Learning is Life .. !!

Live it .. !!

JAVA/ J2EE

[VERSION-2]

Tutorial classes for B.E (CSE/ISE)

All semesters

---

www.aklcindia.com | office@aklcindia.com

9742013378 | 9742024066

Vijayanagar | Jayanagar

Ashok Kumar K | 9742024066 | celestialcluster@gmail.com

Ashok Kumar K | 9742024066 | celestialcluster@gmail.com

UNIT 7: JSP/ RMI

Syllabus

Java Server Pages (JSP)

JSP
JSP Tags
Request String
User Sessions
Cookies
Session Objects

Remote Method Invocation (RMI)

RMI Concept
Server side
Client side

- **6 Hours**

ASHOK KUMAR K

9742024066 | celestialcluster@gmail.com

ASHOK KUMAR LEARNING CENTER (AKLC – INDIA)

9742013378 | www.aklcindia.com | office@aklcindia.com

Learning is LIFE .. Live IT .. !!

JAVA SERVER PAGES (JSP)

Definition:

Java Server Pages (JSP) is a server side program that is similar in design and functionality to a java Servlet.

A JSP is called by client to provide a web service, the nature of which depends on the J2EE Application.

Difference between JSP and a Servlet:

JSP differs from a Servlet in the way in which the JSP is written.

Servlet is written using Java programming language and responses are encoded as an output string object that is passed to the println() method. The output string object is formatted in HTML, XML, or whatever format are required by the client	**JSP** is written in HTML, XML, or in the client's format that is interspersed with the scripting elements, directives, and actions comprised of Java programming language and JSP syntax

JSP ARCHITECTURE

JSP is written in HTML rather than with the Java programming language.

JSP is converted to a Java Servlet the first time that a client requests the JSP.

There are three methods that are automatically called when a JSP is requested and when the JSP terminates normally. These are

1. jspInt()
2. jspDestroy()
3. service()

jspInt() method is identical to init() method of a Servlet and an Applet. It is called when the JSP is requested and is used to initialize objects and variables that are used throughout the life of JSP.
jspDestroy() is identical to destroy() method in a Servlet. It is automatically called when the JSP terminates normally. It is used to clean up the resources used by a JSP.
jspService() method is automatically called and retrieves a connection to HTTP.

JSP TAGS

JSP tags define java code that is to be executed before the output of a JSP program is sent to the browser.

There are five types of JSP tags:

1. Comment Tag
2. Declaration statement Tag
3. Directive Tag
4. Expression Tag
5. Scriptlet Tag

Comment Tag:

A comment tag opens with <%-- and closes with --%>, and is followed by a comment that usually describes the functionality of statements that follow the comment tag.

Ex:

```
<%--
        This is a JSP Comment
--%>
```

Declaration Statement Tag:

A Declaration statement tag opens with <%! and is followed by a Java declaration statements that define variables, objects, and methods.

Ex:

```
<%!
        int a; String s;
        Employee e = new Employee();
%>
```

Directive Tag:

A Directive tag opens with <%@ and closes with %>. There are three commonly used directives.

- Import
  Used to import java packages into JSP program
  Ex: <%@ page import = "java.sql.*" %>

- Include
  It inserts a specified file into the JSP program replacing the include tag
  Ex: <%@ include file = ”Keogh\book.html” %>
- Taglib
  It specifies a file that contains a tag library
  Ex: <%@ taglib uri = “myTags.tld” %>

Expression Tag:

An expression tag opens with <%= and is used for an expression statement whose result replaces the expression tag. It closes with %>

Ex:
<%! int a = 5, b = 10; %>
<%= a+b %>

Scriptlet Tag:

A scriptlet tag opens with <% and contains commonly used java control statements and loops. It closes with %>

JSP EXAMPLES

1. JSP to illustrate variables declarations and usage

```
<html>
	<body>
		<%!
			int a = 10;
			String s = "TEST";
		%>
		<p>
			Int value is <%= a %>
			String value is <%= s %>
		</p>
	</body>
</html>
```

2. JSP that defines a method and illustrates its usage.

```
<html>
	<body>
		<%!
			int findSum(int a, int b) {
				return a+b;
			}
		%>
		<p>
			sum of 2 and 5 is <%=findSum(2,5) %>
		</p>
	</body>
</html>
```

3. JSP to illustrate control statements

```
<html>
	<body>
		<%!
			int grade = 70;
		%>

		<% if (grade >= 70 ) {%>
			FCD
		<% } else if (grade<70 && grade>=60) { %>
			FC
		<% } else if (grade<60 && grade>=50) { %>
			SC
		<% } else if (grade<50 && grade>=35) { %>
			Just pass
		<% } else { %>
			FAIL
		<% } %>
	</body>
</html>
```

4. JSP to illustrate looping statements

```
<html>
	<body>

		<% for (int i=0;i<10;i++) {  %>
			HELLO WORLD
		<% } %>
	</body>
</html>
```

REQUEST STRING

The browser generates a user **request string** whenever the submit button is selected.

The user request string consists of:

1. URL
2. Query String

Ex:
http://www.gmail.com/Service/login.jsp?username=jimkeogh&password=jim1234
|---------- URL ----------| |---------- Query String ----------|

Query String:

JSP program should parse the query string to extract the values of fields that are to be processed by a JSP.

A JSP can parse the query string in two ways:

a) **Using request.getParameter()**
getParameter() method requires an argument, which is the name of the field whose value you want to retrieve.

Ex:
```
<%!
    String uname = request.getParameter("username");
    String pword = request.getParameter("password");
%>
```

b) **Using request.getParameterValues()**
Multi valued fields (such as checkboxes and select menus) can be read using getParameterValues() method. It returns multiple values from the field specified as an argument to this method.

Ex:
```
<%!
    String[] games = request.getParameterValues("favgames");
%>
```

URL:

A URL is divided into four parts:

- **Protocol:**
  It defines the rules that are used to transfer the request string from the browser to JSP program. Three commonly used protocols are HTTP, HTTPS, and FTP

- **Host:**
  It is the internet protocol address (IP) or name of the server that contains the JSP program.
- **Port:**
  It is the port that the host monitors. If the port is not specified, it is defaulted to 80.
  Whenever HTTP is used, the host will be monitoring the port 80
- **Virtual path to JSP program:**
  The server maps this virtual path to the physical path

Example for URL:

https://www.gmail.com:8080/service/login.jsp

Note:
There are four predefined implicit objects that are in every JSP program.
They are request, response, session, and out.

request	It is an instance of HttpServletRequest and represents the request object
response	It is an instance of HttpServletResponse and represents the response object
session	It is an instance of HttpSession.
out	It is an instance of JspWriter that is used to send a response to the client.

USER SESSIONS

A JSP program must be able to track a session as a client moves between HTML pages and JSP programs.

There are three commonly used methods to track a session.

1. Using a hidden field
2. Using a cookie
3. Using a JavaBean

Tracking the user session using a hidden field:

A **hidden field** is a field in HTML form whose value isn't displayed on the HTML page. You can assign a value to the hidden field in a JSP program before the program sends the HTML page to the browser.

Consider a login screen which asks for username and password. Here is how the session is tracked using hidden field:

- Upon submitting the form, the browser sends the username and password to the JSP program.
- The JSP program then validates the username and password and generates another dynamic HTML page. This newly generated HTML page has a form that contains hidden field which is assigned a **userID** along with other fields as well.
- When the user submits the form in the new HTML page, the **userID** stored in the hidden field and other information on the form are sent to the JSP program.
- This cycle continues where the JSP program processing the request string receives the **userID** as a parameter and then passes the **userID** to the next dynamically built HTML page as a hidden field.
  In this way, each HTML page and subsequent JSP program has access to **userID** and therefore can track the session.

COOKIES

A **cookie** is a small piece of information created by a JSP program that is stored in the client's hard disk by the browser.

Cookies are used to store various kind of information such as username, password, and user preferences, etc.

JSP program to create a cookie

```
<html>
	<body>
		<%!
			String cName = "userID";
			String cValue = "1VK06IS009";
			Cookie myCookie = new Cookie (cName, cValue);
		%>
		<%
			response.addCookie(myCookie);
		%>
	</body>
</html>
```

JSP program to read a cookie

```
<html>
	<body>
		<%!
			String cName = "userID";
			String name;
			String value;
			int found = 0;
		%>
		<%
			Cookie[] myCookies = request.getCookies();
			for (int i=0;i<myCookies.length;i++) {
				name = myCookies[i].getName();
				if (name.equals(cName)) {
					value = myCookies[i].getValue();
					found = 1;
				}
			}
			if (found==1) { %>
				<p> Cookie Name: <%= cName %> </p>
				<p> Cookie Value: <%= value %> </p>
		<%	} else { %>
				<p> Cookie NOT FOUND </p>
		<%} %>
	</body>
</html>
```

SESSION OBJECTS

A JSP database system is able to share information among JSP programs within a session by using a session object.

Each time a session is created, a unique ID is assigned to the session and stored as a cookie.

The unique ID enables JSP programs to track multiple sessions simultaneously while maintaining data integrity of each session.

In addition to session ID, a session object is also used to store other types of information called **attributes.**

JSP Program to create a session attribute

```
<html>
	<body>
		<%!
			String attName = "userID";
			String attValue = "1VK06IS009";
		%>
		<%
			session.setAttribute(attName, attValue);
		%>
	</body>
</html>
```

JSP Program to read session attributes

```
<html>
	<body>
		<%!
			Enumeration<String> e;
		%>
		<%
			e = session.getAttributeNames();
			while (e.hasMoreElements()) {
				String name = (String) e.nextElement();
				String value = (String) session.getAttribute(name);
		%>
				<p> Attribute name : <%= name %></p>
				<p> Attribute value : <%= value %></p>
		<%
			}
		%>
	</body>
</html>
```

REMOTE METHOD INVOCATION (RMI)

Definition and concepts:

A java object runs within a JVM. Likewise, a J2EE application runs within JVM; however, objects used by a J2EE application do not need to run on the same JVM as the J2EE application. This is because a J2EE application and its components can invoke objects located on different JVM by using **Java RMI system**.

RMI is used for remote communication between Java applications and components, both of which must be written in Java Programming language.

RMI is used to connect together a client and a server.

- A **client** is an application or component that requires the services of an object to fulfill a request.
- A **server** creates an object and makes the object available to the clients

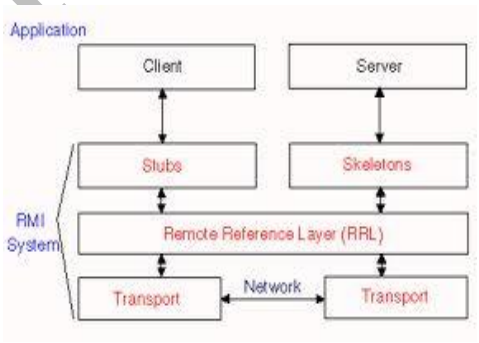
---

Note 1:

RMI handles transmission of requests and provides the facility to load the object's bytecode, which is referred to as **dynamic code loading**

Note 2:

Following fig visualizes the RMI process



When client references a remote object, the RMI passes a remote stub for the remote object. The remote stub is the local proxy for the remote object. The client calls the method on the local stub whenever the client wants to invoke one of the remote object's method.

THE RMI PROCESS

There are three steps necessary to make an object available to remote clients

1. Design an object
2. Compile the object
3. Make the object accessible to remote clients over the network

---

Designing an object

Besides defining the business logic of an object, the developer must define a remote interface for the object, which identifies methods that are available to remote clients.

In addition to methods that can be invoked by remote clients, the developer must also define other methods that support the processing of client invoked methods. There are referred as **server methods,** while methods invoked by a client are called **client methods.**

Compile the object

Compilation of the object is a two step process

- **Compile the object using the javac compiler**
  This will create the byte codes for the objects.
- **Compile the object using rmic compiler**
  This will create a stub for the object

Make the object available to remote client

It is done by loading the object into a server. Make sure the RMI remote object registry is running. If not, type the following at the command line:

```
C:\> start rmiregistry
```

SERVER SIDE

Server side of RMI consists of a remote interfaces and methods.

Remote interfaces provide the API for clients to access the methods.
Methods provide the business logic that fulfills a client's request whenever the client remotely invokes them.

Example for Remote interface is shown below:

HelloInterface.java

```
import java.rmi.*;
public interface HelloInterface extends Remote {
  public String say() throws RemoteException;
}
```

Remote Object that implements the remote interface is shown below:

Hello.java

```
import java.rmi.*;
import java.rmi.server.*;

public class Hello extends UnicastRemoteObject
 implements HelloInterface {

  public String say() throws RemoteException {

       System.out.println("Hello World");

  }

}
```

Below program in the server side makes the Remote object available to the clients by binding it to the naming registry.

HelloServer.java

```
import java.rmi.Naming;
public class HelloServer
{
  public static void main (String[] argv)   {
       if (System.getSecurityManager() == null)

              System.setSecurityManager(new RMISecurityManager());
       try {
         Naming.rebind("MyObj", new Hello ());
         System.out.println ("Server is connected");
```

```
        }
        catch (Exception e) {
           System.out.println ("Server not connected: " + e);
        }
   }
}
```

Clients can access the remote object by looking up the naming registry for the appropriate entry.

CLIENT SIDE

Below example shows how a client can lookup the naming registry to get an instance of the remote object.

It also illustrates how a client can use the remote object to call its methods.

HelloClient.java

```
import java.rmi.Naming;
public class HelloClient
{
  public static void main (String[] argv) {
        if (System.getSecurityManager() == null)

                  System.setSecurityManager(new RMISecurityManager());
        try {
              HelloInterface hello = (HelloInterface)
                                        Naming.lookup ("//192.168.10.201/MyObj");
              System.out.println (hello.say());
         }
         catch (Exceptione){
                    System.out.println ("HelloClient exception: " + e);
         }
   }
}
```

Thank you

ASHOK KUMAR K

9742024066 | celestialcluster@gmail.com

NOTE:

Advantages of JSP over servlets

- ✓ Servlets use println statements for printing an HTML document which is usually very difficult to use. JSP has no such tedius task to maintain.
- ✓ JSP needs no compilation, CLASSPATH setting and packaging.
- ✓ In a JSP page visual content and logic are seperated, which is not possible in a servlet.
- ✓ There is automatic deployment of a JSP, recompilation is done automatically when changes are made to JSP pages.
- ✓ Usually with JSP, Java Beans and custom tags web application is simplified
- ✓ Reduces Development time

- **Tutorial classes for all the semesters (CSE/ISE) by Mr. Ashok Kumar K**
- **'allRounder' – A complete placement Training course, powered by AKLC.**
- **Final year projects for B.E (CSE/ISE). All the projects in AKLC are been developed from the scratch using latest technologies. NO repeated projects.**
- **'JMASTER' – A complete training program on Java and Web Technologies, powered by AKLC**

For any Enquire contact

www.aklcindia.com | office@aklcindia.com

9742013378 | 9742024066

Vijayanagar | Jayanagar

Ashok Kumar K | 9742024066 | celestialcluster@gmail.com

AKLC
ASHOK KUMAR LEARNING CENTER

9742013378 | Vijayanagar | Jayanagar
www.aklcindia.com | office@aklcindia.com