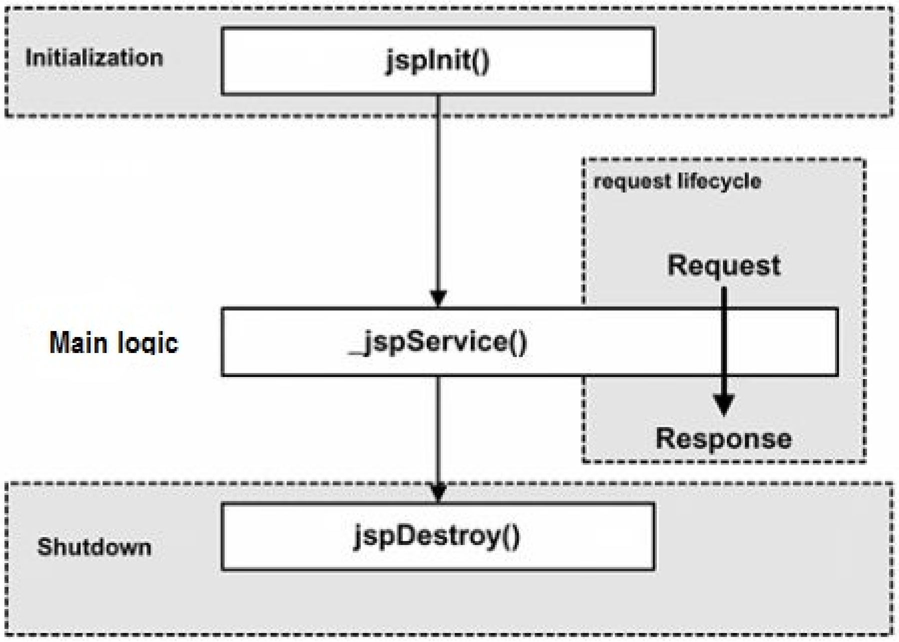
Answer:

A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

The following are the paths followed by a JSP

- Compilation
- Initialization
- Execution
- Cleanup

The four major phases of JSP life cycle are very similar to Servlet Life Cycle and they are as follows:



JSP Compilation:

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps:

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization:

When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method:

```
publicvoid jspInit(){
// Initialization code...
}
```

Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP Execution:

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.
The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

```
void _jspService(HttpServletRequest request,
HttpServletResponse response)
{
// Service handling code...
}
```

The _jspService() method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

JSP Cleanup:

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

The jspDestroy() method has the following form:

```
publicvoid jspDestroy()
{
// Your cleanup code goes here.
}
```

Q.2 a. Write a TCP/IP client and server program.

Answer:

```
import java.net.*;
import java.io.*;

publicclassGreetingClient
{
publicstaticvoid main(String[] args)
{
String serverName = args[0];
int port =Integer.parseInt(args[1]);
try
{
System.out.println("Connecting to "+ serverName
+" on port "+ port);
Socket client =newSocket(serverName, port);
System.out.println("Just connected to "
+ client.getRemoteSocketAddress());
OutputStream outToServer = client.getOutputStream();
DataOutputStreamout=
newDataOutputStream(outToServer);

out.writeUTF("Hello from "
+ client.getLocalSocketAddress());
InputStream inFromServer = client.getInputStream();
DataInputStreamin=
newDataInputStream(inFromServer);
System.out.println("Server says "+in.readUTF());
        client.close();
}catch(IOException e)
{
        e.printStackTrace();
}
}
}
import java.net.*;
import java.io.*;

publicclassGreetingServerextendsThread
{
privateServerSocket serverSocket;

publicGreetingServer(int port)throwsIOException
{
     serverSocket =newServerSocket(port);
     serverSocket.setSoTimeout(10000);
}
publicvoid run()
{
while(true)
{
try
{
System.out.println("Waiting for client on port "+
            serverSocket.getLocalPort()+"...");
Socket server = serverSocket.accept();
System.out.println("Just connected to "
+ server.getRemoteSocketAddress());
DataInputStreamin=
newDataInputStream(server.getInputStream());
System.out.println(in.readUTF());
DataOutputStreamout=
newDataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to "
+ server.getLocalSocketAddress()+"\nGoodbye!");
            server.close();
}catch(SocketTimeoutException s)
{
System.out.println("Socket timed out!");
break;
}catch(IOException e)
{
            e.printStackTrace();
break;
}
}
}
publicstaticvoid main(String[] args)
{
int port =Integer.parseInt(args[0]);
try
{
Thread t =newGreetingServer(port);
          t.start();
}catch(IOException e)
{
          e.printStackTrace();
}
}
}
```

b. Write a servlet to save the state of all the currently loaded servlets.

Answer:

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

public class SaveState 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("Trying to save the state of " + name + "...");
      out.flush();
      try {
        Method save = servlet.getClass().getMethod("saveState", null);
        save.invoke(servlet, null);
        out.println("Saved!");
      }
      catch (NoSuchMethodException e) {
        out.println("Not saved. This servlet has no saveState() method.");
      }
      catch (SecurityException e) {
        out.println("Not saved. SecurityException: " + e.getMessage());
      }
      catch (InvocationTargetException e) {
        out.print("Not saved. The saveState() method threw an exception: ");
        Throwable t = e.getTargetException();
        out.println(t.getClass().getName() + ": " + t.getMessage());
      }
      catch (Exception e) {
        out.println("Not saved. " + e.getClass().getName() + ": " +
                    e.getMessage());
      }

      out.println();
    }
  }

  public String getServletInfo() {
    return "Calls the saveState() method (if it exists) for all the " +
           "currently loaded servlets";
  }
}
```

Q.3 a. Write a program to Fetch all XML Elements of an XML file and count the number of nodes.

Answer:

Fetching

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class fetchrecord {
public static void main(String[] arg){
try {
BufferedReader br = new BufferedReader(
new InputStreamReader(System.in));
System.out.print("Enter the XML File name: ");
String str = br.readLine();
File file = new File(str);
if(file.exists()){

DocumentBuilderFactory fact =
DocumentBuilderFactory.newInstance();

DocumentBuilder build = fact.newDocumentBuilder();
Document docu = build.parse(str);

NodeList list1 = docu.getElementsByTagName("*");
System.out.println("XML Elements: ");
for (int i=0; i<list1.getLength(); i++) {

Element ele = (Element)list1.item(i);
System.out.println(ele.getNodeName());
}
}
else{
System.out.print("File not found!");
}
}
catch (Exception e) {
System.exit(1);
}
}

}
```

//Counting Nodes

```
import org.w3c.dom.*;
import org.apache.xerces.parsers.DOMParser;
import java.io.*;
public class totalnode {
public static void main(String[] args) {
try{
BufferedReader br = new BufferedReader(
new InputStreamReader(System.in));
System.out.print("Enter the file name: ");
String str = br.readLine();
File file = new File(str);
if (file.exists()){
DOMParser pars = new DOMParser();
pars.parser(str);
Document docu = pars.getDocument();
System.out.print("Enter element to count: ");
String ele = br.readLine();
NodeList list1 = docu.getElementsByTagName(ele);
System.out.println("Number of nodes: " + list1.getLength());
}
else{
System.out.println("File not found!");
}
}
catch (Exception e){
e.getMessage();
}
}
}

package org.apache.xerces.parsers;

import org.w3c.dom.Document;

public class DOMParser {

public void parser(String str) {
// TODO Auto-generated method stub
}

public Document getDocument() {
// TODO Auto-generated method stub
return null;
}

}
```

b. What is XSLT? Explain its relationships with XSL. Differentiate between DOM and SAX Parser in Java.

Answer:
XSLT stands for eXtensible Stylesheet Language Transformations. It is a language used to convert XML documents to XHTML or other XML documents. This conversion is done by transforming each XML element into an (X)HTML element.. it uses XPath to find information in a XML document. XSLT is nothing but transforming XSL's. Xpath defines the parts of the source document that must match one or more predefined templates. Once a match is found, XSLT will transform the match into the result document.

1) First and major difference between DOM vs SAX parser is how they work. DOM parser load full XML file in memory and creates a tree representation of XML document, while SAX is an *event based* XML parser and doesn't load whole XML document into memory.

2) For small and medium sized XML documents *DOM is much faster than SAX* because of in memory operation.

3) DOM stands for Document Object Model while SAX stands for Simple API for XML parsing.

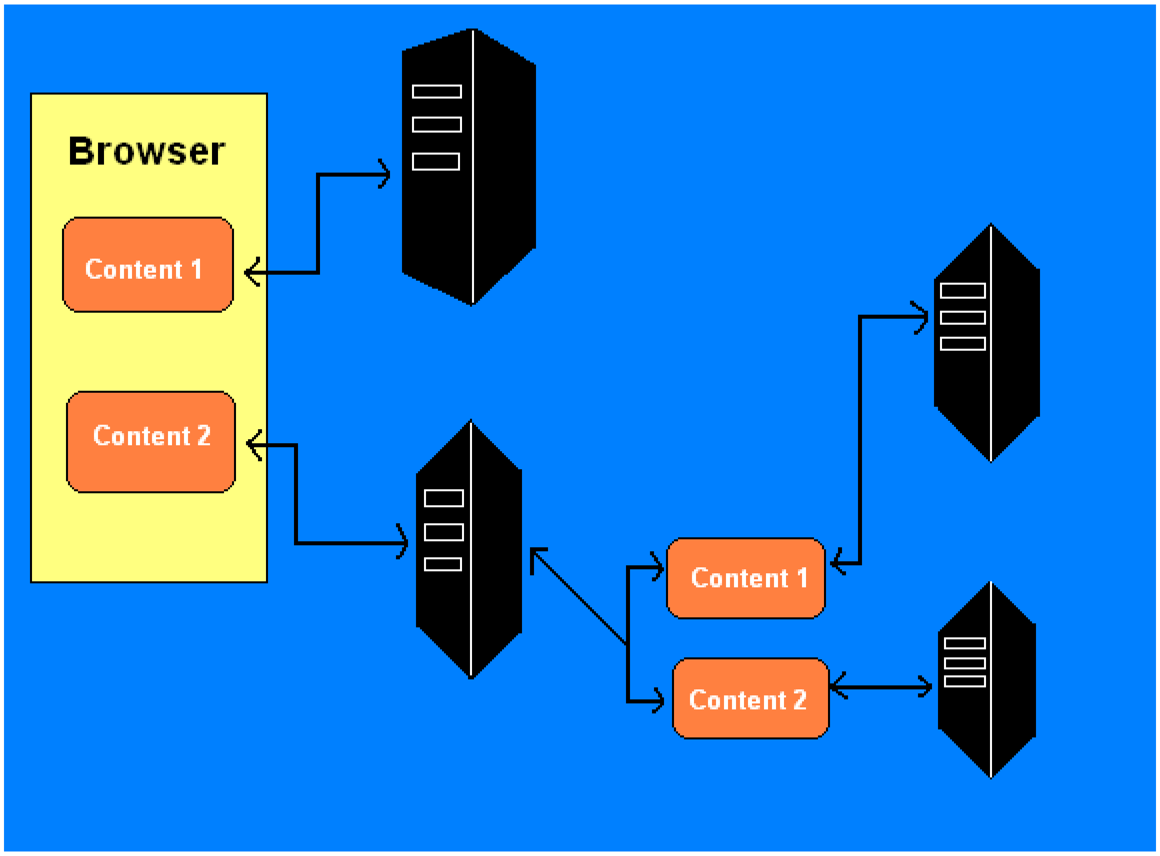
4) Another difference between DOM vs SAX is that, learning where to use DOM parser and where to use SAX parser. DOM parser is better suited for small XML file with sufficient memory, while SAX parser is better suited for large XML files.

Q.4 a. What is the need of Ajax? Explain the working and Architecture of AJAX.

Answer:
In traditional JavaScript coding to get any information from a database or a file on the server, or send user information to a server, its important to make an HTML form and GET or POST data to the server. Wait for the server to respond, then a new page will load with the results when clicking Ok button to send the information. Because the server returns a new page each time . With AJAX, your JavaScript communicates directly with the server, through the JavaScript XMLHttpRequest object With an HTTP request, a web page can make a request to, and get a response from a web server - without reloading the page. The user will stay on the same page, and he will not notice that scripts request

pages, or send data to a server in the background. By using the XMLHttpRequest object, a web developer can update a page with data from the server after the page has loaded. The XMLHttpRequest object is supported in Internet Explorer 5.0+, Safari 1.2, Mozilla 1.0 / Firefox, Opera 8+, and Netscape 7.

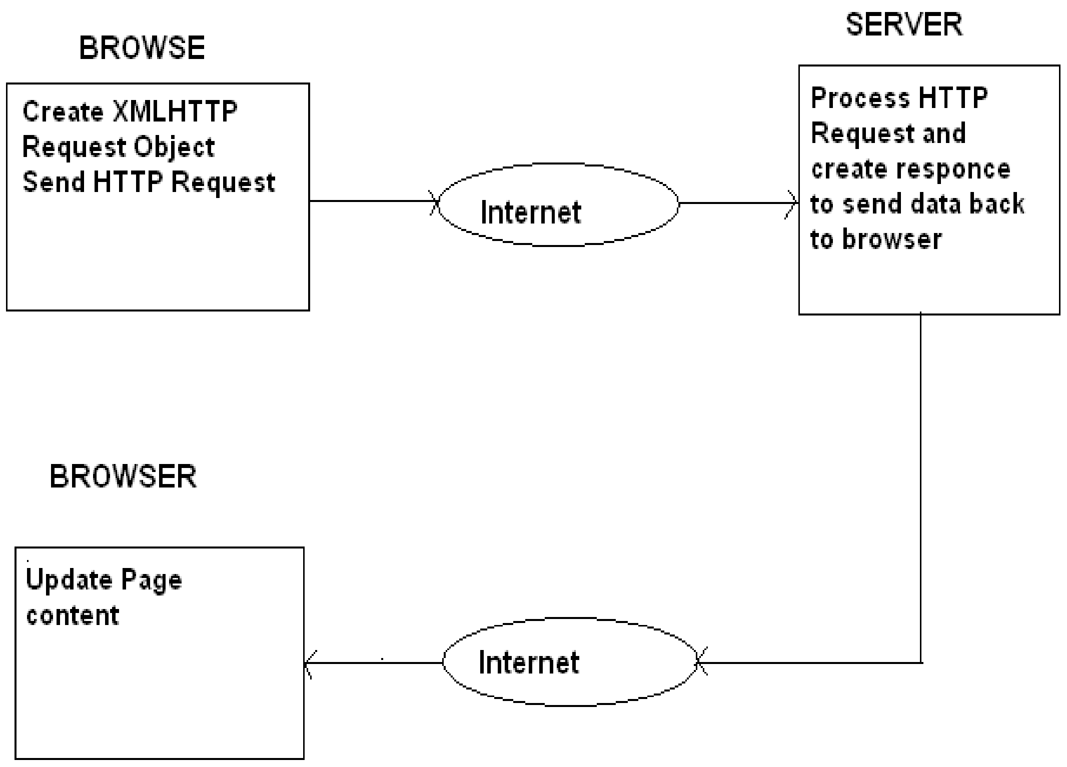
Ajax Architecture-



In Figure there is a browser. The browser has two pieces of content: Content 1 and Content 2. Each piece of content is fetched from a different server. Content 2 is fetched from a server that also has two pieces of content, which are also retrieved from separate servers. From an architectural point of view, Ajax implements the Pipes and Filters pattern

The data is fetched from the server by using a Representational State Transfer (REST) architecture style. The essence of REST is to create a simpler web services architecture by using Hypertext Transfer Protocol (HTTP). REST is used solely for the transfer of data, and in particular is used extensively with Ajax applications. The overall idea is to generate content and to have that content filtered and processed. The filtered and processed content serves as an information basis, where another process acts as a client that filters and processes the information. The filtered and processed information acts as an information basis for another client. Content is fluid and constantly modified.

How Ajax Works?


BROWSE
Create XMLHTTP Request Object Send HTTP Request
Internet
SERVER
Process HTTP Request and create responce to send data back to browser
BROWSER
Update Page content
Internet

b. Write a program to send an image using Ajax.

Answer:
Ajax request Urls of images send as the response and we can set those URLs using DHTML.
Example:
Tn this example an XML document is returned from an AJAX.

```
<categories>
<category>
<cat-id>001</cat-id>
<name>IT</name>
<description>Improve your skill</description>
<image-url>IT_image.gif</image-url>
</category>
<category>
<cat-id>002</cat-id>
<name>Mathematics</name>
<description>Solve problem easily</description>
<image-url>mathematics.gif</image-url>
</category>
</categories>
```

We use image-url element to contain the location of the URL.The callback method that we used in AJAX interaction will parse the response XML document and call the addCategory function.
The addCategory function looks up a table row element "categoryTable" in body of the page and adds a row to the element that is use to contains the image.

```
<scrip type="text/javascript" >
...
function addCategory(id, name, imageSrc) {
var categoryTable = document.getElementById("categoryTable");
var row = document.createElement("tr");
var catCell = document.createElement("td");
var img = document.createElement("img");
img.src = ("images\" + imageSrc);
var link = document.createElement("a");
link.className ="category";
link.appendChild(document.createTextNode(name));
link.setAttribute("onclick", "catalog?command=category&catid=" + id);
catCell.appendChild(img);
catCell.appendChild(link);
row.appendChild(catCell);
categoryTable.appendChild(row);
}
</script>
...
<table>
<tr>
<td width="300" bgoclor="lightGray">
<table id="categoryTable" border="0" cellpadding="0"></table>
</td>
<td id="body" width="100%">Body Here</td>
</tr>
</table>
```

source of the image is set to the image source. And we can load the image using a subsequent HTTP request for the image at the URL. "images/IT_image.gif"
or
"images/Mathematics.gif"
This will occur when you add the image in the categoryTable.

Q.5 a. What are different types of Java Bean? Explain the architecture of EJB.

Answer:
CMP -- Container-managed persistence beans are the simplest for the bean developer to create and the most difficult for the EJB server to support. This is because all the logic for synchronizing the bean's state with the database is handled automatically by the

container. This means that the bean developer doesn't need to write any data access logic, while the EJB server is supposed to take care of all the persistence needs automatically. With CMP, the container manages the persistence of the entity bean. Vendor tools are used to map the entity fields to the database and absolutely no database access code is written in the bean class.

BMP--bean-managed persistence (BMP) enterprise bean manages synchronizing its state with the database as directed by the container. The bean uses a database API to read and write its fields to the database, but the container tells it when to do each synchronization operation and manages the transactions for the bean automatically. Bean-managed persistence gives the bean developer the flexibility to perform persistence operations that are too complicated for the container or to use a data source that is not supported by the container.

EJB Architecture
Session beans:
Session beans are non-persistent enterprise beans. They can be stateful or stateless. A stateful session bean acts on behalf of a single client and maintains client-specific session information (called conversational state) across multiple method calls and transactions. It exists for the duration of a single client/server session. A stateless session bean, by comparison, does not maintain any conversational state. Stateless session beans are pooled by their container to handle multiple requests from multiple clients.

Entity beans:
Entity beans are enterprise beans that contain persistent data and that can be saved in various persistent data stores. Each entity bean carries its own identity. Entity beans that manage their own persistence are called bean-managed persistence (BMP) entity beans. Entity beans that delegate their persistence to their EJB container are called container-managed persistence (CMP) entity beans.

Message-driven beans:
Message-driven beans are enterprise beans that receive and process JMS messages. Unlike session or entity beans, message-driven beans have no interfaces. They can be accessed only through messaging and they do not maintain any conversational state. Message-driven beans allow asynchronous communication between the queue and the listener, and provide separation between message processing and business logic.

Remote client view
The remote client view specification is only available in EJB 2.0. The remote client view of an enterprise bean is location independent. A client running in the same JVM as a bean instance uses the same API to access the bean as a client running in a different JVM on the same or different machine.
Remote interface:
The remote interface specifies the remote business methods that a client can call on an enterprise bean.
Remote home interface:

The remote home interface specifies the methods used by remote clients for locating, creating, and removing instances of enterprise bean classes.

Local client view
The local client view specification is only available in EJB 2.0. Unlike the remote client view, the local client view of a bean is location dependent. Local client view access to an enterprise bean requires both the local client and the enterprise bean that provides the local client view to be in the same JVM. The local client view therefore does not provide the location transparency provided by the remote client view. Local interfaces and local home interfaces provide support for lightweight access from enterprise bean that are local clients. Session and entity beans can be tightly couple with their clients, allowing access without the overhead typically associated with remote method calls.

Local interface:
The local interface is a lightweight version of the remote interface, but for local clients. It includes business logic methods that can be called by a local client.
Local home interface: The local home interface specifies the methods used by local clients for locating, creating, and removing instances of enterprise bean classes.

EJB client JAR file
An EJB client JAR file is an optional JAR file that can contain all the class files that a client program needs to use the client view of the enterprise beans that are contained in the EJB JAR file. If you decide not to create a client JAR file for an EJB module, all of the client interface classes will be in the EJB JAR file.

EJB container
An EJB container is a run-time environment that manages one or more enterprise beans. The EJB container manages the life cycles of enterprise bean objects, coordinates distributed transactions, and implements object security. Generally, each EJB container is provided by an EJB server and contains a set of enterprise beans that run on the server.

Deployment descriptor
A deployment descriptor is an XML file packaged with the enterprise beans in an EJB JAR file or an EAR file. It contains metadata describing the contents and structure of the enterprise beans, and runtime transaction and security information for the EJB container.

EJB server
An EJB server is a high-level process or application that provides a run-time environment to support the execution of server applications that use enterprise beans. An EJB server provides a JNDI-accessible naming service, manages and coordinates the allocation of resources to client applications, provides access to system resources, and provides a transaction service.

b. Explain the following in reference to Spring Framework:
(i) Bean Factory
(ii) Benefits of the Spring Framework transaction management
(iii) Row Call back Handler
(iv) Batch Prepared Statement Setter
(v) Prepared Statement Creator

Answer:
(i) The BeanFactory is provide an advanced configuration mechanism capable of managing beans of any kind of storage facility. The ApplicationContext builds on top of the BeanFactory and adds other functionality like integration with Springs AOP features, message resource handling for use in internationalization, event propagation, declarative mechanisms to create the ApplicationContext and optional parent contexts, and application-layer specific contexts such as the WebApplicationContext, among other enhancements.

Spring Framework supports:
* Programmatic transaction management.
* Declarative transaction management.

(ii) Spring provides a unique transaction management abstraction, which enables a consistent programming model over a variety of underlying transaction technologies, such as JTA or JDBC. Supports declarative transaction management. Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA. Integrates very well with Spring's various data access abstractions.

(iii) RowCallbackHandler interface is used by JdbcTemplate for processing rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of processing each row but don't need to worry about exception handling. SQLExceptions will be caught and handled by the calling JdbcTemplate. RowCallbackHandler object is typically stateful: It keeps the result state within the object, to be available for later inspection.
RowCallbackHandler interface has one method :

void processRow(ResultSet rs) :- Implementations must implement this method to process each row of data in the ResultSet.

(iv) BatchPreparedStatementSetter interface sets values on a PreparedStatement provided by the JdbcTemplate class for each of a number of updates in a batch using the same SQL. Implementations are responsible for setting any necessary parameters. SQL with placeholders will already have been supplied. Implementations of BatchPreparedStatementSetter do not need to concern themselves with SQLExceptions that may be thrown from operations they attempt. The JdbcTemplate class will catch and handle SQLExceptions appropriately.
BatchPreparedStatementSetter has two method:

* int getBatchSize() :- Return the size of the batch.
* void setValues(PreparedStatement ps, int i) :-Set values on the given PreparedStatement.

(v) The PreparedStatementCreator interface is a callback interfaces used by the JdbcTemplate class. This interface creates a PreparedStatement given a connection, provided by the JdbcTemplate class. Implementations are responsible for providing SQL and any necessary parameters. A PreparedStatementCreator should also implement the SqlProvider interface if it is able to provide the SQL it uses for PreparedStatement creation. This allows for better contextual information in case of exceptions.
It has one method:
PreparedStatement createPreparedStatement(Connection con) throws SQLException

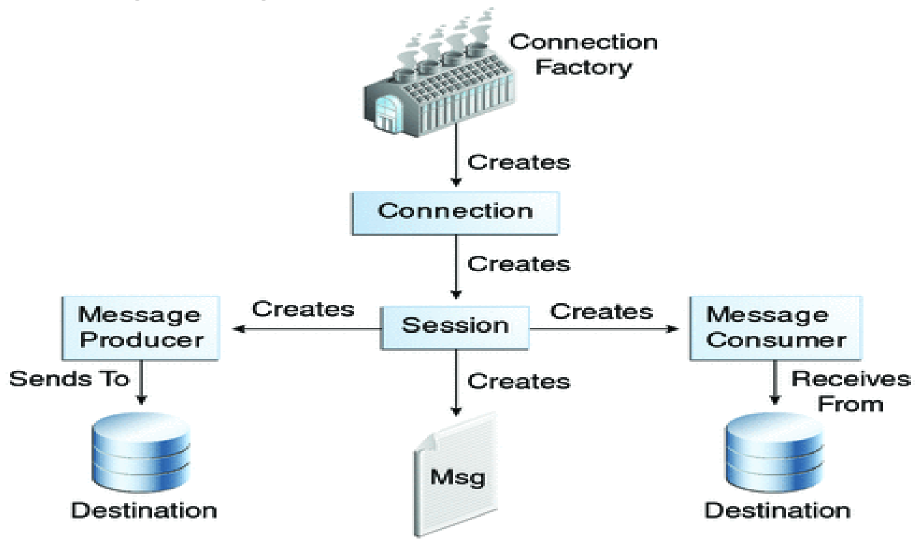
Create a statement in this connection. Allows implementations to use PreparedStatements. The JdbcTemplate will close the created statement.

Q.6 a. Explain the JMS API Programming Model.

Answer:
The basic building blocks of a JMS application are:

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

The JMS API Programming Model



JMS Administered Objects

Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

With GlassFish Server, you can use the asadmin create-jms-resource command or the Administration Console to create JMS administered objects in the form of connector resources. You can also specify the resources in a file named glassfish-resources.xml that you can bundle with an application.

JMS Connection Factories

A **connection factory** is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the ConnectionFactory, QueueConnectionFactory, orTopicConnectionFactory interface. To learn how to create connection factories, see To Create JMS Resources Using NetBeans IDE.

At the beginning of a JMS client program, you usually inject a connection factory resource into a ConnectionFactory object. For example, the following code fragment specifies a resource whose JNDI name is jms/ConnectionFactory and assigns it to a ConnectionFactory object:

```
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

JMS Destinations

A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics. A JMS application can use multiple queues or topics (or both). To learn how to create destination resources, see To Create JMS Resources Using NetBeans IDE.

To create a destination using the GlassFish Server, you create a JMS destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to one domain or the other. To create an application that allows you to use the same code for both topics and queues, you assign the destination to aDestination object.

The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace.

```
@Resource(lookup = "jms/Queue")
private static Queue queue;

@Resource(lookup = "jms/Topic")
private static Topic topic;
```

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the ConnectionFactory interface, you can inject a QueueConnectionFactory resource and use it with a Topic, and you can inject a TopicConnectionFactory resource and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

JMS Connections

A **connection** encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

```
Connection connection = connectionFactory.createConnection();
```

Before an application completes, you must close any connections you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

```
connection.close();
```

Before your application can consume messages, you must call the connection's start method; for details, see JMS Message Consumers. If you want to stop message delivery temporarily without closing the connection, you call the stop method.

JMS Sessions

A **session** is a single-threaded context for producing and consuming messages. You use sessions to create the following:

- Message producers
- Message consumers
- Messages
- Queue browsers
- Temporary queues and topics

JMS Message Producers

A **message producer** is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

You use a Session to create a MessageProducer for a destination. The following examples show that you can create a producer for a Destination object, aQueue object, or a Topic object.

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

JMS Message Consumers

A **message consumer** is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumerinterface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you could use a Session to create a MessageConsumer for a Destination object, a Queue object, or a Topic object:

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

JMS Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you define a class namedListener that implements the MessageListener interface, you can register the message listener as follows:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

JMS Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages that interest it. Message selectors assign the work of filtering messages to the JMS provider rather than to the application.

JMS Messages

The ultimate purpose of a JMS application is to produce and consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message can have three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the Message interface in the API documentation.

Message Headers

A JMS message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. Table 47-1lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors

Message Bodies

The JMS API defines five message body formats, also called message types, which allow you to send and receive data in many different forms and which provide compatibility with existing messaging formats

JMS Queue Browsers

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a QueueBrowser object that allows you to browse the messages in the queue and display the header values for each message. To create a QueueBrowser object, use theSession.createBrowser method. For example:

QueueBrowser browser = session.createBrowser(queue);

JMS Exception Handling

The root class for exceptions thrown by JMS API methods is JMSException. Catching JMSException provides a generic way of handling all exceptions related to the JMS API.

b. Write a Simple JMS Producer class which sends several messages to a queue or topic

Answer:

```
import javax.jms.*;
import javax.naming.*;


public class SimpleProducer {
   /**
    * Main method.
```

```
     *
     * @param args     the destination used by the example
     *                 and, optionally, the number of
     *                 messages to send
     */
    public static void main(String[] args) {
        final int NUM_MSGS;

        if ((args.length < 1) || (args.length > 2)) {
            System.out.println("Program takes one or two arguments: " +
                "<dest_name> [<number-of-messages>]");
            System.exit(1);
        }

        String destName = new String(args[0]);
        System.out.println("Destination name is " + destName);

        if (args.length == 2) {
            NUM_MSGS = (new Integer(args[1])).intValue();
        } else {
            NUM_MSGS = 1;
        }

        /*
         * Create a JNDI API InitialContext object if none exists
         * yet.
         */
        Context jndiContext = null;

        try {
            jndiContext = new InitialContext();
        } catch (NamingException e) {
            System.out.println("Could not create JNDI API context: " +
                e.toString());
            System.exit(1);
        }

        /*
         * Look up connection factory and destination.  If either
         * does not exist, exit.  If you look up a
         * TopicConnectionFactory or a QueueConnectionFactory,
         * program behavior is the same.
         */
        ConnectionFactory connectionFactory = null;
        Destination dest = null;
```

```
try {
    connectionFactory = (ConnectionFactory) jndiContext.lookup(
            "jms/ConnectionFactory");
    dest = (Destination) jndiContext.lookup(destName);
} catch (Exception e) {
    System.out.println("JNDI API lookup failed: " + e.toString());
    e.printStackTrace();
    System.exit(1);
}

/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create producer and text message.
 * Send messages, varying text slightly.
 * Send end-of-messages message.
 * Finally, close connection.
 */
Connection connection = null;
MessageProducer producer = null;

try {
    connection = connectionFactory.createConnection();

    Session session =
        connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(dest);

    TextMessage message = session.createTextMessage();

    for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + (i + 1));
        System.out.println("Sending message: " + message.getText());
        producer.send(message);
    }

    /*
     * Send a non-text control message indicating end of
     * messages.
     */
    producer.send(session.createMessage());
} catch (JMSException e) {
    System.out.println("Exception occurred: " + e.toString());
} finally {
    if (connection != null) {
```

```
              try {
                 connection.close();
              } catch (JMSException e) {
              }
           }
        }
     }
  }
```

Q.7 a. What's the difference between:
(i) Forward and send Redirect
(ii) Custom JSP tags and beans

Answer:
Custom JSP tag is a tag you defined. You define how a tag, its attributes and its body are interpreted, and then group your tags into collections called tag libraries that can be used in any number of JSP files. To custom JSP tags, you need to define three separate components: the tag handler class that defines the tag's behavior, the tag library descriptor file that maps the XML element names to the tag implementations and the JSP file that uses the tag library

JavaBeans are Java utility classes you defined. Beans have a standard format for Java classes. You use tags

Custom tags and beans accomplish the same goals — encapsulating complex behavior into simple and accessible forms. There are several differences:

Custom tags can manipulate JSP content; beans cannot. Complex operations can be reduced to a significantly simpler form with custom tags than with beans. Custom tags require quite a bit more work to set up than do beans. Custom tags usually define relatively self-contained behavior, whereas beans are often defined in one servlet and used in a different servlet or JSP page. Custom tags are available only in JSP 1.1 and later, but beans can be used in all JSP 1.x versions.

forward is server side redirect and sendRedirect is client side redirect. When you invoke a forward request, the request is sent to another resource on the server, without the client being informed that a different resource is going to process the request. This process occurs completely within the web container and then returns to the calling method. When a sendRedirect method is invoked, it causes the web container to return to the browser indicating that a new URL should be requested. Because the browser issues a completely new request any object that are stored as request attributes before the redirect occurs will be lost. This extra round trip a redirect is slower than forward. Client can disable sendRedirect

b. How can you prevent the output of JSP or Servlet pages from being cached by the browser? Create a custom tag hello in JSP.

Answer:
By setting appropriate HTTP header attributes we can prevent caching by the browser

```
<%
response.setHeader("Cache-Control","no-store");//HTTP 1.1
response.setHeader("Pragma","no-cache");//HTTP 1.0
response.setDateHeader("Expires", 0);//prevents caching at the proxy server
%>
```

Create "Hello" Tag:

Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion without a body:

```
<ex:Hello/>
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. So let us create HelloTag class as follows:

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

publicclassHelloTagextendsSimpleTagSupport{

publicvoid doTag()throwsJspException,IOException{
JspWriterout= getJspContext().getOut();
out.println("Hello Custom Tag!");
}
}
```

Above code has simple coding where doTag() method takes the current JspContext object using getJspContext() method and uses it to send "Hello Custom Tag!" to the current JspWriter object.

Let us compile above class and copy it in a directory available in environment variable CLASSPATH. Finally create following tag library file: <Tomcat-Installation-Directory>webapps\ROOT\WEB-INF\custom.tld.

```
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>Example TLD</short-name>
<tag>
<name>Hello</name>
```

```
<tag-class>com.tutorialspoint.HelloTag</tag-class>
<body-content>empty</body-content>
</tag>
</taglib>
```

Now it's time to use above defined custom tag **Hello** in our JSP program as follows:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
<head>
<title>A sample custom tag</title>
</head>
<body>
<ex:Hello/>
</body>
</html>
```

Try to call above JSP and this should produce following result:

```
HelloCustomTag!
```

Accessing the Tag Body:

You can include a message in the body of the tag as you have seen with standard tags. Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion with a body:

```
<ex:Hello>
  This is message body
</ex:Hello>
```

Let us make following changes in above our tag code to process the body of the tag:

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

publicclassHelloTagextendsSimpleTagSupport{

StringWriter sw =newStringWriter();
publicvoid doTag()
throwsJspException,IOException
{
      getJspBody().invoke(sw);
      getJspContext().getOut().println(sw.toString());
}

}
```

In this case, the output resulting from the invocation is first captured into a StringWriter before being written to the JspWriter associated with the tag. Now accordingly we need to change TLD file as follows:

```
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>Example TLD with Body</short-name>
<tag>
<name>Hello</name>
<tag-class>com.tutorialspoint.HelloTag</tag-class>
<body-content>scriptless</body-content>
</tag>
</taglib>
```

Now let us call above tag with proper body as follows:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
<head>
<title>A sample custom tag</title>
</head>
<body>
<ex:Hello>
        This is message body
</ex:Hello>
</body>
</html>
```

This will produce following result:

```
This is message body
```

Custom Tag Attributes:

You can use various attributes along with your custom tags. To accept an attribute value, a custom tag class needs to implement setter methods, identical to JavaBean setter methods as shown below:

```
package com.tutorialspoint;
```

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

publicclassHelloTagextendsSimpleTagSupport{

privateString message;

publicvoid setMessage(String msg){
this.message = msg;
}

StringWriter sw =newStringWriter();

publicvoid doTag()
throwsJspException,IOException
{
if(message !=null){
/* Use message from attribute */
JspWriterout= getJspContext().getOut();
out.println( message );
}
else{
/* use message from the body */
        getJspBody().invoke(sw);
        getJspContext().getOut().println(sw.toString());
}
}

}
```

The attribute's name is "message", so the setter method is setMessage(). Now let us add this attribute in TLD file using <attribute> element as follows:

```
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>Example TLD with Body</short-name>
<tag>
<name>Hello</name>
<tag-class>com.tutorialspoint.HelloTag</tag-class>
<body-content>scriptless</body-content>
<attribute>
<name>message</name>
</attribute>
</tag>
```

```
</taglib>
```

Now let us try following JSP with message attribute as follows:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
<head>
<title>A sample custom tag</title>
</head>
<body>
<ex:Hellomessage="This is custom tag"/>
</body>
</html>
```

This will produce following result:

```
This is custom tag
```

Text Books

1. Java Network Programming, 2nd Edition by Merlin Hughes Visit Amazon's Merlin Hughes Page search results Learn about Author Central (Author), Michael Shoffner (Author), Derek Hamner (Author)

2. Professional AJAX by Nicholas Zakas et alia, Wrox Press