SCWCD

BY

DUGRA

MONOJ ENTERPRISES XEROX

Flot No:40Gayatri Nagar Behind HUDA Ameerpet,Hyd.Cell:9908909131

Software institutes material available

SCWCD

SCWCD

- Servlets (5 units)
- JSP (5 units)
- Design Patterns (1 unit)
  - 6 Design patterns

Servlets

1. Servlet Technology model
   - HTTP methods
   - servlet Life cycle
   - servlet Request
     - Form parameters
     - Request Header
     - cookies
   - Servlet Response
     - Set Content Type
     - set headers
     - add cookies
     - getWriter => For sending Text Data
     - getOutputStream => For Binary Response
     - Send Redirection

2. Structure of web Applications & Deployment Description
   - Directory structure of web Application
   - * web.xml
   - war Files

3) The web Container model

- Servlet Context Parameters
- Servlet Scopes & Attributes
- Request Dispatcher
- Filters
- wrappers

4) Session Management

- Session API
- cookies
- URL Rewriting
- Listeners

5) web Security

- Terminology
  - Authentication
  - Authorization
  - Data Integrity
  - Confidentiality
- Types of Authentication
  - Basic Authentication
  - Digest "
  - Form Based "
  - client cert "
- web.xml declarations
- Programmatic Security.

JSP's

1. JSP Technology model

- JSP Life cycle
- JSP API
- Scripting Elements
- JSP Directives
- JSP Implicit Objects (9)
- JSP Documents
- JSP scopes.

2. Developing JSP's by Standard Actions

- JSP : useBean
- JSP : setProperty
- JSP : getProperty
- JSP : Include
- JSP : Forward
- JSP : param

Not used

- JSP : fallback
- JSP : Plugin
- JSP : Params

3. Building JSP's by Expression Language (EL)

- EL implicit Objects (11)
- EL Operators
- EL Functions

4. JSTL

- Core Library

5. custom Tags

- classic Tags
- simple Tags
- Tag Files

1) Introduction to Design patterns

- Front Controller
- MVC
- Intercepting Filter Design pattern
- Data Transfer Object
- Business Delegate
- Service Locator.

SCWCD 1.5V

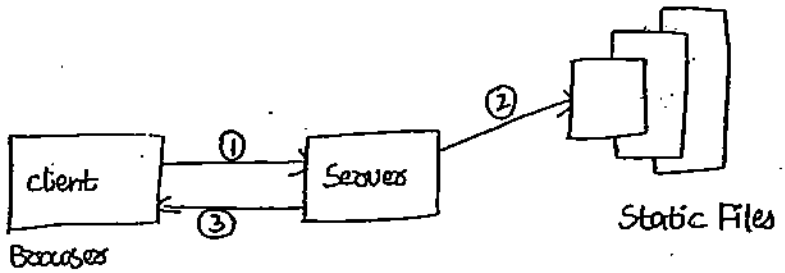
J2EE 1.5 V

- Servlet 2.5 V
- JSP's 2.1 V
- JSTL 1.2 V

UNIT – I

Servlet Technology model

web programming For Static Information –



① Client sends a request for a static Files

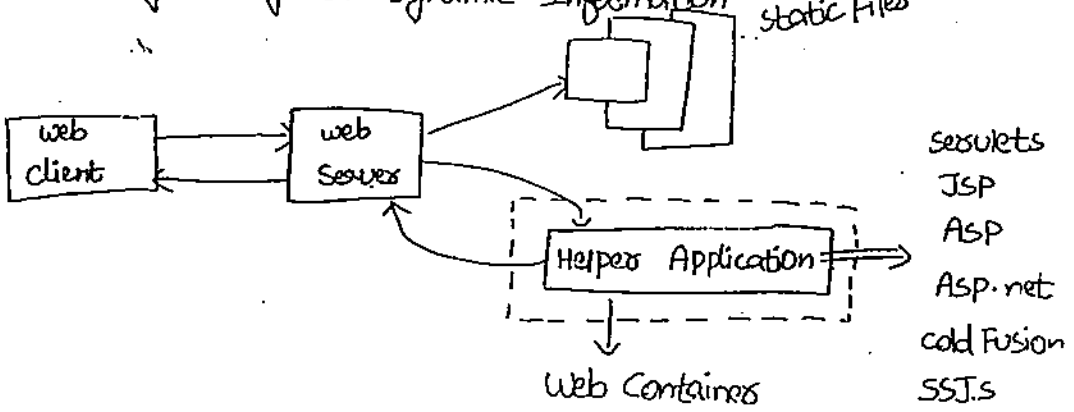
② Server searches whether the request resource is there or not

③ If the request resource is available then it will return that resource as response.

→ If it is not available, server sends 404: Status code saying the requested resource is not available.

Note:- For servicing static Files, no processing is required at server side. Hence web server always loves to serve static Information.

Web Programming for Dynamic Information



→ Browser sends a request to the web Server

→ web Server checks whether the request is for Static or for Dynamic Information.

→ If it is for Static, web Server Searchs for that Static Files in the Local File System.

— If it is available, it returns that Static File otherwise it returns 404 Status code.

→ If the request is for Dynamic Information, the web Server forwards that request to Helper Application.

→ Helper Application analyses & Process the request and generates the required Dynamic Information.

→ Helper Application forwards that Dynamic Information to the web Server & web Server intrns forwards that Dynamic Info to the browser.

→ The following are various Helper Applications

- Servlet
- JSP
- ASP
- ASPnet
- Cold Fusion
- Server side Java Script (SSJS)
- PHP
- CGI

Servlet

Servlet is a Server side web Component managed by web Container for generation of Dynamic Information.

⇒ web Container is the best Assitent for the Programmer. It maintains entire life cycle of the Servlet, So that the Programmer has to concentrate only on servicing Logic.

Types of web Containers

There are 3 types of web Containers are possible

1. Stand - Alone web Containers
2. InProcess "
3. Out Process "

Stand - Alone — Both web Server & web Container are available in a Single Integrated Component.

Ex: Tomcat.

InProcess — If the web Container runs in the Same address Space (Same machine) of web Server & it is available as Plug-ins, Such type of web Containers are InProcess web Contain

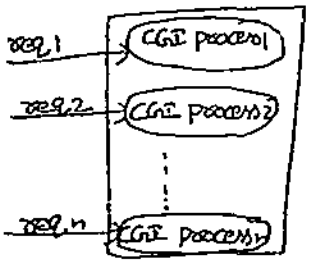
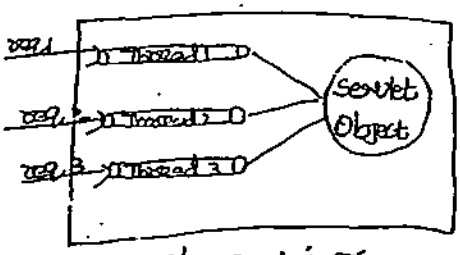
→ In this case both web Server & web Container need not be from the Same Vendor.

Out-process — web Server & web Container both are Running on different machine. web Container is attached to web Server externally.

→ we can configure front End Apache Server to forward the request to the back End web Logic Server.

→ This type of web Containers are most frequently used web Containers.

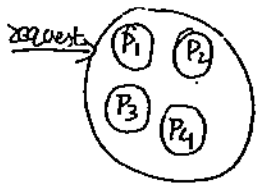
Difference between CGI & Servlets

CGI	Servlet
1) It is process based i.e for every request a seperate process will be created & it is responsible to generate Response.	1) It is Thread based. ie for every request a seperate new Thread will be created and it is responsible to generate Dynamic info
 web Container	 web Container
2) Creation & Destruction of new Process for every request is costly. → If the number of request increases it will effect the Performance of the System. Hence CGI Technology fails to deliver Scalable Applications.	2) Creation & Destruction of new Thread for every Request is not costly. → There is no effect on the Performance even though number of requests increases. Hence Servlet Technology succeds to deliver Scalable Applications.

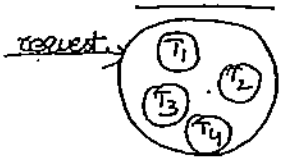
Two processes never share common Address space. Hence there is no chance of concurrency problems.	(3) All the Threads shares the common Address space. Hence there may be a chance of concurrency problems.
CGI program can be written in multiple languages. But most commonly used language is Perl.	(4) Servlets can be written only in Java.
most of the CGI languages are not object oriented. Hence we are missing the benifits of oops	(5) Java language itself is object oriented. Hence we can get all benifits of oops.
CGI Tech is Platform Dependent	(6) Servlet Tech is platform Independent

Fast CGI

It improves the performance when compared with CGI. In this case web containers maintains a pool of processes so that a single process can serve multiple requests one by one.



Servlet 3.0



Servlet API

- we can implement Servlet Programming by using the following 2 packages:

① javax.servlet (for generating Generic Servlet)

② javax.servlet.http

javax.servlet — This package contains several classes & Interfaces used for Developing Servlets irrespective of any protocol.

javax.servlet.http — This Package contains several classes & Interfaces for Developing http based Servlets.

javax.servlet Package Interface

① Servlet
② Servlet Request
③ Servlet Response
④ Servlet Config
⑤ Servlet Context
⑥ Request Dispatcher
⑦ Single Thread model
⑧ Filter
⑨ Filter Config
⑩ Filter chain

} For implementing Filter Concept

⑪ Servlet Request Listener
⑫ Servlet Request Attribute Listener
⑬ Servlet Context Listener
⑭ Servlet Context Attribute Listener

} For Listeners Concept

1. Servlet – Every servlet in Java should implement this interface either Directly or indirectly.

→ This interface Defines the most common general methods which can be applied on any servlet.

→ servlet interface only Contains Life cycle methods.

2. Servlet Request – Servlet Request Object contains the client information. It defines several methods to get client Info.

Ex: getParameter("uname");
getCookies(" ");

3. Servlet Response.

→ By using Response Object servlet can send required response to the End user (Browser).

→ This interface Defines several methods for Preparation of response

Ex: getWriter, sendRedirect

4. Servlet Config – For every servlet web Container creates servlet Config object to hold its Configuration Information. By using Config object servlet can get its Configuration Information like Initialization Parameters, Logical name of servlet

5. Servlet Context – For every web Application, web Container creates one servlet Context object to hold application level Configuration Information.

Note- Servlet Config per servlet where as Servlet Context is per web Application.

6) Request Dispatcher By using Request Dispatcher Object we can dispatch the request from one web Component to another.

=> This interface Contains only 2 methods

① Forward ② Include

3/69

7) Single Thread model

→ This interface is for Providing Thread Safety for the Servlets. If a servlet implement this interface, at a time the servlet object can Process only one client Request.

→ The advantage of this approach is, we can resolve Concurrency Problems.

→ The main limitation of this approach is, it may rise Performance Problems. Due to this reason sun People Deprecated this interface with out any replacement in Servlet 2.4 V.

→ This Interface Doesn't Contain any methods. Hence it is a marker Interface.

→ It is recommended to use Synchronisation blocks & methods to Provide Thread Safety instead of Single Thread Model.

Javax.Servlet Package Classes

Generic Servlet - This is for Developing servlets irrespective of any Protocol. This class implements Servlet Interface

ServletInputStream — This can be used for Reading Binary Data send by the client

ServletOutputStream — For sending Binary Response (PDF, image files) to the client.

ServletRequestWrapper
ServletResponseWrapper
} For Implementing wrapper classes (for converting PDF to doc file)

ServletRequestEvent
ServletRequestAttributeEvent
ServletContextEvent
ServletContextAttributeEvent
} For Defining Events.

Javax.servlet Package Exceptions

This package defines the following 2 Exceptions :

1. ServletException — Servlet can throw this Exception whenever it facing any problem while processing the request.

2. UnavailableException - It is child class of ServletException & it is Deprecated.

Exception
↑
Servlet Exception
↑
UnavailableException

Note In total javax.servlet package contains

14 – Interfaces
9 – classes
2 – Exceptions

Servlet Interface

→ This interface defines the following 5 methods.

1. init
2. service
3. Destroy
4. getServletInfo
5. getServletConfig

1. init :-

Public void init(ServletConfig config) throws ServletException

→ This method called only once by the web container to perform initialization activities immediately after Instantiation.

→ Before servicing any client Request, compulsory init() should be completed.

→ The web Container won't place servlet in to service in the following cases.

1. If init() throws ServletException
2. init() doesn't return with in the Time period specified by web Container.

→ In the above cases, web Container makes that Servlet Object Eligible for Garbage Collection with out calling any other Life Cycle method.

```
init()
{
  =
  ServletException
  =
}
```

```
init()
{
  =
}
```

web Container intern creates another init() and continues the process until init() completes.

2) Service

For every Request, web Container executes this method to provide Response.

```
public void Service (ServletRequest req, ServletResponse res)
                throws ServletException, IOException.
```

→ Entire Servicing logic we have to define in this method only.

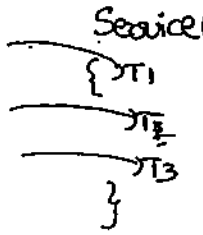
3) Destroy

```
public void destroy()
```

→ This method called only once by the web Container to perform clean-up Activities. whenever web Container takes servlet

from out of Service. This is usually happens at the time of Application unDeployment (or) Server ShutDown (or) whenever web Container requires Some Free Memory.

→ whenever web Container calls destroy() it may not execute immediately. It will wait untill the completion of all currently Executing Threads.



→ suddently when web Container calls destroy() it waits untill all Threads got released.

⇒ init, Service & Destroy also considered as Life Cycle methods of Servlet.

4) getServletConfig()

→ public ServletConfig getServletConfig()

- Returns Servlet Config of the Servlet Object.

→ Servlet can get its configuration information by using ~~this~~ its Config Object.

5) getServletInfo()

public String getServletInfo()

→ This method returns information about Servlet like author, version, copy write information etc.

→ init, service(), destroy() are call back methods. web Container internally calls these methods.

→ Externally we have to call getServletConfig(), getServletInfo().

→ Program for writing Servlet that implements Servlet Interface Directly.

```
import javax.servlet.*;
import java.io.*;
Public class FirstServlet implements Servlet, SingleThreadModel
{
    static
    {
        S.o.p("servlet Loading");
    }
    Public FirstServlet()
    {
        S.o.p("Servlet Instantiation");
    }
    Public void init(ServletConfig config) throws ServletException
    {
        S.o.p("Init method");
    }
    Public void service(SR req, SR resp) throws SE, IoE
    {
        PrintWriter out = resp.getWriter();
        out.Println("<h1> welcome to SCWCD </h1>");
    }   S.o.p("Service method");
```

```
public void destroy()
{
    S.O.P ("Destroy method");
}
public ServletConfig getServletConfig()
{
    return null;
}
public String getServletInfo()
{
    return null;  (or) return "By Durga";
}
}

<web-app>
  <servlet>
    <servlet-name> FirstServlet <servlet-name>
    <servlet-class> FirstServlet </servlet-class>
    <load-on-startup> 10 </load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name> FirstServlet </servlet-name>
    <url-pattern> /hello </url-pattern>
    <url-pattern> /first </url-pattern>
```

</servlet mapping>

</web-app>

without Load-on-Startup	with Load-on-Startup
1st Request	⇒ At the time of Server Startup
Servlet loading	Servlet Loading
Instantiation	Instantiation
init()	init()
service ()	
2nd Request	1st Request
service ()	service ()
	2nd Request
	service ()

Note

from Servlet 2.5 V onwards a single Servlet can map with multiple Url-patterns.

Life Cycle of Servlet that implements Servlet Interface

Servlet class loading by class Loader

Servlet Instantiation by web Container, for this web Container allways calls public no-Argument Constructor. Hence every Servlet class should compulsory Contain public no-argument constructor otherwise Instantiation Problems will rise.

) Execution of init() by web Container.

Note
The above 3 steps will be executed generally at the time of First Request.

⇒ If load-on-Startup is configured these will be executed at the time of Server Start-up (or) at the time of Application Deployment.

) Execution of service() by web Container

) Execution of destroy() by web container

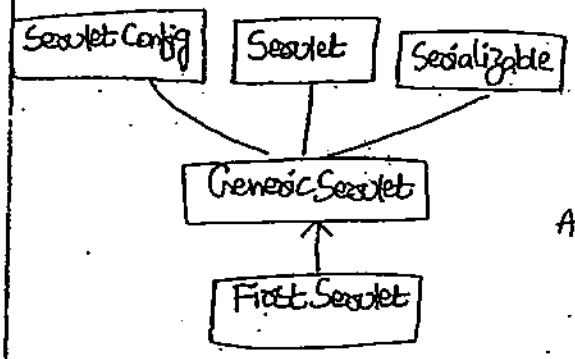
Generic Servlet

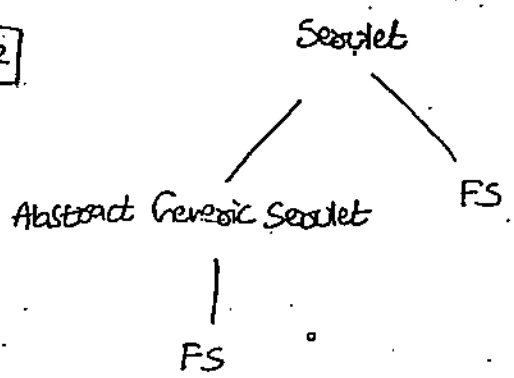
> we can Develop Servlet by implementing Servlet Interface directly. Here we should provide implementation for all 5 methods of Servlet Interface eventhough most of the times our concentration is on service() only.

> This makes code ugly & unnecessarily complicated. we can resolve this problem by using Generic Servlet.

> Generic Servlet already implemented Servlet Interface & provide implementation for all methods except service().

> we can make our Servlet as child class of Generic Servlet and provide implementation only for service().





→ Generic Servlet is an Abstract class and implements ServletConfig, Servlet & Serializable Interfaces.

Demo program for writing Servlet that extends Generic Servlet

```
Public class FirstServlet extends GenericServlet.
{
    Public void service (SR req, SR resp) throws SE, IOE
    {
        PrintWriter out = resp.getWriter();
        out.Println("writing Servlet by Extending GenericServlet
                                    is very Easy");
    }
}
```

implementation of Generic Servlet

```
public class GenericServlet implements Servlet, ServletConfig,
                                       Serializable.
{
    Private transient ServletConfig Config;

    Public void init(ServletConfig config)
    {
        this.Config=config;                 For webContainer
                                                purpose
        init();
    }
    Public void init() throws SE
    {                                   For Programmer
                                            Purpose
    }

    Public ServletConfig getServletConfig()
    {
        return Config();
    }
```

=> Generic Servlet Contains 2 init()'s.

=> we can override init() in our Servlet as follows

```
① Public void init(ServletConfig config) throws SE
  {
        // DataBase Operations

  }
```

-> This approach is not recommended bcz we are not saving Config object Provided by the web Container for the

future purpose. In this case, getServletConfig() returns null.

(2) Public void init(ServletConfig Config) throws SE

```
{
    super.init(config);
    // DataBase operation.
}
```

->This is Perfectly valid approach but not recommended to use bcz we are increasing complexity and it may effect performance.

(3) Public void init() throws SE

```
{
    // DataBase operations.
}
```

->This approach is valid and highly recommended

(1) what is the purpose of 2 init()'s in GenericServlet.

(2) what is the necessity of Declaring Config variable as Transient in genericServlet.

javax.servlet.http package

→ This package defines several classes & interfaces which can be used for developing http based servlets.

Interfaces of javax.servlet.http

1) HttpServletRequest – This is the child interface of ServletRequest. It contains client information.

2) HttpServletResponse – It is the child interface of ServletResponse. By using response object we can send the response to the client.

3) HttpSession – It is used for session management.

⇒ In addition to the above 3 interfaces javax.servlet.http pkg contains the following interfaces also.

4) HttpSessionListener
5) HttpSessionAttributeListener
6) HttpSessionBindingListener
7) HttpSessionActivationListener

For implementing Session Based listeners

8) HttpSessionContext – Deprecated in 1.5 V and not recommend to use

Classes of javax.servlet.http Pkg.

1. HttpServlet – It is the child class of GenericServlet. It can be used for Developing Http Based Servlets.

2. Cookie – used for Session Management.

3. HttpSessionEvent
4. HttpSessionBindingEvent

} For Defining Events in Session listeners

5. HttpServletRequestWrapper
6. HttpServletResponseWrapper

} For Defining Http Based wrappers.

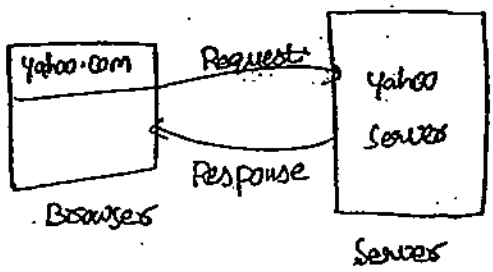
7. HttpUtils – Deprecated in Servlet 2.3 version.

=> In total javax.servlet.http Pkg defines

8 - Interfaces

7 - Classes

Common Terminology



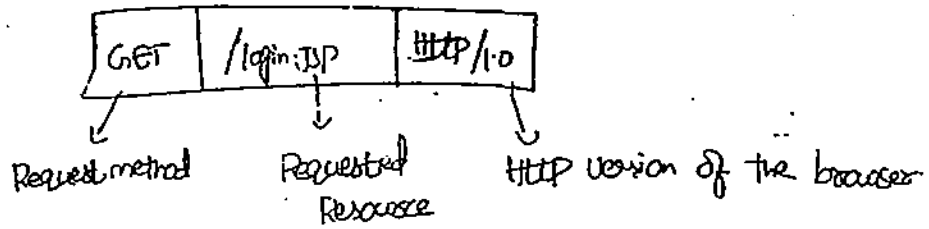
→ web client & web server communicates by using some common language known as HTTP.

→ HTTP Defines standard structures for Request & Response.

Structure of HTTP Request

Request Line
Request Header
Request Body

Request Line –



Request Headers –

These provide ~~Seskers~~ Browsers configuration information like media types accepted by browser, supporting Encoding types etc.

→ Server uses these request headers while preparing the Response.

Request Body — It contains End-users provided Information.

⇒ For the GET Request it is optional. But mandatory for the Post Request.

Structure of HTTP Response.

Status Line
Response Headers
Response Body

Status Line --

HTTP/1.1	200	OK

HTTP Version of Server → Status Code → Description of Status Code.

1xx ⇒ Information

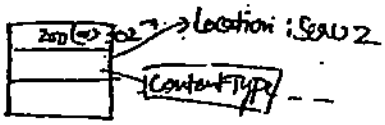
2xx ⇒ Successfull

3xx ⇒ Redirection

4xx ⇒ Sender Error

5xx ⇒ Server Error.

Response Headers — These provides Configuration information of the server & the information abt the Response. Like Content type, Content length, Last modified Date etc.



→ The browsers use these Response Headers to Display Response properly to the End users.

Response Body — It Contains the original Response provided by the Server.

Types of HTTP Request methods

→ Based on the type of information Requested by the browser all http methods are divided as follows.

① GET ② POST ③ HEAD	Introduced in HTTP 1.0 V	⑧ CONNECT ⑨ MOVE ⑩ PROFIND ⑪ Lock
④ OPTIONS ⑤ PUT ⑥ DELET ⑦ TRACE	Introduced in HTTP 1.1 V	

→ Untill Servlet 2.4 version web server can provide the support for the first 7 HTTP methods. But from Servlet 2.5 ver onwards it provides the support for the remaining methods also.

(1) GET

→ If we require information from the server then we should go for GET request.

→ usually GET requests are Read-Only.

→ In GET Request the client provided information appended to the URL as the part of Query String. & it is visible to the outside. Hence there is no security for the client Data & we can't send Sensitive information.

→ By using get Request we can send only limited number of Information. we can't send Binary Data & we should send only Text Data.

Book marking of URL's is possible for the GET Request.

Idempotent ($A^2=A$)

By making the request either 1 time or several times if we are getting the same Response then that Request is said to be Idempotent.

GET Requests are Idempotent.

⇒ If we send a req for HeadFirst Servlet book we get the same Response, irrespective of no of requests.

SAFE Request

By Repeating a Request several times. If it won't create any side Effects at server side, such type of Requests are said to be safe Request.

⇒ GET Requests are safe but post Requests are not safe.

GET

⇒ Request for Head Post Servlet

⇒ Response is same for any no of requests

Post

Durga $\xrightarrow{1\ lach}$ Chiranjevi

→ He wants to send 1 lk to Chiru Acc

→ If he clicks for 3 or 4 times bcz of Process time. At server side problem arises.

Triggers For Sending GET Request

① Typing url in the Address Bar & Submit is always GET Requ

② Clicking Hyperlink

③ Submitting the form where method Attribute is not Specified

④ Submitting the form where method Attribute is specified with getvalue.

Formation → with method / without method

or

② Post

⇒ This method is usually for sending (or) posting huge information to the Server.

Ex: uploading Resume in Job portal.

→ In Post request the client information encapsulates in the request body instead of appending to the Url.

Hence we can send sensitive Info by using Post method.

→ There is no limit on size of request body. Hence we can

send huge amount of information.

→ we can send Binary Information also in addition to text data.

→ we can't Book mark Post Url's.

→ Post Requests are not Idempotent & not safe bcz at Server side Some updations will take Place in post Req.

Trigger For Sending Post Request

→ To send Post Request Compulsory there should be form with method Attribute Value as post.

```
<form action = /web-app/Test1  method = "POST" >
  =
</form>
```

Differences btw GET & POST

HEAD

→ This method is for getting only Response Header Information like content type, Content size, Last modification

→ For Head Request internally doGET() will executes.

→ Head Requests are Idempotent & Safe.

Options

→ This method is for getting supporting methods for Retriving a particular resource from the web Server.

→ Options method is Idempotent & Safe.

5) Put

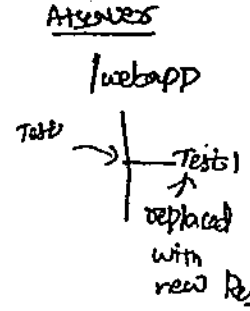
→ It is for placing a resource at the server side where the Location is specified by URL.

→ At the specified location if already another resource present then the old Resource is replaced with provided new Resource.

→ By means of Status Code. we can identify whether the Replacement is happened or not.

200 : if Replacement Happen

201 : otherwise.



→ Put Request are Idempotent but not Safe

6) DELETE

→ This method is for deleting a particular Resource at the Server side. It is exactly counterpart (opp) to PUT.

→ Delete method is Idempotent but not Safe.

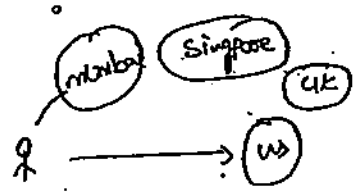
Note

Put & Delete methods are not Safe. Hence most of the web servers doesn't provide the support.

7) Trace

→ we can use these methods for Debugging purposes. If we want to know what Request Server getting as Response. then we should go for Trace method.

→ Trace method is Idempotent & Safe.



HTTP method	Idempotent	Safe
1) GET	yes	yes
2) POST	no	no
3) HEAD	yes	yes
4) OPTIONS	yes	yes
5) PUT	yes	no
6) DELETE	yes	no
7) TRACE	yes	yes

→ The only non-Idempotent method is "POST".

→ The following methods are not Safe

POST

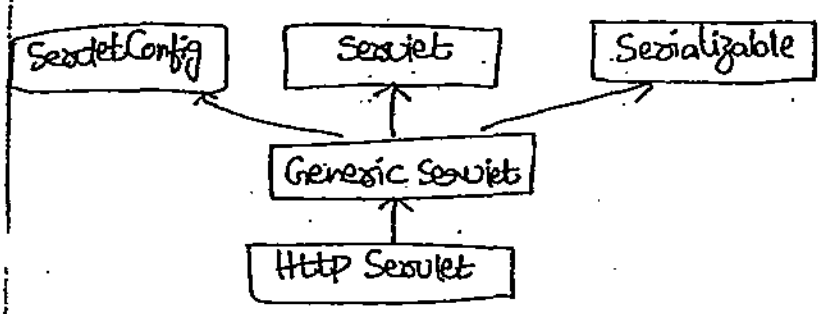
PUT

DELETE.

HTTP Servlet

→ This class can be used for Developing HTTP Based Servlets.

→ It is the child class of Generic Servlet.



→ For every HTTP method XXX, HTTP Servlet class Contains the Corresponding doXXX methods.

```
protected void doXXX(HttpServletRequest req,
                     HttpServletResponse res) throws IOE, SE
```

Ex: Protected void doGET(HSR req, HSR res) throws SE, IOE

→ HTTP Servlet class contains 2 Service methods.

(1) public void Service(SR req, SR res) throws SE, IOE

(2) protected void Service(HSR req, HSR res) throws SE, IOE.

Demo. Program for HttpServlet

form.html

```
<html>
<body> This is SCWCD class </body>
<form action = "/SCWCDIA /Test"  method="GET">
Enter Name : <input type=Text  name =Uname >
<input type="Submit">
</body>
</form>
</html>
```

```
import javax.Servlet.*;  (ServletException)
import javax.Servlet.http.*;
import javax.io.*;

Public Class FirstServlet extends HttpServlet
{
  Public void doGET( HSR req, HSR res) throws SE, IoE
  {
      String name = req.getParameter ("uname");
      PrintWriter out = req.getWriter();
      out.println("This is doGET() ");
      out.println(" Hello" + name + "doget() got executed");
  }
```

```
public void doPost(HSR req, HSR res) throws SE, IOE
{
    // same code as in doGET().

}
}
```

Life cycle of HTTP Request

→ when ever we are submitting the form, Browser prepares HTTP Request & sends to the Server.

→ web Server checks whether the request is for static or Dynamic Information.

→ If the request is for static Information web Server provides the required Response.

→ If the request is for Dynamic Information, web server forwards it to the web Container.

→ web Container Identifies the Corresponding Servlet class & checks whether the Servlet Object is already available or not

→ If the servlet Object is not already available then web Container loads the Corresponding Servlet class, perform Instantiation (creates object) and executes init()

→ web Container Creates ServletRequest & ServletResponse Objects & invokes Public Service() by passing these Objects as Arguments.

```
Public void Service (SReq req, SResp resp) throws SE, IOE
{
    HttpServReq req1 = (HttpServReq) req;
    HttpServResp resp1 = (HttpServResp) resp;
    Service (req1, resp1);
}
```

→ Public Service method Type Casts ServletRequest & Response Objects in to HttpServletRequest & HttpServlet Response. Then it invokes Protected Service().

→ The Protected Service() identifies the corresponding request method and invoke the corresponding doXXX().

→ If our Servlet Contains that method it will be executed otherwise HttpServlet doXXX() will execute & provide the response 405 status code saying Requested method is not supported by us.

→ If the Requested method is not a valid Http method then we will get 501 status code saying Requested method is not a valid Http method.

```
protected void Service (HSR req, HSR res) throws SE, IOE
{
    String method = req.getMethod();
    if(method.equals("GET"))
    {
        doGET(req, resp);
    }
    else if (method.equals("POST"))
    {
        doPOST(req, resp);
    }
    else
    {
        return SC1 status code;
    }
}
```

Case1

If our Servlet Contain Service() with SR & SResp as an Arguments and another Service() with HSReq, HSResp as an Arguments then for any Request first Service() will be executed.

If our Servlet Contain Service() & doGET() then for any kind of Request including GET only Service() will be executed.

Case 3 If we are sending GET request but our servlet don't contain doGET() then HttpServlets doGET() will be executed and returns 405 Status Code saying Http method GET is not supported by this URL.

Case-4

For providing same Response for both get() & POST() we have to implement servlet as follows.

```
Public class HelloServlet extends HttpServlet
{
    Public void doGET(HSR, HSR) throws SE, IoE
    {
        // Implementation.
    }
    Public void doPOST(      )
    {
        doGET(req, resp);
    }
}
```

Case-5

For providing Response to Head request we have to override either doHEAD() or doGET() otherwise we will get 405 : saying Http GET() is not supported by URL.

```
doHEAD()
{
    doGET();
}
```

not present, HttpServlet GET() will be executed — 405.

- In our Servlet it is not Recommended to override the following methods.

~~Servlet Request~~

public Servlet (SR, SResp)
protectedServlet (HSR, HSR)

HttpServlet Class Contains Proper Implementation.

doHEAD()
doOPTIONS()
doTRACE()

Servlet
↑
GenericServlet → Abstract
↑
Abstract ← HttpServlet — doGET(){ }
doPOST(){ }
⋮

→ GenericServlet implements Servlet interface but doesn't provide implementation for service(). Hence GenericServlet is declared as Abstract.

→ All the methods are available in HttpServlet are Concrete methods and doesn't contain Single Abstract method. But HttpServlet class Declared as Abstract bcz most of the methods of HttpServlet are JUST for sending Error Information.

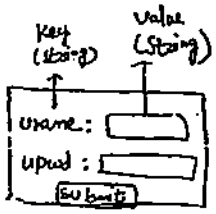
→ creating an object of HttpServlet & calling doGET() explicitly

is senseless. Bcz it will provide only Error Information as the Response. To prevent Instantiation HttpServlet is Declared as Abstract.

HttpServletRequest

Q) → using HttpServletRequest write code to

1. Retrive Form parameters
2. Retrive Header Information
3. Retrive Cookies.



Retrieving Form Parameters

→ Form Parameters are key-value pairs where both key & value are String objects only.

→ A Form Parameter can be Associated with single or multiple values. (selecting a course c, c++, java ---, we can select more than one course)

Servlet Request Interface Defines the following methods to Retrive Form parameters from the request Object.

1. String getParameter(String parametername) — Returns the parameter value associated with specified parameter.

→ If the parameter Associated with multiple values then this method returns First value.

→ If the specified parameter is not available then this method returns "null".

→ The Argument is case sensitive.

Ex: String course = req.getParameter("ucourse");

2) String[] getParameterValues(String Pname)

- Returns all the values associated with the specified parameter.
- If the specified parameter is not available this method returns "Null".
- Argument is case-sensitive.

Ex: String[] s = req.getParameterValues("ucourse");

```
for(String s1 : s)
{
    out.Println(s1);
}
```

3) Enumeration getParameterNames()

- Returns all the form parameter names associated with request.
- If the request doesn't contain any form parameter, this method returns empty Enumeration object but not null.

Ex: Enumeration e = req.getParameterNames();

```
while(e.hasMoreElements())
{
    String Pname = (String) e.nextElement();
    String PValue = req.getParameter(Pname);
    out.Println(Pname + " ---- " + PValue);
}
```

(4) Map getParametersMap();

(22)

- Returns the Map Object Containing Parameters Names as keys & Parameters Values as Map Values.
- Keys are String Objects but Values are String[]
- If the request doesn't Contain any form Parameters then this method returns Empty map Object but not null.

map m →

Keys String	Values String[]
uname	durga
upwd	anushka
ucourse	SCJP, SCWCD

```
map m1 = req.getParametersMap();
Iterator itr = m1.iterator();
while (itr.hasNext())
{
    map.Entry m1 = (Map.Entry) itr.next();
    String Pname = (String) m1.getKey();
    String[] s = (String[]) m1.getValue();

    out.Println (Pname);

    for (String[] s1 : s)
    {
        out.Println (s1);
    }
}
```

```
Map m1 = req.getParameterMap();

for(MapEntry m2 : m1)
{
    String Pname = (String) m2.getKey();
    String[] s = (String[]) m2.getValue();

      out.Println (Pname);
      for (String[] s1 : s)
      {
          out.println( s1);
      }
}
```

Demo program

```
<html>
<body>
<form action="/scwcDID/test" >
Enter Name : <input type=text  name=uname >
Enter Contact No : <input type=text  name =ucontact>
Enter Course:
<select  name=Course  multiple >
<option  value=scjP > </option >
<option  value=scwcD> </option>

</select >
<input type = submit >
</form > </body> </html>
```

Enter Name: [ ]
Enter Contact: [ ]
Course: [ ]
[Submit]

```
import javax.servlet.*, Servlet.http.*, io.*, util.*;

Public class FormDemoServlet extends HttpServlet
{
    Public void doGET (HSR req, HSR res) throws SE, IOE
    {
        PointWriter out = req.getWriter();
        out.Println ("GetParameter Demo");
        String uname = req.getParameter ("uname");
        String ucontact = req.getParameter ("ucontact");
        String ucourse = req.getParameter ("ucourse");
        out.Println("Student name:", +uname );
        out.Println ("Student Contact :", +ucontact);
        out.Println (" Student Course :", +ucourse);
        out.Println ("GetParameter Values ");
        String[] s = req.getParameterValues("course");
        for(String[] s1 : s)
        {
            o.pln(s1);
        }
        o.pln ("Get Parameter Names");
        Enumeration e = req.getParameterNames();
        while (e.hasMoreElements())
        {
            String s1 = (String) e.nextElement();
```

```
out.pln( S1)

out.pln("Get Parameters Map Demo");

  Map m = req.getParametersMap();

  Set s = m.entrySet();


  Iterator itr = s.iterator();
  while (itr.hasNext())
  {
    Map.Entry m1 = (Map.Entry) itr.next();
    String pname = (String) m1.getKey();
    String [] S1 = (String[]) m1.getValues();
      o.pln (pname + "---");
      for(String[] S2 : S1)
      {
          o.pln (S2);
      }
}
```

(2) http://www.durgasoft.com/ordersServlet.do ?uname:durga & item=1238 & item=456 & item=789.

(1) getParameter("uname"); → pavan

(2) getParameter("item"); → 1238

(3) getParameterValues("uses"); { pavan }

(4) getParameterValues("item"); {1238, 456, 789}

(5) getParameterNames() → Enumeration containing uname & item

(6) getParameterMap();

uname	{pavan}
item	{1238, 456, 789}

(7) getParameter("age") → null

(8) getParameterValues("Pointy") → null

Retriving Request Headers

→ For every Request Browsers sends its Configuration Information in the form of Headers. These may include the media types accepted by Browsers, Encoding types supported by browsers, the type of Browsers etc.

→ Servers uses these Request Headers to send proper Response to the browsers.

→ The following is some of important Request Header.

(1) Accept : media Types Accepted by Browsers (PDF, Zeus, ~~zip~~ etc..)

(2) Accept-Encoding : Encoding Types supported by Browser (ZIP, DFlate ..)

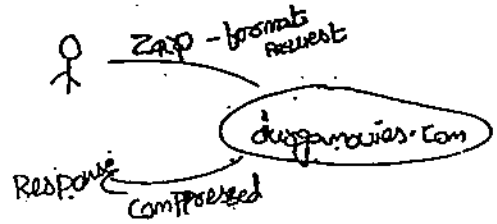
) User-Agent : It represents the type of Browser (IE, mozilla)

) Content-Length : It represents the length of request Body.

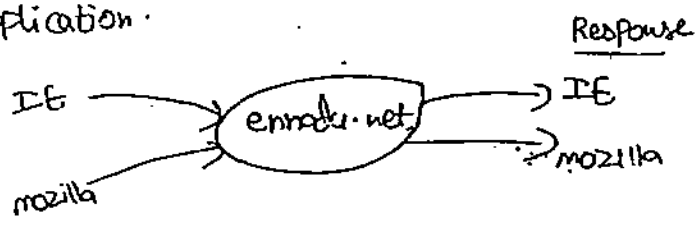
) cookie : used to send cookies for Session Management.

Utilities of Request Headers.

) By using Accept-Encoding Request Header the Server can send compressed Data instead of Original Data. It reduces Download time and improves performance



) By using User-Agent Request Header, the Server can send customized Response. we can Increase hit rate of the Application.



) By using cookie Request Header, we can send cookies to the Server so that we can achive Session Management.

) HttpServlet Request defines the following methods, to Retrive Header Information at Server side.

1. String getHeader(String name)
   - Returns the value associated with specified Header.
   - If the specified Header associated with multiple values then this method returns First value.
   - If the specified Header is not available it returns "NULL"
   - Arguments are in case-insensitive.

2. Enumeration getHeaders(String headersname)
   - Returns all the values associated with specified Header.
   - If the specified Header not available then this method returns Empty Enumeration Object but not NULL.
   - → Argument is case-insensitive.

3. Enumeration getHeadersNames()

→ HttpServletRequest contains the following more convient methods to retrive int & Date Header values.

1. String s = req.getHeader("Content-Length");
   int l = Integer.parseInt(s);

   replaced with, start
   → int l = req.getIntHeader("Content-length");

   int getIntHeader(String name);

⇒ If the specified Header not associated with int value then this method rises NumberFormatException at RunTime.

5) long getDateHeader (String name)

— Returns the Date value Associated with the Header as the no. of millisecounds Since Jan 1st 1970.

Jan 1st 1970 to 2009 4th Apr

— If we can pass these millisec as the Argument to Date Constructor, then we will get Exact Date.

```
long l = req.getDateHeader
Date D = new Date (long l)
```

— If the specified Header not representing Date Value then this method rises IllegalArgumentException.

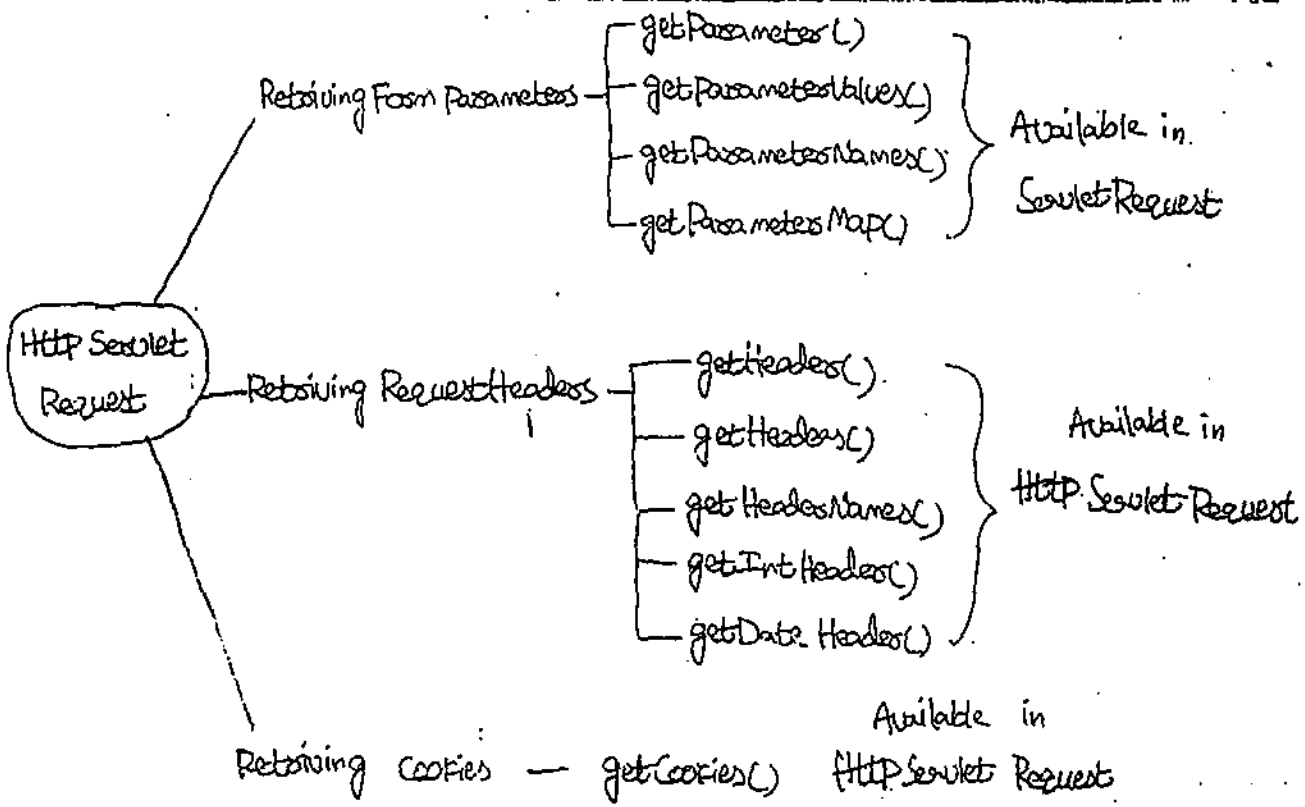
— Demo Program for Displaying all Request Header Sent by Browser

```
Public Class RequestHeaderDemoServlet extends HttpServlet
{
    Public void doGET(
    {
        PrintWriter out = resp.getWriter();
        Enumeration e = req.getHeaderNames();
        while (e.hasMoreElements())
        {
            String hname = (String)e.nextElement();
            String hvalue = req.getHeader("hname");
            O.pln (hname +" --- " + hvalue);
    } }
```

Retrieving Cookies From the Request

1. HttpServletRequest contains the following methods to retrieve cookies from the Request object.

```
Cookie[] c = req.getCookies();
```



Retriving Client & Server Information from the Request

we can retrive Client & Server Information from the Request by using the following methods of Servlet Request.

1. Public String getRemoteHost()
   - Returns Fully Qualified Name of the client which Sends the Request.

   Ex: gmail.com, yahoo.com.

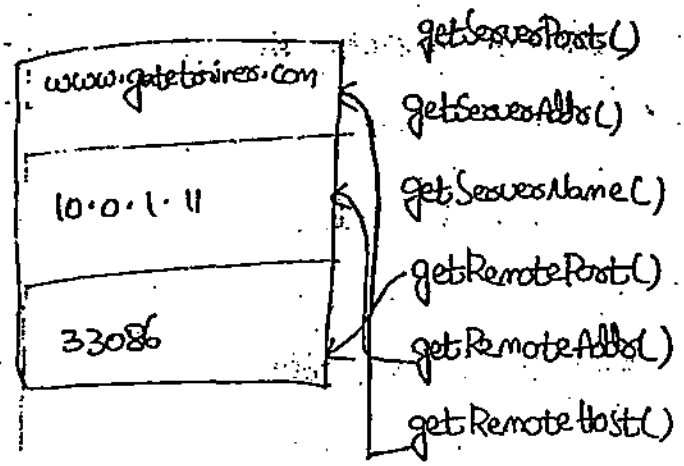
2. Public String getRemoteAddress()
   - Returns the IPAddress of the client which Sends the Request

3. Public int getRemotePort()
   - Returns the Port Number on which the client is Running.

4. Public String getServerName()
   - Returns the Name of the Server to which the request has Sent.

5. Public int getServerPort()
   - Returns the Port Number at which the Server is Running.

Even a Request from www.gatetrainer.com with an IPAddress 10.0.1.11 on Port 33086 place the appropriate Servlet Request methods on to their corresponding values.



HttpServletResponse

→ By using HttpServletResponse Interface write code

1. To set Response Headers
2. To set Content Type of Response
3. To Acquire Text Stream for Response.
4. To Acquire Binary Stream For Response.
5. To Re-Direct Request to another Url.
6. To Add cookies to the Response.

Setting HTTP Response Headers

→ HTTP Response Header Describes the Configuration Information of the Server and Information abt the Response like Content Type, content length etc..

→ Browsers using these Response Headers to Display Response Body properly to the End user.

⇒ HttpServletResponse Defines the Following methods to Add Headers to the Response.

① Public void addHeader (String hname, String hvalue);

— If the specified HeadersName is already available, to the existing values this new value will also be added.

— Replacement will not occur. $x : z, y$ (x, y)

② Public void setHeader (String hname, String hvalue);

— If the specified Header is already available than the old value will be replaced with new value. $x : \cancel{z, y}$ (w) (x, w)

⇒ Some times Headers Associated with Int & Date Values, HttpServlet Response Defines the following more Convient methods to add these methods.

③ Public void addIntHeader (String hname, int hvalue);

④ Public void setIntHeader (String hname, int hvalue);

⑤ Public void addDateHeader (String hname, long ms)

⑥ public void setDateHeader (String hname, long ms)

Set Content Type of Response

→ content Type Header Represents MIME Type (multipurpose Internet mail Extention) of the Response.

Common MIME Types

1. text/html → Html text as Response
2. application/PDF → PDF file as Response
3. image/JPEG
4. Application/zip → zip file as Response.

→ we can set MIME Type by using the following 2 ways.

1. By Servlet Response
   - Its contains the following method to set the Response

   Public void setContentType (String MIMEType);

   Ex: resp.setContentType ("application/PDF");

2. By HttpServletResponse
   - Its contains setHeader()

   Ex: resp.setHeader ("content type", "text/html");

TO Acquire Text Stream For the Response

→ we can send Text Data as the Response by using PrintWriter Object.

→ we can get PrintWriter Object by using getWriter() of ServletResponse Interface.

Public PrintWriter getWriter() throws IoException.

Ex: PrintWriter out = resp.getWriter();

2) To Acquire Binary Stream as Response

- we can send Binary Information (video files, image files etc) by using ServletOutputStream Object.
- By using getOutputStream() of ServletResponse we can get this object

Public ServletOutputStream getOutputStream() throws IoE

Ex: SOS os = resp.getOutputStream();

1\09 Demo program to send Image File as Response

```
import javax

Public class BinaryStreamDemo extends HttpServlet
{
  Public void doGET( __ ) ___
  {
    resp.setContentType("image.jpeg");
    ServletOutputStream os = ~~new ServletOutputStream~~ resp.get();
```

```
String path = getServletContext().getRealPath("sunset.JPEG");

File f = new File(path);

FIS fis = new FIS(f);

byte[] b = new byte[(int)f.length()];

fis.read(b);

os.write(b);

os.flush();

os.close();

}
}
```

f.length()
returns "long"
so we have to convert in to int
=> [] accepts int

```
SCWCDIG
|-sunset.JPEG
|-WEB-INF
  |-web.xml
  |-classes
    |-BinaryStreamDemo
```

Note:

At any Point of time we can get either PointWriter object or ServletOutputStream Object but not both simultaneously. otherwise we will get RunTime Exception Saying IllegalStateException.

Ex:

```
public void doGET(--)---
{
    PointWriter out = resp.getWriter();
    SOS os = resp.getOutputStream();
    =
}
```

(or)

```
SOS os =
PW out =
```

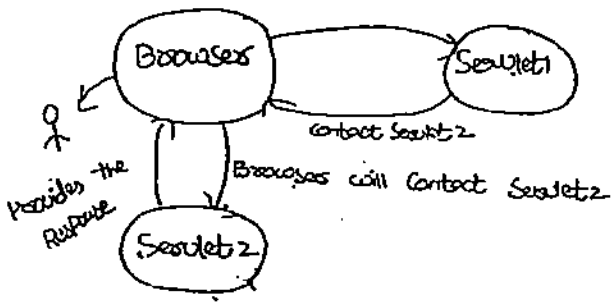
↓

RunTime Exception Saying - IllegalStateException
getWriter() has already been called for this Response.

By using which of the following stream we will send both binary & Text data as response from the Servlet.

1. PrintWriter
2. ServletOutputStream
3. ServletInputStream
4. None of the above.

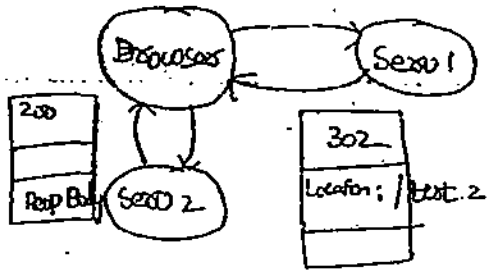
Re-Directing HTTP Request to Another URL



→ Some times we have to Re-Direct HTTP Request to another URL. we can Achieve this by using SendRedirect() of HTTPServletResponse.

Public void SendRedirect (String TargetPath) throws IOE

Process of Send Re-direction



→ Here operation takes place at Browser (i.e client side)

→ Browser sends a Request to Servlet 1.

→ If Servlet 1 is not Responsible to provide Response, then it will update Browser to contact Servlet 2. This can be Informed by setting Status code 302 & Location Header with Servlet 2.

→ Browser by seeing Status code 302 creates a new request Object & sends to Servlet 2.

→ Servlet-2 provides Desired Response to the browser & the browser inturn that Response to the End user.

Demo program For Send-Redirect

```
import

Public class FirstServlet extends HttpServlet
{
    Public void doGET( --) ---
    {
        resp.sendRedirect ("test2");
            (or)
        resp.setStatus("302");
        resp.setHeader ("Location", "stuff/test2");
    }
}
```

```
Public class SecondServlet extends HttpServlet
{
    Public void doGET( -- ) ---
    {
        PrintWriter out = resp.getWriter();
        out.Println("Hi This is second servlet");
    }
}
```

→ wherever we are sending Request to FirstServlet, Second Servlet will provide the Response through Re-Direction.

→ After commiting the Response we are not allow to perform Re-Direction. otherwise we will get RunTimeException saying Illegal State Exception (Tomcat People doesn't provide support for this).

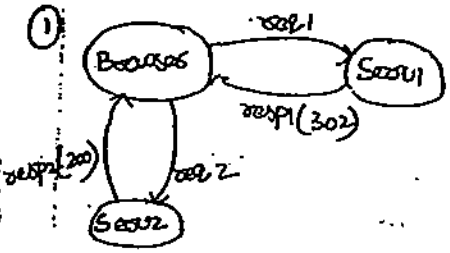
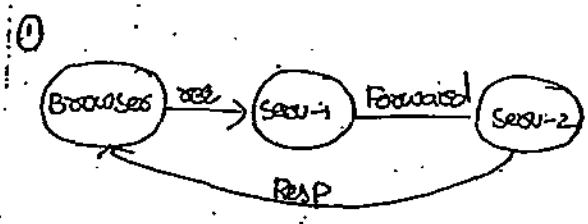
```
Public void doGET( -- )
{
        out.flush();
        resp.SendRedirect("/test2"); // IllegalStateException,
}
```

Durga —10,000→ Ravi

10,000

⇒ After getting 10,000 if Ravi asks Durga to contract Siva.

8/04/11 Differences btw send-redirection & forward mechanisms.

Send Re-Direct	Forward
	
This mechanism will work at client-side. Hence client aware of which servlet is providing the required Response.	This mechanism will work at server-side and it is not visible to the client. Hence client is not aware of which servlet is providing the required Response.
(2) In this approach an extra Trip is required to the client-side. Hence network traffic increases & effects the performance of the system.	(2) No extra Trip is required to the client & hence there are no n/w traffic & performance problems
(3) This is only possible mechanism to communicate with Resources which are present outside of web container.	(3) Forward mechanism will work with in the web container only. & we can't use this mechanism to communicate with the Resources present outside of web container.
(4) A seperate new Request Object will be created in send Re-Direction. Hence Information sharing btw the components is not possible.	(4) The same Request object will be forwarded to the second servlet & Hence Information sharing btw the components is possible in the form of Attributes.

(5) By using HttpServletResponse Object we can achive Send-ReDirection.	(5) By using RequestDispatcher Object we can achieve Forward mechanism.
(6) After commiting the Response we are not allow to perform Send-ReDirection. otherwise we will get RunTime Exception Saying Illegal State Exception.	(6) After commiting Response we can't perform Forward otherwise IllegalStateException.

Adding Cookies to the Response

→ HttpServletResponse Contains the following method to add Cookie to the Response Object.

```
Public void addCookie(Cookie c)
```

Ex:
```
Cookie c=new Cookie(String name, String Value);
resp.addCookie(c);
```

HttpServletResponse

1. Add Response Header
   - addHeader()
   - setHeader()
   - addIntHeader()
   - setIntHeader()
   - addDateHeader()
   - setDateHeader()
2. Set ContentType
   - resp.contentType("text/html");
   - resp.setHeader("contentType", "text/html");
3. To Acquire Text Stream of Response — resp.getWriter();
4. To Acquire Binary Stream of Response — resp.getOutputStream();
5. To Perform Send Re-Direction
   - resp.sendRedirect("/test2");
   - resp.setStatus(302);
     resp.setHeader("Location", "/test2");
6. Add Cookies — resp.addCookie(c);

UVII - 2

Structure and Deployment of web Applications

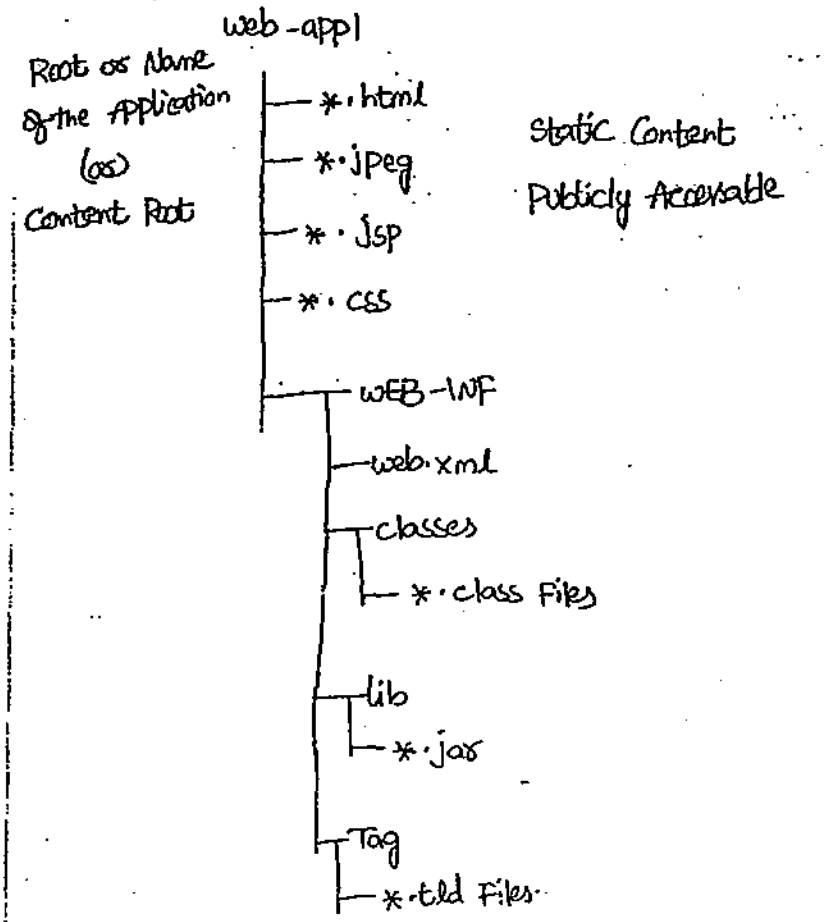
1. Directory structure of web Application
2. web.xml
3. war File.

Directory structure of web Application.

(b) Construct File & Directory Structure of web Application that may Contain

1. Static Content
2. JSP pages
3. Servlet classes
4. Deployment Descriptor (DD)
5. Tag Libraries
6. Zar Files
7. Java Class Files.

→ Servlet Specification Defines a Standard structure for web-Applications So that every web Servers can provide that Directory structure, irrespective of vendor.



All the static Information we have to place with in the Context root Directly. All the Resources present in Context Root are publicly Accessible Resource. i.e any person can Access these Resources Directly by its Name. Hence we can't place Secured Resource with in the Context Root Directly.

For every web-Application we have to maintain WEB-INF folder. All the Secured Resources we have to place in WEB-INF. And these Resources are not Publicly Accessible. If any Person trying to Access these Resources directly by their name we

will get 404 Status Code.

http://localhost:8888/web-app1/WEB-INF/web.xml ↵

→ we have to place Deployment Descriptor (web.xml) in WEB-INF Directly.

→ web-Containers will use this web.xml file to get Deployment Information.

→ with in the WEB-INF we have to place classes folder to place all java.class files (both Servlet & General Utility classes).

→ web Containers by Default will search for Required.class file in the classes folder only. Hence it is not required to set class path if we are placing .class files in the classes folder

→ with in WEB-INF we have to maintain lib folder to place all the required .jar files.

→ For the required .class file web Container first Search in the classes folder & if it is not available then only it will search in the lib folder.

→ All the Tag libraries we have to place any where with in web-INF either Directly or Indirectly.

Web-Resource	Location to place
1) Static Content	① with in Context Root Directly.
2) JSP pages	② with in Context Root Directly. If JSP is secured we have to place inside web-INF.
3) Servlet classes	③ Inside classes folder
4) Deployment Descriptor	④ with in web-INF Directly.
5) Tag Libraries	⑤ Anywhere with in WEB-INF either Directly or Indirectly.
6) Jar Files	⑥ Inside lib folder
7) Java classes Files	⑦ Inside classes folder.

Deployment Descriptor

1) Describe the Purpose, Symantics & the correct structure of the following Deployment Descriptor Elements.

① Error Page

② Init-Param

③ <MIME-mapping>

④ <Servlet>

⑤ <Servlet-mapping>

⑥ <Servlet-class>

⑦ <Servlet-name>

⑧ <welcome-file>

→ Deployment Descriptor is an .xml file named with web.xml and should be placed in WEB-INF folder Directly.

→ web Container uses this web.xml to get web-Applications Deployment Information i.e web.xml acts as a Guide to the web-Container.

→ web.xml Provides Declarative mechanism for customising our web-Application with out touching the source-code.
(To change username, pwd, To change url patterns, Initialization parameters in web.xml).

→ For every web Application we have to maintain exactly one web.xml.

Anatomy of web.xml

```
<web-app>
        <description>
        <display-name>
        <servlet>
        <servlet-mapping>
        <welcome-file-list>
        <Error-page>
        <mime-mapping>
        <context-param>
        <Security-Constraints>
        <JSP-Config>
        <Session-Config>
</web-app>
```

In total there are 27 Top level child Tags are possible

→ untill Servlet 2.3 V web.xmx is validated by using DTD's where the order is important. Hence untill servlet 2.3 V the order of these Top-Level Elements is important.

→ But from Servlet 2.4 V. onwards web.xml is validated by using Schemas where the order is not important. Hence from servlet 2.4 V onwards we can take these Top-Level Elements in any Order.

→ Among all 27 Top Level Tags no tag is mandatory. Hence the following are also valid web.xml's.

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

[or]

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

① <Servlet>

→ we can use <Servlet> to declare a servlet in web.xml.

→ Servlet Tag Contains the following 9 child Tags.

```
<web-app>
  <servlet>
        <description>
        <display-name>
        <icon>
        <servlet-name>
        <servlet-class> [or] <jsp-file>
        <init-param>
        <load-on-Startup>
        <run-as>
        <Security-role-ref>
  </servlet>
</web-app>
```

→ The order of these 9 Tags is important and we should not change,

→ Among these 9 Tags the following 2 are mandatory

① <servlet-name>

② <servlet-class> [or] <jsp-file>

② <servlet-name>

→ According to Servlet-Specification there are 3 names are possible for every Servlet.

1. Developer's Name specified by <Servlet-class>
2. Deployer's Name [or] logical-Name specified by <Servlet-name>.
3. End-user's Name specified by <url-pattern>

```
<web-app>
    <Servlet>
        <servlet-name> FirstServlet </servlet-name>
        <servlet-class> OrderProcessingServlet </servlet-class>
    </Servlet>
    <Servlet-mapping>
        <Servlet-name> FirstServlet </servlet-name>
        <Url-pattern> /order </url-pattern>
    </Servlet-mapping>
</web-app>
```

Deployer's (or) logical Name

Developer's Name

End user's Name

=> The main Advantages of logical-Name are

1. we can achieve Security by we are not Highlating our Internal Naming Convention to the End-user.
2. we can map the Same Servlet for Different url-patterns.
3. with out effecting End-user we can change our Internal-Naming Convention.

=> with in the web-Application the logical name should be unique.

→ we can get Logical_Name of Servlet with in the Servlet class by using getServletName() of ServletConfig Interface.

```
Public String getServletName()
```

Ex: Public void doGET(--)---

```
{
    PrintWriter out = resp.getWriter();
    String name = getServletName();
    out.println("The Logical Name is" + name);
}
```

<Servlet-class>

→ By using this Tag we have to specify the Original Name of Servlet.

→ For this class only web-Container performs Instantiation.

04/09 Configuring JSP in web.xml

→ Generally JSP's we can place in Context Root Directly. In this case End-Users can access Directly by its name.

→ But if the JSP is secure then it is not recommended to place in the Context Root Directly. we have to place such type of Resources Inside WEB-INF and we have to provide access through Url-pattern

→ we can Configure JSP by using <jsp-file> in web.xml.

Demo program

```
<h1> The Server Time is :
        <%= new java.util.Date() %>
</h1>
```

web-app1
- WEB-INF
  - date.jsp
  - web.xml

→ we can place JSP files in classes folder also. In that case we have to specify hole path in <jsp-file>

```
<web-app>
    <servlet>
        <servlet-name> JSPDemo </servlet-name>
        <jsp-file> /WEB-INF/date.jsp </jsp-file>
    </servlet>
    <servlet-mapping>
        <servlet-name> JSPDemo </servlet-name>
        <url-pattern> /test </url-pattern>
    </servlet-mapping>
</web-app>
```

<init-param> (Servlet Initialization parameters)

→ If the value of a variable will change frequently those values, it is not recommended to hard-code inside servlet class.

→ Bcz for every modification it requires to recompile the servlet class, Rebuild the application, Redeployment & Some times even require Server Restarts also.

→ such type of variables we have to configure in web.xml

by using <init-Param>. To reflect the changes Just Re-Deployement is enough.

```
<web-app>
    <servlet>
        <init-Param>
0 to    <Param-name> user </Param-name>
many    <Param-Value> Scott </Param-Value>
        </init-param>
    </servlet>
</web-app>
```

→ we can configure any number of Initialization Parameters but for each parameter one <init-Param>.

→ with in the Servlet we can Access Servlet Initialization parameters by using Servlet Config Object.

→ ServletConfig Interface defines the following methods to access these parameters Inside Servlet.

① Public String getInitParameter (String name)

- Returns the value associated with specified Initialization Parameter
- If the specified parameter is not available then this method returns "Null".

(2) Public Enumeration getInitParametersNames()

→ If the Servlet doesn't Contain any Initialization Parameters then this method returns Empty Enumeration Object but not NULL.

→ Generic Servlet implements ServletConfig Interface, hence Generic Servlet provides implementation for the above 2 methods.

Demo Program for Displaying all Servlet Initialization Parameters.

```
<web-app>
    <servlet>
     <s-n> DemoServlet </s-n>
     <s-c> InitializationParameters </s-c>
      <init-param>
       <param-name> Phonenumber </param-name>
       <param-value> 9848022338 </param-value>
       </initParam>
      <init-param>
           <param-name> MailID </param-name>
           <param-value> Info@durgasoft.com </param-value>
      </init-param>
     <init-param>
           <P-n> Username </P-n>
           <P-V> Scott </P-V>
    </init-param>
  </servlet>
```

```
<Servlet-mapping>
        <S-n> DemoServlet </S-n>
        <url-pattern> /test </url-pattern>
</S-m>
</web-app>
```

```
import Javax.Servlet.*;

public class InitializationParameter extends HttpServlet.
{
    public void doGET( -- ) - - -
    {
        PrintWriter out = resp.getWriter();

        Enumeration e = getInitParameterNames();

        while (e.hasMoreElements)
        {
            String pname = (String) e.nextElement();

            String pvalue = getInitParameter("pname");

            out.Println( pname + " - - - " + pvalue );

        }
    }
}
```

⇒ Servlet Initialization Parameters are key-value pairs where both key & value are String Objects only.

→ From the Servlet we can Access these parameters but we can't modify them. i.e we have only getter methods but there are no setter methods.

→ Hence these Initialization Parameters are Considered as Deployment time parameters.

Servlet-Config

→ For Every Servlet web-Container creates one Servlet-Config Object.

→ By using Servlet-Config Object Servlet can get its Configuration Information.

→ Servlet-Config Interface defines the following 4 methods.

1. public String getServletName();
2. public String getInitParameter(String name);
3. public Enumeration getInitParameterNames();
4. public ServletContext getServletContext();

<load-on-Startup>

→ usually Servlet class loading, Instantiation, Execution of init() will takes place at the time of first request. It Increases the Response Time of the first Request.

→ If we are Configuring <load-on-startup>, these stuffs will be performed at the time of either Server StartUP [or] Application Deployment.

```
<web-app>
    <Servlet>
        |
        <load-on-Startup> 10 </load-on-Startup>
        |
    </Servlet>
</web-app>
```

-> The main Advantage of load-on-Startup is, we can maintain uniform Response Time for all the Requests

-> The Disadvantage of this Tag is, Creating Servlet Objects at the beginning may effect Performance, Cause memory problems.

-> The Servlet whose <load-on-Startup> Value is less will be loaded First.

-> If 2 Servlets having the Same load-on-Startup Value or if the load-on-Startup Value is negative then we Can't Predict order of Loading. It is vendor dependent.

14/04/04 <Servlet-mapping>

→ By using this tag we can map a servlet with the url-pattern

→ upto servlet 2.4V with in the <servlet-mapping> a single servlet can map to exactly one <url-pattern> but from servlet 2.5V onwards a single servlet can map with multiple url-patterns. i.e we can take multiple <url-pattern> Tags inside servlet mapping.

```
<servlet-mapping>
    <servlet-name> FirstServlet </servlet-name>
    <url-pattern> /test </url-pattern>
    <url-pattern> /hello </url-pattern>
    <url-pattern> /order </url-pattern>
</servlet-mapping>
```

→ This is valid in servlet (2.5V) (Tomcat-6). But Invalid in servlet 2.4V (5.x)

→ According to servlet Specifications there are 4 types of url-patterns are Possible.

(1) Exact matching url-pattern

ex: /test

(2) Longest path prefix url-pattern

ex: /test/test/*

(3) url-pattern by Extension

*.ext ex: *.do, *.jsp

(4) Default Url-Pattern.

ex: /

1) which of the following url-patterns are valid.

a) /test

b) /

c) /test/test/*/test

d) /test/*.JSP

e) /*.do or /*.durga.

→ web-Container always gives the precidence in the following order

1. Exact match
2. LongestPath Prefix
3. By Extension.
4. Default url-pattern.

⇒ If no other url-pattern matched then only default url-pattern will get a chance.

Demo program for url-patterns.

1. /test — FS
2. /test/test/* — SS
3. *.do — TS
4. / — Default Servlet.

http://localhost:8080/scwcd2c/test → First Servlet (41)

/test/test/durga.do → Second Servlet

/test/durga.do → Third Servlet

/durga.do → Third Servlet

/durga → Default Servlet.

How we can Configure DefaultServlet in web.xml & in which cases Default Servlet will get the chance.

with "/" we can Configure Default Servlet. If no Servlet url-pattern is matching the DefaultServlet will be executed.

⇒ Getting Information from the url.

→ Servlet Request Interface defines the following methods for this.

1. getRequestURI();
2. getContextPath();
3. getServletPath();
4. getPathInfo();
5. getQueryString();

```
Public class Fs extends HttpServlet
{
    Public void doGET(--)---
    {
        PointWriter out = resp.getWriter();

     out.Pointln("<h1> RequestURI : " + req.getRequestURI()+"</h1>");
     o.pln("<h1> Contextpath : " +req.getContextPath()+"</h1>");
     o.pln("<h1> ServletPath : " + req.getServletPath() "+ </h1>");
     o.pln("<h1> pathInfo : " +req.getPathInfo() + "</h1>");
     o.pln("<h1> QueryString : " +req.getQueryString() + "</h1>");

    }
}
```

```
<w-a>
        FS
        ↓
        /test/test/*
<|w-a>
```

http://localhost:8080/SCWCD2CA/test/test/durga/software?

uname = Durga & uvalue = DSS

RequestURI — /SCWCD2A/test/test/durga/software

Contextpath — /SCWCD2CA

ServletPath : /test/test

PathInfo : durga /Software

QueryString: uname =Durga & uvalue=DSS

http:localhost:8080 /SCWCD2CA /test /test/

Request URI : /SCWCD2CA /test /test

Context path : /SCWCD2CA

Servlet path : /test /test

Path Info : /

Query String : null

http:localhost :8080 /SCWCD2CA /test /test

PathInfo : null

QueryString : null

104 Configuring welcome Pages for web Application.

→ It is recommended to configure welcome files in web-Application

→ we can configure welcome-files in web.xml as follows

```
<web-app>
    <welcome-file-list>
        <welcome-file> home.jsp </welcome-file>
        <welcome-file> login.jsp </welcome-file>
    </welcome-file-list>
    :
</web-app>
```

→ <welcome-file-list> is the Direct child Tag of <web-app> and we can take any where.

→ we can Configure more than one welcome file but the order is important.

In any web-Application Index.html acts as Default welcome page. If it is not available then Index.jsp acts as Default welcome page.

→ whenever we are configuring welcome-files explicitly in web.xml then this Default welcome-file is not applicable.

we can Configure welcome files even Folder wise also.

According to Servlet Specification leading or tailing back slashes are not allowed for the <welcome-files>.

Ex: <welcome-file> /login.jsp </welcome-file> ✗ Invalid

```
webapp1
 |- home.jsp
 |- Durga
 |   |- login.jsp
 |   |- Durga1
 |   |    |- home.jsp
 |   |    |- login.jsp
 |- Durga2
 |   |- index.jsp
 |- Durga3
     |- form.html
```

```
<w-app>
  <w-f-L>
    <w-f>home.jsp</w-f>
    <w-f>login.jsp</w-f>
    <w-f>index.jsp</w-f>
  </w-f-L>
</w-ap>
```

http://localhost:8080/web-app1 – home.jsp

/web-app1/durga – login.jsp

/web-app1/durga/durga1 — home.jsp

/web-app1/durga2 – Index.jsp

/web-app1/durga3 – 404 Status code Error.

If we have both index.html & index.jsp, Index.html will be executed.

If we are not using <w-f-l> then only Default index.html will be executed.

Default html (or) jsp is not applicable for sub-folders. But only for Context root folder.

Configuring Error pages in web.xml

It is not recommended to send Error Information Directly to the End user. we have to convert Java Error Information in to End user understandable form.

- we can achieve this by Configuring Error pages in web.xml.
- we can configure Error page either based on Status code or based on Exception Type.

Configuring Error page based on Exception Type

```
<web-app>
    <error-page>
        <exception-type> java.lang.A-E </exception-type>
        <location> /error.jsp </location>
    </error-page>
</web-app>
```

Configuring Error page based on Status Code.

```
<web-app>
    <error-page>
        <error-code> 404 </error-code>
        <location> /error404.jsp </location>
    </error-page>
</web-app>
```

```
Public class FS extends HttpServlet
{
    public void doGET(--)---
    {
        PW out = resp.getWriter();
        o.pln (10/0);
    }
}
```

Error.jsp

```
<h1> Your Provided Values
are not correct plz Provide
Valid Input. </h1>.
```

```
<web-app>
    FS
    ↓
    /test
  <error-page>
    <exception-type> Java.lang.A-E </exception-type>
    <location> /error.jsp </location>
  </error-page>
</web-app>
```

⇒ If we are sending Request to the Servlet Instead of getting Arithmatic Exception we will get error page as response.

14/07

→ We can set Error Code programmatically.

→ HttpServlet Response Contains the following method for this.

```
Public void sendError(int ErrorCode);

Ex: resp.sendError(503);
```

login.html

```
<form action="/SCWCD/test" >
Enter name : <input type=text name=uname >
  <input type=Submit >
</form action>
```

DS.java

```
Public class DS extends Hs
{
   Public void doGET(--)---
   {
      Pw out=resp.getWriter();
      String name=req.getParameter("uname");
      if(name.equals("Durga"))
      {
         o.println("<h1> Authentication Successfull </h1>");
      }
      else
```

```
        resp.sendError(503);
    }
}
```

web.xml

```
<w-ap>
    <error-page>
        <error-code> 503 </error-code>
        <location> /error503.jsp </location>
    </error-page>

        FirstServlet
            ↓
            /test

</w-a>
```

Error503.jsp

```
<h1> your Authentication is fail ... Plz provide valid credentials
                                                        </h1>
    (we can keep hyperlink to login page)
```

(1) we can Configure Error Page either based on ErrorCode or ExceptionType but not both simultaneously in the same Error-Page Tag.

```
<error-page>
    <error-code> 503 </error-code>
    <exception-type> java.lang.AE </exception-Type>
</error-page>
```

X → Invalid Configuration.

(2) The Exception-Type Tag Value Should be Fully Qualified Name of Exception.

Ex:

```
<error-page>
    <exception-type> IoException </exception-type>
    <location> /error.jsp </location>
</error-page>
```

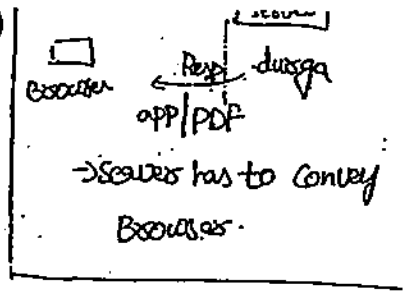
Fully Qualified Name Required
i.e java.io.IoException

(3) The <location> value must be compulsory starts with "/". otherwise we will get Deployment problems.

Ex:

```
<error-page>
    <error-code> 503 </error-code>
    <location> error.jsp </location>
</error-page>
```

"/" must be Required
i.e /error.jsp



<mime-mapping>

→ we can use <mime-mapping> to map File Extension with the Corresponding <mime-type>

Ex:

```
**<mime-mapping>
        <extention>durga</extension>
        <mime-type>application/PDF <mime-type>
    </mime-mapping>
```

→ <mime-mapping> is the Direct child Tag of <web-app> & Hence we can take any where.

War File

b) Explain the purpose of war File & Describe the content of war File & the process of Construction.

→ war File (web-Archieve) provides a convinent way to store the resources of a web-Application in to a single Component.

→ we can deliver, Transport & Deploy web-Application very easily. If it is available in the War-File form.

→ Sun People provides a standard structure for the war-File & every web-Servers provides support for that war-File.

→ A war-File Represents a web-Application where as Ear-File represents an Enterprise Application.

→ Jar-File Represents a group of .class Files.

Commands

1. Creation of war-File

```
jar -cvf myapp.war *.*
```

2. Extraction of war-File

```
jar -xvf myapp.war
```

3. Display Contents of war-File with Out Extraction

```
jar -tvf myapp.war
```

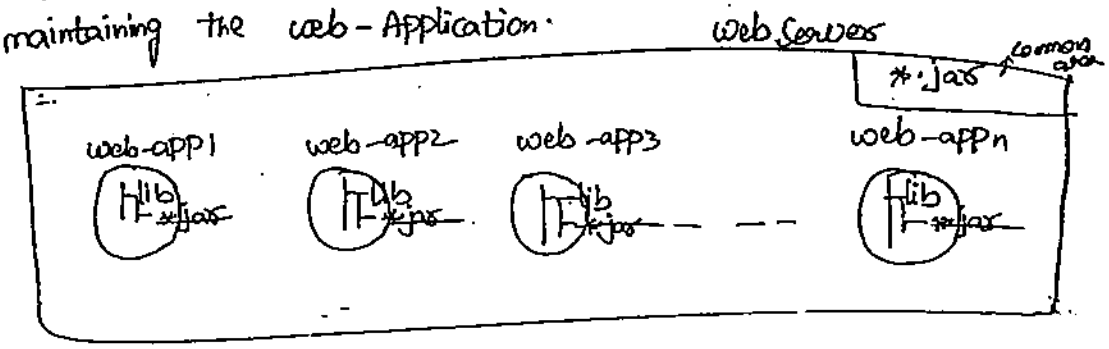
Structure of war-File

```
myapp
 |- META-INF
 |   |- MANIFEST.MF
 |
 |- WEB-INF
 |   |- web.xml
 |   |- classes
 |   |   |- *.class
 |   |- lib
 |   |   |- *.jar
```

META-INF

→ Every war-File should compulsory contain meta-INF folder which should contain MANIFEST.MF, i.e META-INF & MANIFEST.MF are mandatory for any war-File.

→ META-INF Folder contains Security Related Resources like signature Files, Digital certificates which are necessary for maintaining the web-Application.



MANIFEST.MF

→ The library Dependencies of a web-Application we can define in this file only.

→ It contains the class path of the common Jars Files stored outside of web-Application.

→ If there is any Jar File which is common to several web-Application then it is not recommended to place that Jar File at application level. we have to place such type of Jar's at some common location outside of web-application & we can define their paths in MANIFEST.MF.

→ If we are trying to Access the Resources Present in META-INF

(or WEB-INF) we will get 404 Status Code i.e these Resources are not publicly Accessable.

http://localhost:8080/myapp/META-INF/MANIFEST.MF

↓

404-Status Code Error

The web Container Model

Servlet Context

For the Servlet Context Initialization parameters

1. write Servlet Code to Access Initialization Parameters.
2. Create the Deployment Descriptor Elements for the Initilization parameters.

For every Servlet, web Container will create one Servlet Config Object to maintain Servlet level Initialization parameters. By using this Object Servlet can get its Configuration Information.

Similarly for every web-Application, web container creates one Servlet Context object to maintain Application level Configuration Information.

Servlet can get Application level Configuration Info through this Context Object only.

Servlet Config per Servlet where as Servlet Context per web-Application.

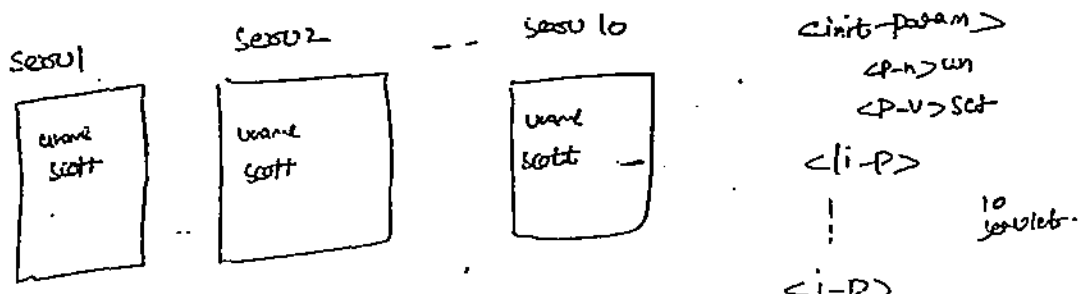
If Initialization Parameters are common for several Servlets then its not recommended to declare those parameters at Servlet level. we have to declare such type of Parameters at Application-level by using <Context-Param>

```
<web-app>
    <context-param>
        <param-name> uesrname </param-name>
        <param-value> Scott </param-value>
    </context-param>
    :
</web-app>
```

→ we can Declare any numbers of Context parameters but one <Context-param> for each parameter.

→ <context-param> is the Direct child Tag of web-app & Hence we can Declare anywhere with in <web-app>

→ The Context Initialisation parameters are available through out the web-Application anywhere.



→ If we want to use same init-parameters then we can declare at Context-level as

```
<Context-Param>
    <P-n>
    <P-V>
</context-param>
```

→ with in the servlet we can access these Context Initialization parameters by using ServletContext Object.

→ we can get Servlet Context Object by using getServletContext() of ServletConfig Interface.

```
ServletContext context = getServletContext();
```

→ ServletContext Interface Defines the following methods for Accessing Context Initialization Parameters.

① String getInitParameter(String name);

② Enumeration getInitParameterNames();

Demo Program to Display Context Initialization Parameters.

```
Public class SDemoServlet extends HttpServlet
{
    Public void doGET(--)---
    {
        Pw out = resp.getWriter();
        ServletContext context = getServletContext();
        Enumeration e = Context.getInitParameterNames();
        while (e.hasMoreElements)
        {
            String pname = (String) e.nextElement();
            String pvalue = Context.getInitParameter("pname");
        } } } o.pln(pname + "----" + pvalue);
```

web.xml

```
<w-a>
    <i-p>
        <p-n> uname <|p-n>
        <p-v> scott <|p-v>
    <|i-p>

    <i-p>
        <p-n> pwd <|p-n>
        <p-v> Tiger <|p-v>
    <|i-p>

        ScDemoServlet
            ↓
            /test
<|w-a>
```

109

Note:

```
<i-p>
    <p-n> movie
    <p-v> Bill9
<|i-p>
```

→ we can Access Servlet Initialization parameters in the following ways.

String value = getInitParameter ("movie");

String value = getServletConfig().getInitParameter ("movie");

→ we can Access Context Initialization parameters in the following ways.

String value = getServletContext().getInitParameter("movie");

String value = getServletConfig().getServletContext().getInitParameter ("movie");

→ By Default Initialization parameters means servlets

Initialization parameter.

Comparision btw

comparision btw Servlet & Context Initialization parameters.

Properties	Servlet Init-Param	Context Init-Param
1) Deployment Descriptor Declaration.	By using <init-param> with in Servlet <Servlet> <init-param> <Param-name> <param-Value> </init-param> </Servlet>	By using <Context-Param> with in <web-app> <web-app> <context-Param> <Param-name> <Param-Value> </context-Param> </web-app>

2) Servlet Code to Access the Parameters	String Value = getInitParameter("Pname");
	(or)
	String Vale = getServletConfig().
	getInitParameter("Pname");
		String Value = getServletContext().
		getInitParameter("pn");
		(or)
		String Value = getServletConfig().
		getServletContext().getInitParameter("Pname");

(5) Availability (scope)	Available only for a particular servlet in which <init-param> is declared.	Available for all servlets & JSP's with in the web-application.

Note

whether Servlet or Context Init-parameters, these are Deploy time Constants i.e from the Servlet we can Read their values but we are not allow to modify i.e we have only getter()'s but not Setter()'s.

Servlet Scopes (2 to 3)

1) For the fundamental Servlet Attribute Scopes (Request, Session, Context).

(a) write ServletCode to add, Retrive & Remove Attributes

(b) Given a usage Scenario Identify the proper Scope.

(c) Identify multiThreading Issues Associated with each Scope.

2) There are 3 types of parameters are possible.

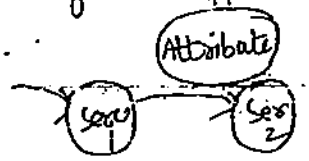
(1) Form parameters.

(2) Servlet Initialization parameters.

(3) Servlet Context Parameters.

These parameters are Read only i.e from the Servlet we can perform Read Operation, we can't modify & Remove these Values based on our requirement. Hence parameters are not useful for sharing the data btw the Components of a web-Appli.

To Resolve this problem Sun people introduced Attributes Concept. Based on our Requirement we can create a new Attribute or we can modify & Remove existing Attributes. Hence Attributes are best suitable for storing & sharing the Data btw the components of web-Applications.



→ Based on our Requirement we have to save the Attributes in the proper scopes.

→ There are 3 scopes are possible for the Servlets.

① Request Scope – ATM – credit card.

② Session Scope – login → uname, pwd for our users

③ Application / Context Scope. Globally, for any number of users.

Request Scope

→ we can maintain this scope either by using ServletRequest or HttpServletRequest Object.

0 ✓ SR object creation
P V service(--)
Ø destruction

→ Request Scope will start at the time of request Object creation (i.e Just before starting service method) & Ends at the time of Request Object Destruction. (i.e Just after completing service()).

→ The Data (or Attribute) stored in the Request Scope are available to all components which are processing that Request.

→ Servlet Request Interface defines the following methods for Attribute management in the Request Scope.

① public void setAttribute (String name, Object value).

– If the specified Attribute is already available then the old value is replaced with new value.

② Public Object getAttribute (String name).

- Retrieves the value associated with specified Attribute Name.
- If the specified Attribute is not available then this method returns "Null"

③ Public void removeAttribute (String name)

④ Public Enumeration getAttributeNames();

Example

→ The most common Area where we can use Request Scoped Attributes is Request Dispatcher Forward & Include mechanisms.

→ For every Request a seperate new request object will be created which is responsible Accessed by only the current thread. other Threads are not allowed to Access Request scoped Attributes. Hence Request SCOPed Attributes are always Thread safe.

Session Scope

→ Session scope is maintained by HttpSession Object

→ Session scope will start at the time of Session Object creation (HttpSession hs = req.getSession();) & will lost at the time of Session Object destruction. It may be either by invalidate() or by TimeOut mechanism

→ The information stored in the Session Scope is available for

all the components which are participating in that Session.

→ HttpSession Interface defines the following methods for Attribute management in Session Scope.

① Public void setAttribute(String name, Object value);

② Public ~~void~~ object getAttribute (String name);

③ Public void removeAttribute (String name);

④ Public Enumeration getAttributeNames();

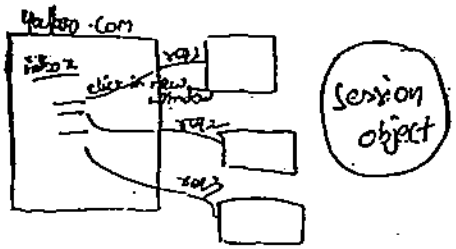
→ once the Session Expires we are not allowed to call these methods violation leads to RunTime Exception Saying

Illegal State Exception.

Example

→ login Information should be available for Entire Session. Hence we have to store this information in the Session Scope.

⇒ with in the same session we can send multiple Requests simultaneously by opening new Browser windows. Hence Session Objects can be Accessed by multiple Threads Simultaneously & Hence Session Scoped Attributes are not Thread Safe.



Application Scope

-> Application Scope is maintained by Servlet Context Object.

-> It starts at the time of ContextObject creation. (ie at the time of Application Deployment or Server Startup). & will be lost at the time of Context Object Destruction. (ie At the time of Application un-Deployment or Server ShutDown).

-> The Data stored in the Application scope will be available to all the components of web-application, irrespective of request & users.

-> ServletContext Interface defines the following methods for Attribute management in the Application Scope.

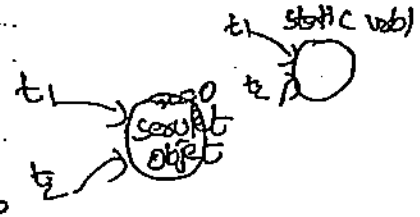
1. Public void setAttribute (String name, Object value);
2. public ~~void~~ object getAttribute (String name);
3. public void removeAttribute (String name);
4. public Enumeration getAttributeNames ();

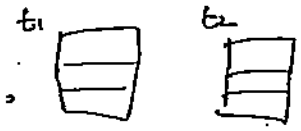
-> Context Object can be Accessed simultaneously by multiple Threads & hence Context Scoped Attributes are not Thread-Safe.

-> Instance & static variables can be accessed multiple threads simultaneously & hence these are not thread safe

-> For every thread a seperate copy of local variables will be created & hence local variables are always thread safe.

```
public class DS extends HS
{
    int x=0 ; — Not Thread Safe
    static int y=0; — Not Thread Safe

    Public void Service (—) ---
    {
        int z=0; Thread Safe
    }
}
```





members	Thread Safe
1) Requeste Scope Attributes	Yes
2) Session Scope "	No
3) Context Scope "	No
4) Instance Vars	No
5) Static Vars	No
6) Local Vars	Yes

→ Parameters are key-value pairs where both key & value are String objects only. Hence at the time of Retrival it is not required to perform any typecasting. we can assign Directly parameter value to the String Directly.

```
String value = request.getParameter("uname");
String value = request.getInitParameter("uname");
```

→ Attributes are also key-value pairs. But keys are string objects but values can be any Type of object. Hence at the time of Retrival compulsory we should perform typecasting.

```
X String value = request.getAttribute("uname"); --> CTError
                                                    Incompatible Type
                                                    found : object
                                                    required : String

✓ String value = (String) request.getAttribute("uname");
```

→ To Access the value of RequestScoped Attribute uname which of the following is valid way.

(1) String value = request.getParameter("uname");

(2) String value = request.getInitParameter("uname");

3) String value = getInitParameter ("uname");

4) String value = request.getAttribute ("uname"); Person type is object

5) String value = (String) request.getAttribute ("uname");

Differences btw Parameters & Attributes.

Parameters	Attribute
1) Parameters are Read only i.e with in the Servlet we can perform Read operation but we can't modify their values i.e we have only getter()'s but not setter()'s.	1) Based on our Requirement we can get & set the Attributes i.e these are not Read only & we have both getter & setter methods.
2) Parameters are Deployment Time Constants.	2) Attributes are not Deploy Time Constants.
3) Parameters are key-value pairs & both key & value are String objects only. Keys – String, Values – String	3) Attributes are Key-value pairs where key is String but value can be any type of object. Key – String, Value – Object (any object).
4) At the time of Retrival it is not necessary to perform Type Casting.	4) At the time of Retrival we should perform Typecasting.

Demo program to Display Hit-Count (no. of request.) of web-Application.

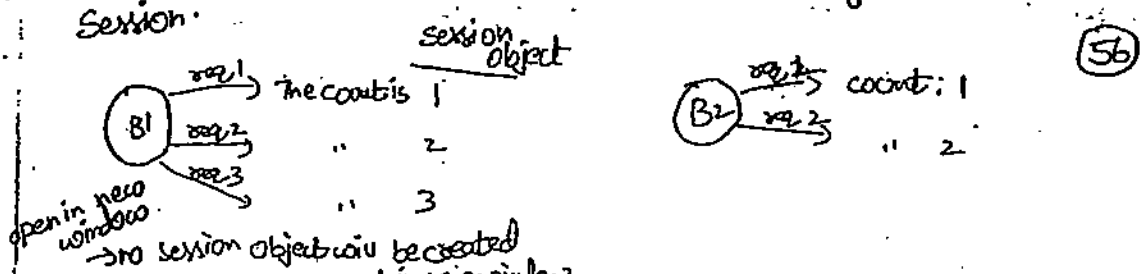

```
import java.util.*

Public class HitCountDemo extends HttpServlet.
{
  Public void doGET(--)---
  {
     PrintWriter out = resp.getWriter();
     ServletContext context = getServletContext();
     Integer count = (Integer) context.getAttribute("hitCount");
      if(count==null)
      {
          Icount=1;
      }
      else
      {
          count++;
      }
      Context.SetAttribute("hitcount", count);
      out.pln("The hit count is", +count);
  }
}
```

Demo Program for Displaying the number of users currently login (who are in on-line).

```
import java.util.*;
```

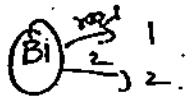
```
Public class usersCount extends HttpServlet
{
   Public void doGET(--)--
   {
     PrintWriter out = resp.getWriter();
     ServletContext context = getServletContext();
     Integer count = (Integer) context.getAttribute("usersCount");
     HttpSession session = req.getSession();
     if(session.isNew())
     {
        if(count==null)
        {
           count=1;
        }
        else
        {
           count++;
        }
        context.setAttribute("usersCount", count);   -> we can keep outside of if also.
     }
     out.println("The no. of usersCount Value", +count);
   }
}
```

→ Demo program to display the total no. of request in each session.




```
import java.util.*;
Public class RequestHitCount extends HttpServlet
{
    Public void doGET(---)---
    {
        PrintWriter out=resp.getWriter();
        HttpSession session = req.getSession();
        Integer count=(Integer) session.getAttribute("requestCount");
        if (count==null)
        {
            count=1;
        }
        else
        {
            count++;
        }
        session.setAttribute("requestCount", count);
        o.pln("The no. of request in current session is", +count);
    }
}
```

Demo program to Display no. of Request from each IPAddress.



open another browser

req -> 3

=> Its for same IPAddress.

```
import java.util.*;
Public class IPRequestCount  extends HttpServlet
{
    Public void doGET(---)---
    {
     PrintWriter out = resp.getWriter();
     ServletContext context = getServletContext();
     String IPAddress = req.getRemoteAddr();
     Integer count = (Integer) Context.getAttribute(IpAddress);
     if (count==null)
     {
          count=1;
     }
     else
     {
       count++;
     }
     Context.setAttribute("IPAddress", count);
    o.println("The no. of request from " +IPAddress+ "is" +count);
    }
}
```

23/04/09

Request Dispatcher

→ (1) Describe RequestDispatcher mechanism.

(2) write ServletCode to create RequestDispatcher.

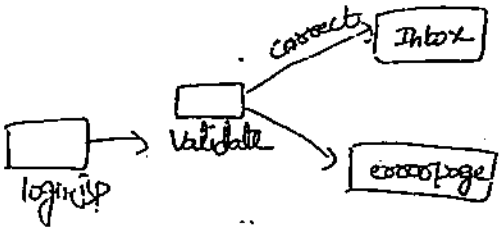
(3) write ServletCode to Forward or Include TargetResource.

(4) Identify & Describe the Attributes Added by web-Container while Forwarding & Including.

→ It is not recommended to Define Entire functionality in a single Component. It has Several Serious Disadvantages.

(1) with out effecting Remaining Components we can't modify any Component. Hence enhancement will become very complex & maintainability will be down.

(2) It doesn't promote Re-usability.



→ we can resolve these problems by maintaining a seperate component for each task.

→ The main Advantages of this Approach are without effecting remaining Components we can modify any Component. Hence enhancement will become very easy & improves maintainability.

→ It promotes Re-usability.

Ex: where ever Validation is required, we can use the same Validate Servlet with out Re-writting.

→ If Entire functionality is Distributed across several Components, these Components have to communicate with each other to provide Response to Endusers. we can achieve this communication by using Request Dispatcher.

Servlet Code for getting RequestDispatcher

→ we can get Request Dispatcher either by using ServletRequest object or Servlet Context object.

(1) By ServletRequestObject

→ ServletRequest Interface defines the following method for this.

public RequestDispatcher getRequestDispatcher (String TargetResource).

— The Target Resource can be specified either by Absolute path or Relative path.

Ex: RequestDispatcher rd = req.getRequestDispatcher("/test2"); (58)
Absolute path

RequestDispatcher rd = req.getRequestDispatcher("test2");
Relative path

→ If the target Resource is not available we will get 404-Status code saying Required Requested Resource is not available

(7) By Servlet Context Object.

→ Servlet Context Interface Defines the following 2 methods for getting RequestDispatcher Object.

(a) Public RequestDispatcher getRequestDispatcher(String TargetResource)

→ The target Resource should be specified by only Absolute Path

→ If we are using Relative path i.e if the path not starts with "/" we will get RuntimeException Saying IllegalArgument Exception.

Ex: ✓ RequestDispatcher rd = context.getRequestDispatcher("/test2");

✗ RequestDispatcher rd = context.getRequestDispatcher("test2"); Error

→ If the target Resource is not available we will get 404 status code Saying Requested Resource is not available.

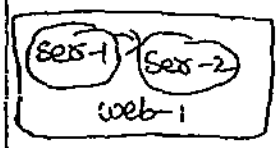
(b) Public RequestDispatcher getNamedDispatcher(String Servlet-name)

→ The Argument represents the Value Associated with the <Servlet-name> in web.xml.

→ If the specified Servlet is not available then we will get 'null'. on that null if we are trying to call forward we will get NullPointerException.

Ex: RD rd = context.getNamedDispatcher("FirstServlet");

Difference btw getting Request Dispatcher by ServletRequest & by Servlet Context.

RD from ServletRequest	RD from ServletContext
1) RD rd=req.getRequestDispatcher("/test2"); → we can specify the target Resource either by Absolute or Relative Path.	① RD rd= context.getRD("/test2"); → The target Resource Path can be specified only by Absolute Path. otherwise we will get Illegal Argument Exception.
2) we can't get foreign Request Dispatcher by Request Object 	② we can get foreign RD by Context Object.
3) we can't get RD by using <servlet-name>	③ we can get RD by <servlet-name> for this Servlet Context Defined getNamedDispatcher();

methods of Request Dispatcher

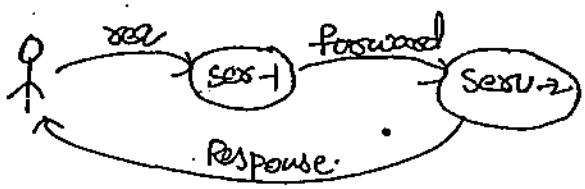
① Request Dispatcher defines the following 2 methods.

① public void Forward (SR req, SResp resp) throws SE, IoE

② public void include (SR req, SResp resp) throws SE, IoE

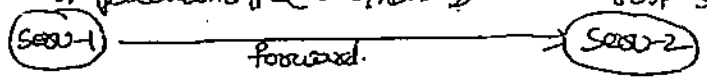
Forward Mechanism

→ If the First Servlet is responsible for some Preliminary policies & 2nd Servlet is Responsible to provide Required Response then we should go for Forward mechanism.



→ Just before forwarding, the ~~Request~~ Response Object will be created automatically by web-container. Hence if any Response added by the first Servlet won't be delivered to the End users.

→ Second Servlet has Complete Control on the Response Object It can change the Response Headers also.



→ The same Request Object will be forwarded to the 2nd Servlet Hence Information Sharing btw the components in the form of Request Scoped Attributes.

Note

→ once the Response Commited we are not allowed to call forward mechanism. Violation leads to Illegal State Exception.

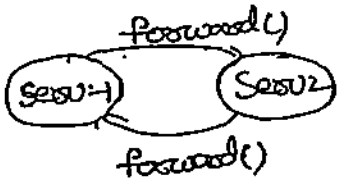
Ex: PW out = resp.getWriter();

out.pln ("Hai");

out.pln ("flush"); committed → Ready to send the Resp Object

rd.forward (req, resp); Error → IllegalStateException.

→ Recursive forward call rises StackOverflowError.



→ After forward call the control comes back for execution of Remaining Stmts. In those stmts if we are trying to add any Response those stmts will be Ignored by the web-container.

Example-1

```
Public class FS extends HS
{ P V doGET (--)---
  {
    PW out = resp.getWriter();
    o.pln (" FirstServlet");
    RequestDispatcher rd = req.getRequestDispatcher("/test");
    rd.forward (req, resp);
    S.o.pln(10/0) ; [or] out.pln(10/0); Exception
  }
}
```

```
Public class SecondServlet extends HttpServlet.
{
    P V doGET(--)---
    {
        out.pln("Second Servlet");
    }
}
```

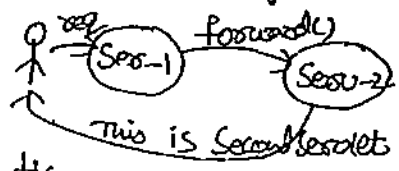
⇒ when ever we are sending the Request to 2nd Servlet we will get Arithmatic Exception has the Response.

Example-2

FirstServlet.java

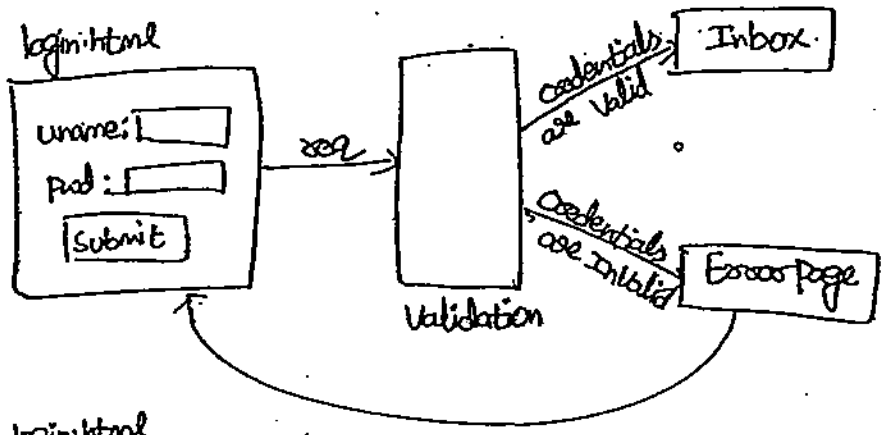
```
Public class FS extends HS
{
    P V doGET(--)---
    {
        PW out = resp.getWriter();
        O.pln("This is first Servlet");
    RequestDispatcher rd = req.getRequestDispatcher("/test2");
        rd.forward(req, resP);
        out.pln("This is first Servlet again");
        S.O.pln("The control comes back again to FS");
    }
}
```

Ignored by web Container



SecondServlet.java

```
Public class SS extends HS
{
    P V doGET(--)---
    {
        PW out = resp.getWriter();
        out.pln("This is Second Servlet");
    }
}
```



login.html

```
<html> <body> This is forward Demo>
 <form action = "SCWCD /test1" >
  Enter Name : <input type = text  name = uname >
  Enter Pwd : <input type = text  name = Pwd>
    <input type = submit>
 </form> </body> </html>
```

validation.java

```
Public class validation extends HS
{   P V doGET(--) ---
    {
        Pw out = resp.getWriter();

    String name = req.getParameter("uname");
    String Pwd = req.getParameter("Pwd");
  if(name.equals("Durga") && Pwd.equals("SCWCD"))
  {
        RD rd = getServletContext.getRD("/inbox.jsp");
          rd.forward(req, resP);
  }
```

```
else
{
    RD rd = req.getRD("/error.jsp");
    rd.forward(req, resp);
  }
}
}
```

Inbox.jsp

This is your Inbox.
check your mails.

Error.jsp

This is Error page. Credentials are invalid. Please Login again
<a href="/SCWCD/login.html">
Login </a>.

01/01/09

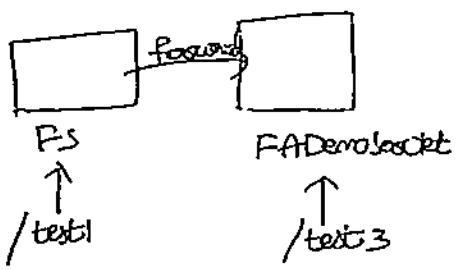
Attributes Added by web-Container while forwarding the Request.

→ while forwarding the request from one Servlet to another Servlet the Container adds some attributes in the Request Scope. These attributes provides Information to second Servlet about original Request.

Attribute Name	Description	Equivalent Servlet method.
① javax.servlet.forward.request_uri	The uri of the original Request [/webapp/test/test/dugga/software]	getRequestURI()
② javax.servlet.forward.context_Path	The context path of the application [/webapp]	getContextPath()

3) javax.servlet.forward.servletPath	The servletPath of the forwarding Servlet [/test/test]	getServletPath();
4) javax.servlet.forward.pathInfo	Returns the extra path Info associated with Request URI. i.e this is the String which follows Servlet path but precedes QueryString. [durga/software]	getPathInfo();
5) javax.servlet.forward.queryString.	The QueryString Associated with the Request.	getQueryString();

Demo program



→ Through forward call only FADS will point the added Attribute

→ If we are sending Direct call to

```
Public class FS extends HS
{
    P. V doGET(- -) ---        Context.getRD("/test3");
    {                          (or)
        RequestDispatcher rd = req.getRequestDispatcher("/test3");
        rd.forward(req, res);
    }
}
```

Forward AttributeDemo.java


```
public class FAD extends HS
{
    P V doGET(--) ---
    {
        PrintWriter out = resp.getWriter();
        out.println("Forward Request Attribute");
        Enumeration e = req.getAttributeNames();
        while (e.hasMoreElements())
        {
            String name = (String)e.nextElement();
            Object value = req.getAttribute(name);
            out.println(name + "---" + value);
        }
    }
}
```

→ If we are sending the Request to the forward DemoServlet through forward call then we will get all the Attributes which are added by web Container.

http://localhost:8080/scwcd/test1?uname=durga&pwd=PR.

→ If we are sending the Request Directly to the Forward Attribute DemoServlet, then we won't get any Attribute.

→ If we are performing forward by the RequestDispatcher which is obtained by getNamedDispatcher() then WebContainer won't add any Attributes in the request Scope.

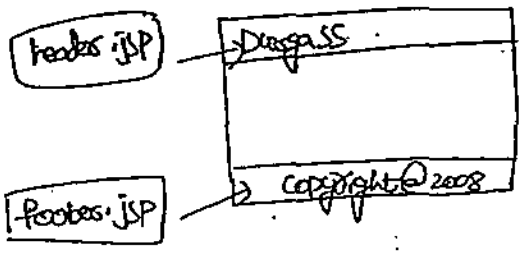
RD rd = Context.getNamedDispatcher("FDServlet");

rd.forward(req, resp);

=> In this web-Container won't add any Attributes.

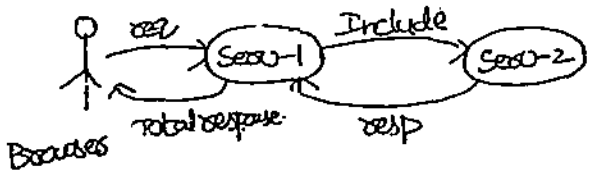
-> After Forwarding the request if we are trying to forward in the same Servlet we will get IllegalStateException.

Include Mechanism



-> If we want DSS, copyright in 20 JSP pages make it seperate as header, footer. where ever useful Just Include that JSPs.

->



-> This mechanism is helpful to include the Response of other Resources in the current Response.

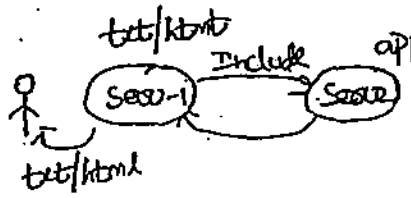
-> This Inclusion mechanism is best suitable for Including Banner Information like copyright, Logo etc.

-> The Servlet who get the request Initially is responsible to Provide Response.

-> After commiting the Response we can perform include() call

but we can't perform forward call.

In the Include, the Second Servlet doesn't have complete control on the Response object. It is not allowed to change the Response headers. If it is trying to perform any changes these will be ignored by the web-Container.



application/PDF → changing the headers in SS will be ignored by web Container.

FirstServlet.java

```
public class FS extends HS
{
  P v doGET(--) ---
  {
     PW out = resp.getWriter();
     o.pln ("This is firstServlet");
     RequestDispatcher rd = req.getRequestDispatcher ("/test2");
     rd.include (req, resp);
     o.pln ("This is firstServlet Again");
  }
}
```

SecondServlet.java

```
     PW out = resp.getWriter();
     o.pln("This is SS");
```

→ wherever we are sending the request to the Main servlet the following is the o/p.

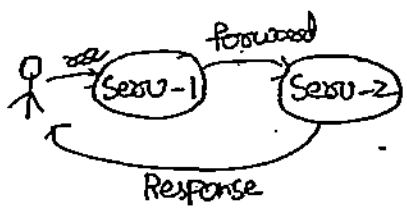
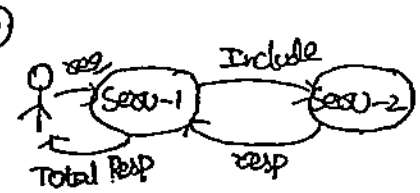
This is FS
This is SS
This is FS again.

→ Attributes added by web-Container while include call in Request Scope.

1. javax.servlet.include.request-Uri
2. " " " Context-Path
3. " " " Servlet-Path
4. " " " Path-info
5. " " " query-String.

→ If we are getting RequestDispatcher by using getNamedDispatcher then webContainer won't add any Attributes in the RequestScope.

Difference b/w forward & Include

Forward	Include
① 	① 
→ once a servlet forwards the Request to another servlet then the forwarded servlet (S-2) is complete Responsible to provide required Response.	→ In the case of Include the Including servlet (S-1) is Responsible to provide the required Response.
② while Performing forward() the response Object will be cleared by the web-Container.	② In the Include web Container won't clear the Response Object.
③ After commiting the Response we are not allowed to Perform forward() otherwise - Illegal State Exception.	③ After commiting the Response we can perform Include.
④ with in any servlet we can call forward() only once & mostly as the last stmt.	④ we can call Include() any no. of times in the servlet.
⑤ In the case of forward, the forwarded servlet having the	⑤ In the case of Include, the Included servlet (S-2) doesn't have complete control on the

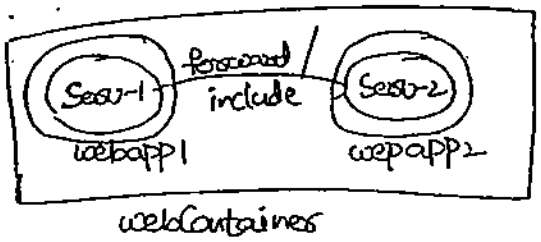
complete control on the response object. It is allowed to change the response headers.

Response object. It is not allowed to change the Response headers.

5) Forward mechanism, we can use frequently in Servlets, bcz it is associated with processing

6) Include mechanism, we can use frequently in JSP's, bcz mostly it is associated with View Part.

Foreign RequestDispatcher




```
ServletContext context = getServletContext();

ServletContext foreignContext = context.getContext("webapp2");

RequestDispatcher foreignRD = foreignContext.getRD("/test2");

        foreignRD.forward(req, resp);
```

→ Request Dispatcher mechanism will work with in the same web Container. Hence both web Application should be Deployed

in the Same Server.

→ most of the webservers won't provide support for cross Context Communication due to security Reasons. In this case we will get "Null Pointer Exception".

what is the purpose of RequestDispatcher?

How many possible ways are there for getting RD? 3

what is the Diff btw obtaining RD from Servlet Request & Servlet Context Object.

what is the Diff btw forward & Include.

what is the Diff btw forward & Send ReDirect.

what is foreign Request Dispatcher & How we can get it?

what are the Attributes added by the web Container while forwarding & Including the Request.

what is the purpose of these Attributes & explain the meaning.

4/09

Filters 3 to 4 Q

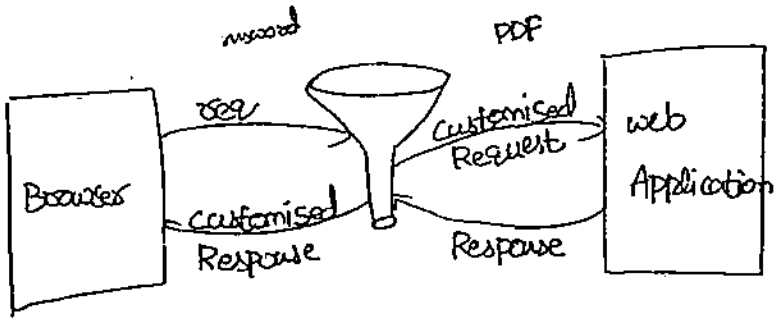
1(a) Describe the web Container Request Processing Model

(2) write & configure a filter.

(3) Create a Request & Response Wrapper for the given Design Problem.

(4) Describe how to apply a filter or wrapper...

→ Filters are used for pre-processing of Request & Post processing of Response before they reach target Resource in the web application.



→ The most common application areas of filters are logging, security checks, Altering Request Information, compressing the response, Encryption of Response, Authentication etc--

→ Filters concept has Introduced in Servlet 2.3 Version.

Filters API

→ Filters API Contains the following 3 Interfaces.

1. Filter
2. FilterConfig
3. FilterChain

Filter

→ Every Filter in Java has to implement Filter Interface either directly or Indirectly.

→ Filter Interface Defines the following 3 methods.

1. init()

```
public void init(FilterConfig config) throws SE
```

→ This method will be executed only once to perform Initialization Activities.

2. Destroy();

```
public void destroy();
```

→ This method will be executed only once to perform clean up Activities Just before taking the filter from out of Service.

3. doFilter()

```
public void doFilter(ServletRequest req, ServletResponse resp,
                     FilterChain fchain) throws
                     SE, IOE
```

→ Entire filtering logic we have to define in this method only.

→ This method will be executed for every Request.

→ By using FilterChain Object we can forward the request to the next Level. (it may be Servlet or another Filter).

FilterConfig

→ For every Filter, web Container creates a filter Config Object and hand overs it to the filter by init().

→ Filter can get its Configuration Information by using this Filter Config Object.

→ Filter Config Interface defines the following methods.

1. Public String getFilterName();
   - Returns the Logical name of the Filter Configured in web.xml by using <filter-name>
2. Public String getInitParameter(String name)
3. Public Enumeration getInitParameterNames();
4. Public ServletContext getServletContext();

Filters Chain

→ Filters Chain Object can be used for forwarding the request to the next level. (it may be filter or Servlet).

→ Filterchain Interface contains the following method.

(1) Public void doFilter(SR req, SR resp);

Demo program for Filters

DemoFilters.java

```
Public class DemoFilters implements Filter
{
  Public void init(FilterConfig config) throws SE, IOE
  {
  }
  Public void doFilter (SR req, SResp resp, Filterchain fc)
                                        throws SE, IOE
  {
      PW out = resp.getWriter();
      o.pln ("This line is added by DemoFilters before processing
                                        the request");

      fc.doFilter (req, resp);

      o.pln ("This line is added by DemoFilter after processing
                                        the request");
  }
  Public void destroy()
  {
}
```

TargetServlet.java

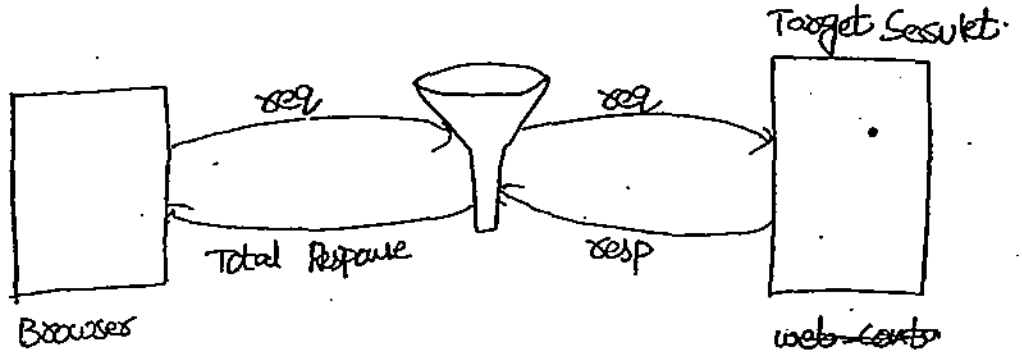
```
Public class TargetServlet extends HttpServlet
{
    P v doGET(--) ---
    {
        Pw out = resp.getWriter();
        o.pln("This is TargetServlet");
    }
}
```

web.xml

```
<web-app>
    <servlet>
        <servlet-name> TargetServlet </s-n>
        <servlet-class>     "     </s-c>
    </servlet>
    <s-m>
        <s-n> TargetServlet </s-n>
        <url-pattern> /test </url-pattern>
    </s-m>

    <filter>
        <filter-name> DemoFilter </filter-name>
        <filter-class> DemoFilter </filter-class>
    </filter>
    <filter-mapping>
        <filter-name> DemoFilter </filter-name>
        <url-pattern> /test </url-pattern>
    </filter-mapping>
```

→ should be same name.

Analysis



→ whenever we are sending Request to the Servlet, web Container checks is there any Filter Configure for this Servlet or not

→ If any Filter is configured, web Container forwards that request to the Filter instead of Servlet.

→ After Completing Filtering logic, Filter forwards the request to the Target Servlet

→ After Processing the request by Servlet, the response will be forwarded to Filter instead of Browser.

→ After Executing Filtering logic, Filter forwards Total Response to the End user.

Configuring Filter in web.xml

→ we can configure Filter by using <filter>.

```
<web-app>
    <filter>
        <filter-name>
        <filter-class>   } mandatory.
    </filter>
```

```
        <init-param>
            <param-name>  } 0
            <param-value> } to many
        </init-param>
    </filter>
</web-app>
```

→ we can map filter either for a particular url-pattern or to a particular servlet or to the whole webApplication.

Filter-mapping to a particular url-pattern

```
<filter-mapping>
        <filter-name> DemoFilter </filter-name>
        <url-pattern> /test </url-pattern>
</filter-mapping>
```

→ If the request is comming with the specified url-pattern automatically this filter will execute.

Filter-mapping to a particular servlet

```
<filter-mapping>
    <filter-name> DemoFilter </filter-name>
    <servlet-name> TargetServlet </servlet-name>
</filter-mapping>
```

Filter-mapping for entire web application.

```
<filter-mapping>
        <filter-name> Demo Filter </filter-name>
        <url-pattern> * </url-pattern>.
```

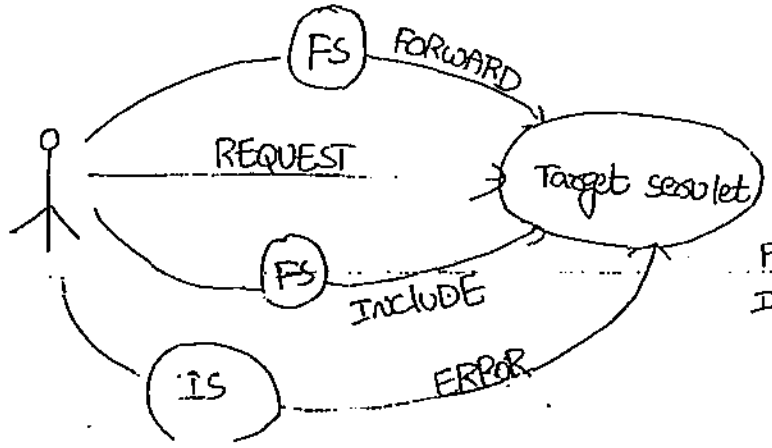
→ For any Request to the web-Application whether it is for Servlet or JSP, this filter will be executed.

→ mapping a Filter to whole web-application is Possible from Servlet 2.5 version onwards.

<dispatcher>

→ A servlet can get the Request in one of the following possibilities

1. A Request Directly from the browser. (REQUEST)
2. By RequestDispatcher Forward Call (FORWARD).
3. By RequestDispatcher Include call (INCLUDE).
4. By ErrorPage call (ERROR)



<error-page>
<error-Type>
<location>/test

FS → FirstServlet
IS → Intermediate Servlet.

→ In Servlet 2.3 V Filters Concept is Applicable only for Direct End-users Request. It is not applicable for Request Dispatcher forward, Include & Error page calls.

→ But in the Servlet 2.4 V, Sun People introduced <dispatcher> to extend Filters Concept for the remaining cases also.

→ For which type of Request the filters will be executed is decided by <dispatcher>. The allowed values for the Dispatcher Tag are

(1) REQUEST
→ Filters will execute only for Direct end users Request.
→ This is the Default value.

(2) FORWARD
– Filters will execute for RequestDispatchers Forward call.

(3) INCLUDE
– Filters will execute for RD Include call.

(4) Error
– Filters will execute for error page calls.

```
<error-page>
    <exception-type>
    <location> / test 2 <location>
<error-page>
```

1) If we want to execute Filter for Direct End-user Request & Request Dispatcher forward call. we have to configure filter Mapping as follows.

(70)

```
<filter-mapping>
    <filter-name> DemoFilter </filter-name>
    <url-pattern> /test </url-pattern>
    <dispatcher> REQUEST </dispatcher>
    <dispatcher> FORWARD </dispatcher>
```

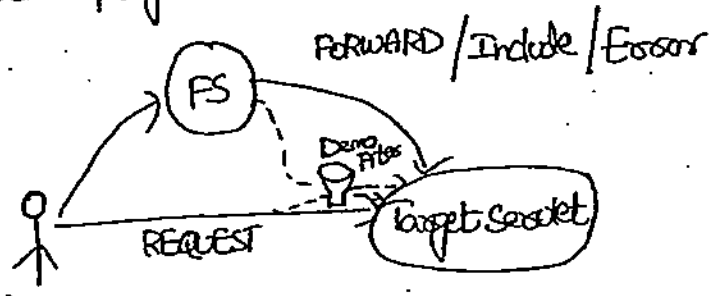
→ In this case Filter ~~only~~ won't executed for the RD Include calls & Error pages.

(71)

```
<filter-mapping>
    <filter-name>
    <url-pattern>
    <dispatcher> INCLUDE </dispatcher>
</filter-mapping>
```

→ In this case filter will execute only for RD Include call.

Demo program two <illustrates>



TargetServlet.java

```
Pw out=resp.getWriter();
o.pln("This is TargetServlet");
```

FirstServlet.java

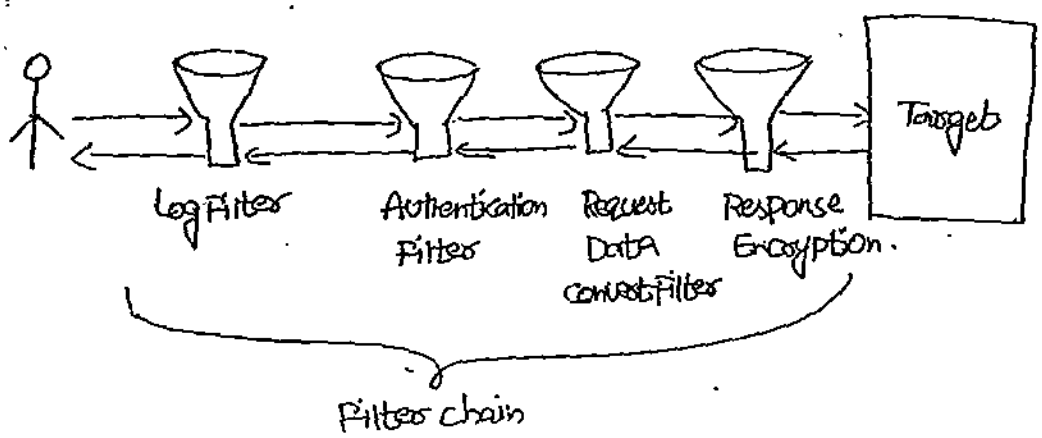
```
Public class FirstServlet extends Hs
{
    Public void doGET(--)---
    {
        Pw out=resp.getWriter();
        S.o.pln(10/0);
        RD rd=req.getRD("/test2");
        rd.forward(req,resp);
        rd.Include(req,resp);
    }
}
```

DemoFilter.java

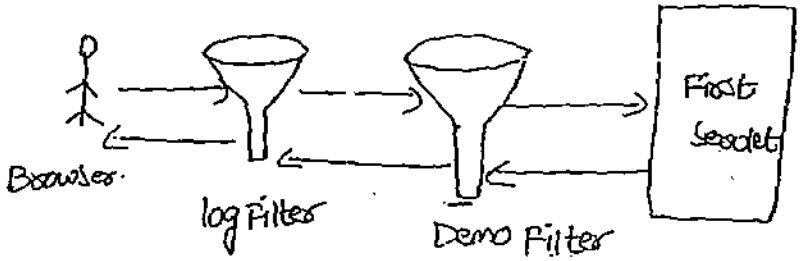
```
<web-app>

    FS          TargetServlet
     ↑            ↑
   test1        test2

<filter>
    <filter-name> DemoFilter
    <filter-class>  "
<\filter>
<filter-mapping>
        <filter-name> DemoFilter
        <servlet-name> TargetServlet
        <dispatcher> Error
        <dispatcher> REQUEST
<filter-mapping>

<error-page>
        <exception-type> java.lang.ArithmeticException
        <location> /test2 <\location>
<\error-page>

<\web-app>
```

Filter Chain

→ we can configure more than one filter for a Target Resource and all these Filters will be executed one by one and forms Filterchain.



Demo Program for the Filter chain



LogFilter.java

```
Public class LogFilter implements Filter
{
    FilterConfig conf;
    Public void init (FilterConfig conf) throws SE
    {
    } this.conf=conf;
```

```
Public void doFilter(SR req, SR resp, Filterchain fc) throws
                                                    SE, IoE
{
    PrintWriter out = resp.getWriter();
    out.pln("This line is added by LogFilter before processing the
            request");

    HSR req1 = (HSR)req;

    ServletContext context = config.getServletContext();
                    write this Info to the log File as Seperate
    Context.log("A Request is comming from "+req1.getRemoteHost();
                for"+req1.getRemoteURL() + "at"+
                req1.get new Date();

    fc.doFilter(req,resp);

    out.pln("This line is added by LogFilter after Processing the
                                                Request");
}
Public void destroy()
{
    Conf = null;
}
```

DemoFilter.java

FirstServlet.java

web.xml

```
<w-a>                                   c:\Tomcat 6.0\log\localhost.
  <filter>                                                2009-04-07.
    <f-n> LF <|f-n>                                          txt
    <f-c> LF <|f-c>
  <|f>
  <f>
    <f-n> DF <|f-n>
    <f-c> DF <|f-c>
  <|f>
  <filter-mapping>
    <filter-name> Log Filter
    <url-pattern> /test1   [or] <s-n> FS <|s-n>
                                 ① DF  execution
  <|filter-mapping>              ② LF
  <filter-mapping>
    <filter-name> DemoFilter
    <url-pattern> /test1
  <|filter-mapping> O/P

      FS          This line added by LF before Proc Req
       ↑          " " " " DF " " "
      /test1      This is TargetServlet
                  This line added by DF After Proc req
<|w-ap>           " " " " LF " " "
```

Containers Role for Ordering of Filters in Filter chain. (73)

- Identify all the filters which are configure according to url-pattern. & execute all these filters from TOP to Bottom.
- Identify all Filters which are configure according to <Servlet-name> & execute all these filters from TOP to Bottom.

8/04/09

Consider the following web.xml Declarations.

```
<filter-mapping>
        <filter-name> Filter1 </filter-name>
        <url-patter> /Recipes/* </url-pattern>
</filter-mapping>

<filter-mapping>
        <filter-name> Filter2 </filter-name>
        <servlet-name> /Recipes/Hoplist.do </servlet-name>
</filter-mapping>

<filter-mapping>

        <filter-name> Filter3 </filter-name>
        <url-pattern> /Recipes/Add/* </url-pattern>
</filter-mapping>

<filter-mapping>
        <filter-name> Filter4 </f-n>
        <servlet-name> /Recipes/modify/modify.do </s...
```

```
<filter-mapping>
        <f-n>filter5 </f-n>
        <url-pattern> / * </u-p>.
</f-m>.
```

Request URI	Order of Filters Execution.
① /Recipes/HopRepost.do	Filter1 → Filter 5
② /Recipes/Hoplist.do	① → ⑤ → ② ① Url-pattern ↓ T-B ② servlet-name ↓ T-B
③ /Hoplist.do	⑤
④ /Receips/modify/modify.do	① → ⑤ → ④
⑤ /Receips/Add/AddRecipes.do	① → ③ → ⑤

⇒ Filter & Filter mapping Tags are the Direct child Tags of web-app and we can take these tags any where with in <web-app>

```
doFilter(SR req, SR resp, fc fc)
{
}
doFilter(SR req, SR, resp)
}
```

Call back method

InLine method

Difference btw FilterChain doFilter() & Filter's doFilter method.

Filter's doFilter()	FilterChain's doFilter()
① Public void doFilter(SR req, SR resp; FC fc) throws IoE, SE	① public void doFilter(SR req, SR resp)
② For Defining Entire Filtering Logic	② For ~~Defing~~ forwarding Request to the next level (it may be servlet or filter)

③ It is call back method bcz web-Containes calls these method Automatically.

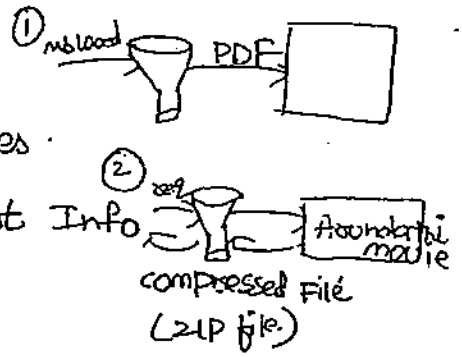
③ It is Inline method bcz programer has to call these method explicitly.

Wrappers

→ Some times it is required to alter Request & Response Information in the filters. we can achieve this by using wrapper classes.

Exam-1 - with in the filter we have to convert End users Resume from word format to PDF format.

Examp-2 - In the Filter we have to compress the Response and that compressed Response we can send to the Browser so that we can reduce Download Times.

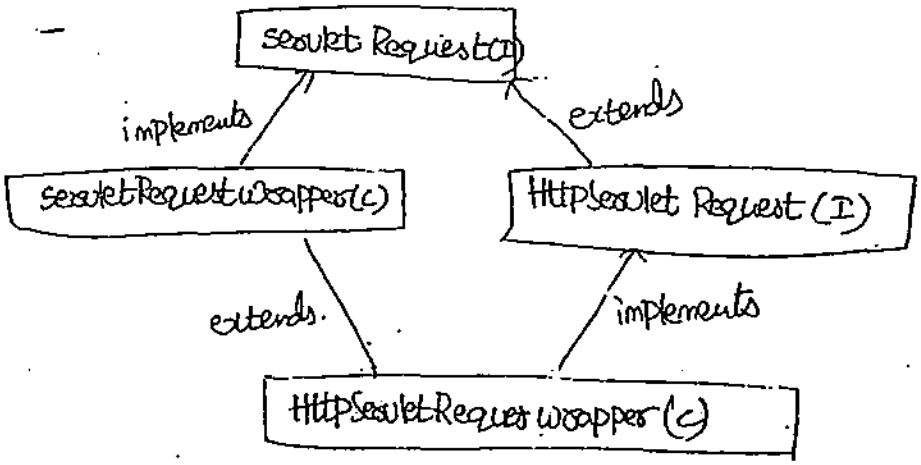


⇒ There are 2 types of wrapper classes.

① Request Wrappers — To Alter Request Info

ⓐ ServletRequestWrapper

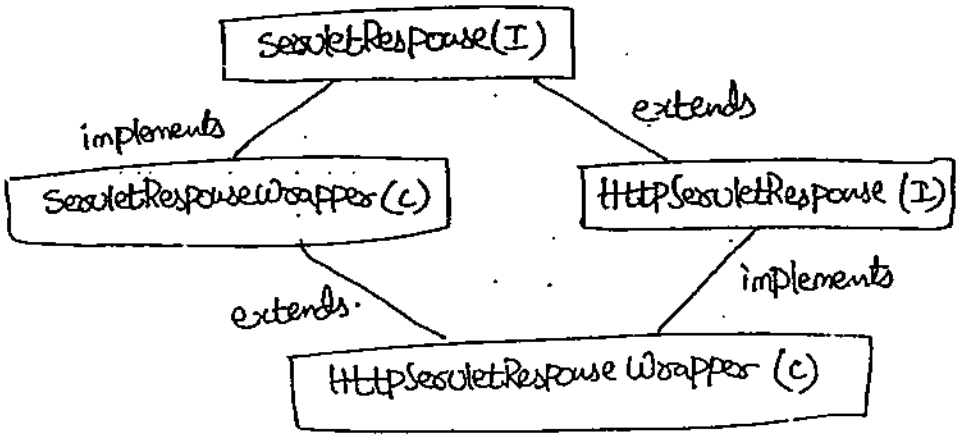
ⓑ HttpServletRequestWrapper



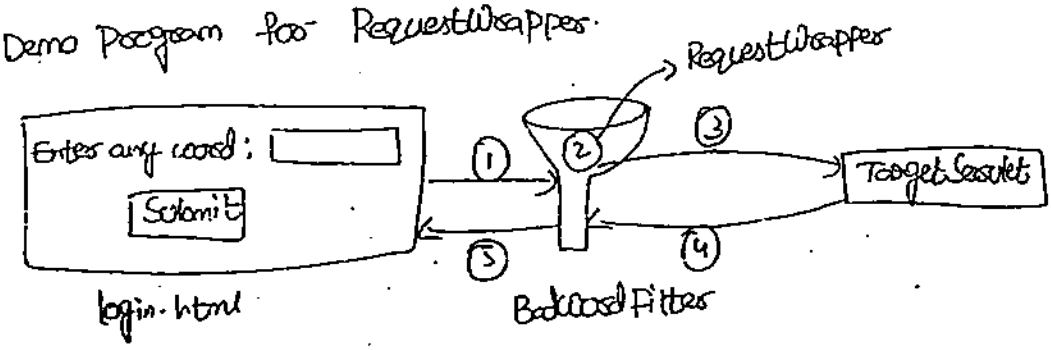
(2) Response Wrapper — To Alter Response Information.

(a) ServletResponseWrapper

(b) HttpServletResponseWrapper.



→ Demo Program for RequestWrapper.



⇒ End user Entered the word & click the submit button.

→ Filter creates a customized Request object by using wrapper class.

→ Filter forwards that customized request object to the Target Servlet instead of original Request

→ with in the Servlet if we are applying any operation on the Request object our own customized behaviour will be Reflected.

→ Target Server prepares the Response & forwards to Filter

→ Filter forwards that Response intern to the End user. (78)

AB +ve Blood Group
99491 78512 — fessoz than.

login.html

```
<html>
 <form action="/wrappers/test1">
 Enter your word : <input type=text  name=word>
 <input type=submit >
```

BadwordFilter.java

```
public class BWF implements Filter
{
   P V init(Fc config)throws SE
   {
   }

   P V doFilter( - --) --
   {
     customizedRequest  req1 = new customizedRequest ((HttpServlet
                                                        Request) req );

     fc.doFilter(req1 ,resp);
   } -

   P V destroy()
   {
   }
}
```

customized request.java

```
Public class CR extends HttpServletRequestWrapper
{
   Public CR( HSR req)
   {
        super(req);
   }
   Public String getParameter( String word)
   {
     String word1 = super.getParameter("word");
   if(word1.equals("Java") || word1.equals("SCJP"))
     return "Sleep"
   else
     return word;
   }
```

TestServlet.java

```
P class TS extends HS
{
   P V doGET( - -)
   {
      Pw out = resp getWriter();
      String word = req.getParameter("word");
      o.pln("your Entered word is" + word);
   }
}
```

web.xml

<w-a>

TargetServlet — /test

BadwordFilter — /test

</w-a>

⇒ For wrappers not required to perform any configuration in web.xml.

Note

(web-fn)

1. Filter Object is created by the web container, for this it is always calling Public no-arg constructor. Hence every filter class should compulsory contain Public no argument Constructor

2. usage of Filter is nothing but following Intercepting Filter Design Pattern. Hence it is recommended to use Filter Concept.

3. usage of wrappers is nothing but following Decorator Design pattern.

UNIT-4

8 to 10 Ques

Session Management

b) (1) Given a Senario, Describe the Session API

(2) Explain the Process of creating a Session object.

(3) what are Various Different mechanisms to Invalidate a Session.

→ The basic limitation of HTTP is, it is stateless Protocol. i.e it is unable to remember client State across multiple Request.

→ Every Request to the Server is considered as a new Request. Hence some mechanism is required to remember client Information across multiple Request. This mechanism is nothing but Session Management Mechanism.

→ The following are Various Session Mangement mechanisms.

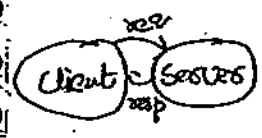
(1) Session API

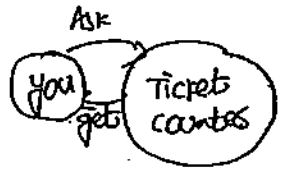
(2) cookies

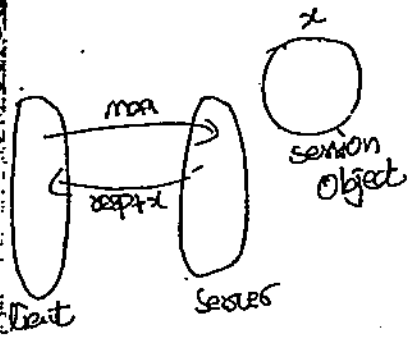
(3) url-Rewriting

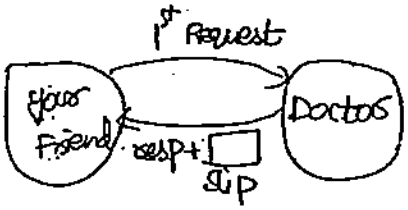
(4) Hiden Form Field — Not official mechanism from sun, it is Just Programmers trick to remember client Information.

Creation of Session Object







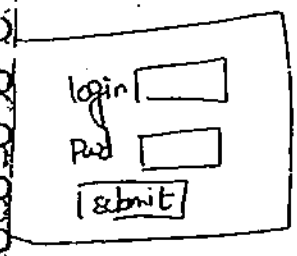


→ Some. after 2 days Along with your slip

→ This is client side Session mgt.

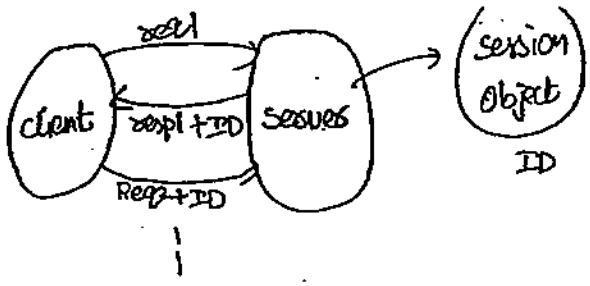
→ This Type is cookies.

At server side, Session Object



whenever client sends the 1st Request, if the server wants to Remember Clients Information then it will create a Session Object & sends the corresponding Session ID to the Browser as the part of 1st Resp.

Browser sends that Session ID as the part of consequitive request, By Accessing that Session ID & the corresponding Sess Obj, server able to remember client Info across multiple Request.



:HHTP Servlet Request defines the following methods for the Creation of Session Object.

① public HttpSession getSession();

Ex: HttpSession Session = req.getSession();

> If the request Associated with Session Object already then existing session Object will be returned.

> If the request doesn't associated with any Session, this method will create a new Session Object & returns it.

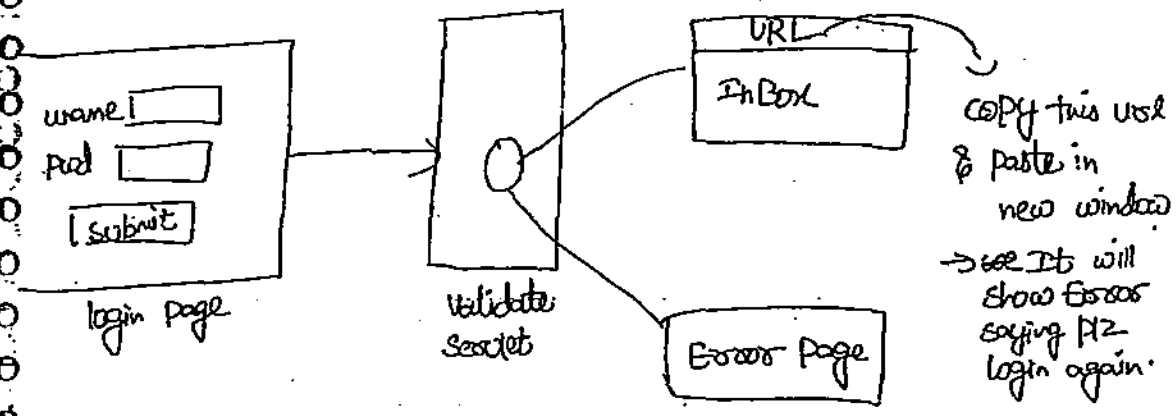
> There is a guarantee that this method will always return session Object. It may be newly created or existing Object.

② public HttpSession getSession(Boolean b);

→ If the Argument is true then this method simple acts as getSession();

→ If the Argument is false then this method first checks whether the request Associated with any Session or not. If the request already associated with Session then this method returns Existing Session Object.

If the request doesn't associate with any session then this method returns "NULL" without creating any new Session Object.



To Access Inbox Page compulsory the request should contain session. If the request doesn't associated with any session we can't create new session Object & we have to forward that request to the login page. In this case we have to use getSession(Boolean b).

which of the following 2 stmts are equal.

(1) Session = req.getSession();

(2) session = req.getSession(false);

(3) Session = req.getSession(true);

(4) Session = Context.getSession(true);

Invalidating a session.

⇒ we can invalidate a session by using the following 2 ways.

(1) By invalidate()

(2) By Time Out Mechanism.

invalidate()

⇒ HttpSession Contains this method to invalidate a Session.

Public void invalidate()

Ex: whenever we are clicking [logout] internally this method will be executed.

session.invalidate();

TimeOut Mechanism.

Q If we are not performing any operation on the Session Object for a pre-defined amount of time then automatically session will be expired. This pre-defined amount of time is called Session-TimeOut.

⇒ we can configure Session Timeout either Server Level (or) web-Application Level or a particular Session Object Level.

1) Automatic support from the web server

→ most of the web servers provide default support for session-timeout. mostly it is 30 minutes.

→ we are allowed to change this server-level session time Out value based on our requirement.

2) Configuring SessionTimeOut at Web-Application Level.

→ we can configure Session TimeOut for entire web-Application in web.xml has follows.

```
<web-app>
    <session-config>
        <session-timeout> 10 </session-timeout>
    </session-config>
    :
</web-app>
```

In minutes

→ <session-config> is the Direct child Tag of web-app & we can ~~configure~~ see place anywhere with in web-app.

→ The unit to the Session-timeOut value is minutes

→ 0 or '-ve' value Indicates that session never expires.

→ This Session-timeOut is applicable for all the Sessions which are created in that web-application.

3) Individual setting SessionTimeOut for a Particular Session Object.

→ we can set SessionTimeOut for a particular Session Object by using the following method of HttpSession.

Public void setMaxInactiveInterval (int seconds)

=> The Argument must be in seconds.

-> -ve value indicates the session never expires

-> '0' indicates SessionExpires immediately.

-> This sessionTimeOut is applicable only for a particular sessionObject on which this method has called.

Comparision btw 2 sessionTimeOut mechanisms.

Property	<session-timeout>	SetMaxInactiveInterval()
(1) Scope	It is applicable for all the sessions which are created in that web-application.	It is applicable only for a particular Session Object on which we called this method.
(2) units	minutes	seconds.
(3) 0-value	Session never Expires	Expires immediately.
(4) -ve value	Session never Expires	Session never Expires.

Important methods of HttpSession

1) public boolean isNew()
- To check whether the SessionObject is newly created or not

2) public void invalidate()
- To expire a session forcefully.

3) public void setMaxInactiveInterval (int Sec)
- To set Session-timeOut for a particular Session Object.

4) public int getMaxInactiveInterval();
- Returns Session-timeOut value in Seconds.

5) public String getID();
- Returns the Session-ID.

6) public long getCreationTime()
- Returns the time when the Session was created in millSeconds since Jan 1st 1970.
- If we are passing this long value to the Date Constructor we will get exact date & Time.

1970 Jan 1st
↕ in millsec
may 1st 7:30Am.

(Date d = new Date (l));

7) public long getLastAccessedTime();
- Returns the time when the client Accessed recently the Session Object in milliseconds.

8) public ServletContext getServletContext();
- Returns the ServletContext Object to which this session belongs.

→ once session expires, we are not allowed to call above methods violation leads to illegal state Exception.

→ But this rule is not applicable for the following 3 methods.

① getCreationTime()
② getLastAccessedTime()
③ getServletContext()

```
HttpSession s = req.getSession();
s.invalidate();
s.open(s.getCreateTime());
```

→ HttpSession Defines the following methods to perform Attribute management in Session Scope.

① public void setAttribute(String name, Object value);
② public Object getAttribute(String name);
③ public Enumeration getAttributeNames();
④ public void removeAttribute(String name);

Demo program for Session Management by using Session API.

login.html

```
<html>
<body>
<form action="/session1/test1">
<h1> Enter session Information </h1>

Enter Name : <input type=text name=uname>
Enter Value : <input type =text name =value>
  <input type= submit>

</form>
<a href ="/session1/test2"> Session Information </a>

</body> </html>
```

Session Servlet1.java

```
public class SS extends Hs
{
  P V doGET (---)
  {
      PrintWriter out = resp.getWriter();
      HttpSession session = req.getSession();
      if (session.isNew())
      {
        o.pin("New session get created with Session ID:"+
                                          Session.getId());
      }
```

```
else
{
  o.pln("Existing Session only using with SessionID:" +
                                  session.getID);
}

String name = req.getParameter("uname");
String value = req.getParameter("uvalue");
Session.setAttribute(name, value);
Session.setMaxInactiveInterval(120);
RD rd = req.getRD("login.html");
  rd.include(req, resp);
}
}
```

SessionServlet2.java

```
Public class SS extends HS
{
  P V doGET(--)
  {
    PW out = resp.getWriter();
    HttpSession Session = req.getSession(false);
    if (session == null)
    {
      o.pln("No Session Information is available");
    }
```

```
else
{
    Enumeration e=hs.getAttributeNames();
    o.pln(" <table> <tr><tr><td> SessionInformationNames
        <th> SessionInformation Values <

    while(e.hasMoreElements())
    {
        String name =
        String value=(String)session.getAttribute(name);
        o.pln(name +"----" + value);
    }

    o.pln("<table">);

    long l1=Session.getCreationTime();
    long l2 = Session.getlastAccessedTime();
    int l3 = Session.getMaxInactiveInterval();

    o.pln(" <h1> creation time is :" + new Date(l1) +"</h1>");
    o.pln(  "                    "  + new Date(l2)  "  );
    o.pln(" maxInactiveInterval  + l3
```

web.xml

```
<web-app>
  <session-config>
    <session-timeout> 3 </s-to>
  </session-config>
```

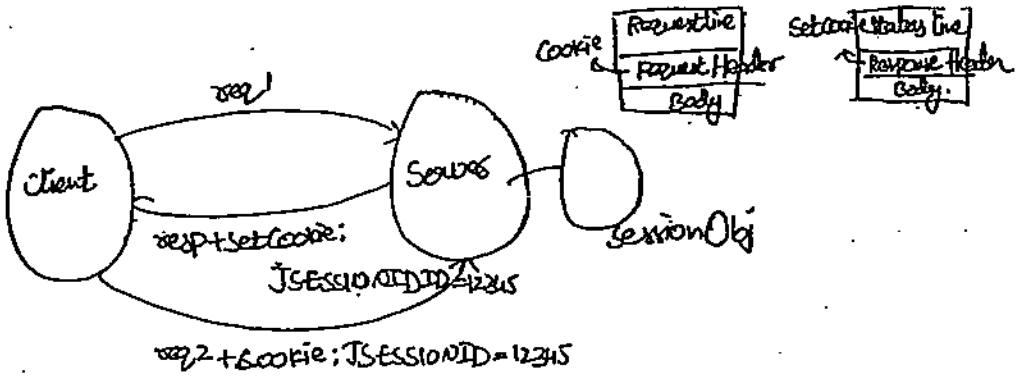
Not using
↑ Defaults 1800 → 30 minutes will be displayed

SS1 ↑ /test1

SS2 ↑ /test2

</web-app>

How the SessionID Exchanging btw client & server



→ whenever Browser sends a Request, Server creates a Session Object.
If

→ The corresponding Session-ID Server will send as the part of Response. For this Server uses SetCookie Response Header.
The Browser will retrive that Session-ID & will send with every consecutive request. For this the browser uses cookie request Header.

→ a href = "/session/test3"

RequestHeaderServlet → for checking the SessionID.

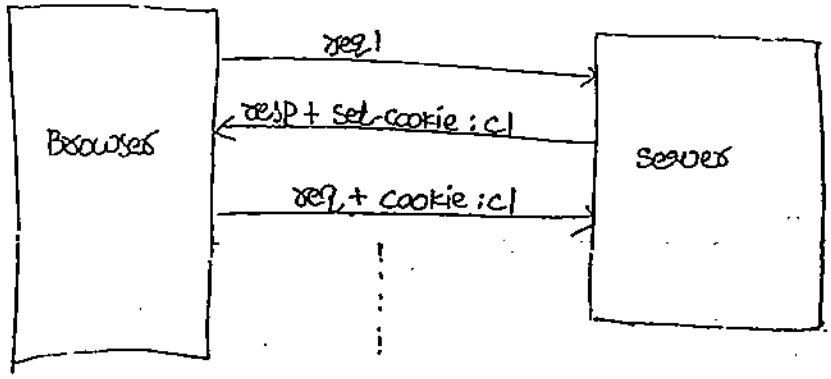
1/09/09 **Note** :- If the required session info is very less then creating a seperate session Object & maintaining that Object at server side is not recommended bcz it impacts the performance of the system.

→ To resolve this we should go for cookie concept where required session Info is maintained at client side & server is not responsible to maintain session Info.

Cookie

→ Cookie is a small amount of Information (key-value pair) which is created by server & send to the browser. The browser stores that cookies in the local file system & sends with every consequitive request to the server. By Accessing that cookie server able to remember client Information.

→ Server sends cookies by using setCookie Response Header & Browser resends those cookies by using cookie Request Header. This is exactly similar to exchanging Session-ID btw client & server.



→ we can create a cookie object by using cookie class constructor.

```
Cookie c=new Cookie(String name, String value);
```

→ After creating cookie object we have to add that object to the response by using addCookie();

```
resp.addCookie(c);
```

→ At server side we can retrive cookies by using getCookies()

```
cookie[] c = req.getCookies();
```

→ If the request doesn't associated with any cookies this method returns "null".

→ Important methods of cookie class.

1. Public String getName();
   -- Returns name of the cookie
2. Public String getValue();
3. Public int getMaxAge();
   - Returns the maximum age of the cookie in seconds.
4. Public void setMaxAge(int seconds);

→ Setting MaxAge as "-1" cookies will be disabled automatically whenever Browser window closed.

Gmail

Keep login for 15 days.

c.setMaxAge(15×24×60×60);

Cookie.html

```
<form action="/Session1/test1">
<h1>Enter cookie information</h1>
Name: <input type=text name=uname>
Value: <input type=text name=value>
  <input type="submit">
</form>
<a href="Session1/test2">Cookie Information</a>
```

CookieDemoServlet-1.java

```
public class CDS1 extends HS
{
  PW doGET(--)---
  {
      PW out=resp.getWriter();
      String name=req.getParameter("uname");
      String value=req.getParameter("value");
      Cookie c=new Cookie(name, value);
      c.setMaxAge(300);
```

```
    res.addCookie(c);
    o.pln("Cookie Added Successfully");
    RD rd = req.getRD("cookie.html");
      rd.include(req, resp);
  }
}
```

CookieDemoServlet2.java

```
public class CDS2 extends HS
{
    P v doGET(--) ---
    {
        resp.setContentType("text/html");
        PW out = resp.getWriter();
        Cookie[] c = req.getCookies();
      if(c==null)
      {
        o.pln("No Cookies are Associated with the request");
      }
      else
      {
        for(c1 : c)
        {
          String name = c1.getName();
          String value = c1.getValue();
```

```
o.pln (name +" - -- "+value);
}
}
```

7/05/09

Advantages of cookies

1. Very Easy to implement.
2. Persist across Browsers shutDowns also.
3. All web servers provides automatic support for cookies.

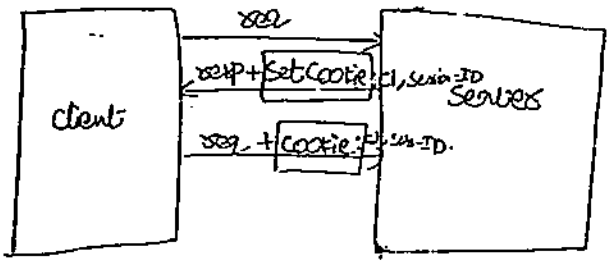
Disadvantages of cookies

1. cookies can be Enable or Disable at client side to meet security Constraints.

→ If the cookies are Disabled then session management by using cookies is not possible.

→ The no. of cookies supported by any Browser is always fixed.

→ The size of cookies is also fixed, hence we can't store huge amount of Information by using cookies.



→ Session ID & cookies are exchanging btw client & server by using Set-cookie Response Header & Cookie Request Header.

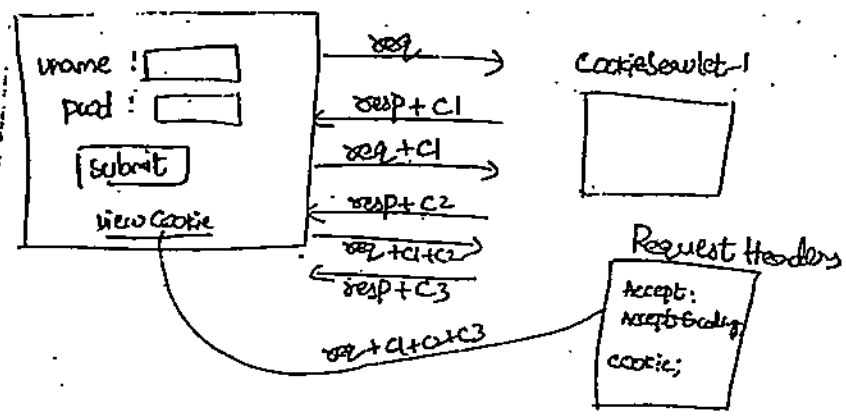
-> If the cookies are Disabled at client side then the Browser is unable to see setcookie Response Header. Hence Browser won't get any cookies or session-ID sent by server.

-> If the cookies are Disabled at client-side then the Browser unable to send cookie Request Header. Hence server won't get any cookies or Session-ID from the Request & Every Request is treated as New Request. Hence Entire Session management Fails.

-> To Resolve this problem Sun People Introduced the most Powerful & painful Technique — URL-Rewriting.

Comparision btw Session API & cookies

Session-API	Cookies
1) Session Information will be maintained at Servers side	1) Session Info will be maintained at client side.
2) Best suitable if we want to store huge amount of Info	2) Best suitable if we want to store very less amount of Info.
3) The Session Information can be any type i.e need not be String Info.	3) The Session Info should be String Info (both cookie name, value should be string objects only).
4) Security is High & won't create any into traffic Problems.	4) Security is less & may arise into traffic problems.



Note

→ If we are setting maxAge to the cookie such type of cookies are called Persistant cookies. & these are available in the local file system.

→ If we are not setting any maxAge such type of cookies are called Temporary cookies. once the Browser closes automatically the cookies will be expired. These cookies will be stored in the Browsers cache & not visible in the local file system.

URL-Rewriting

→ URLs can be Re-Written or Encoded to include Session Information. This Technique is called URL-Rewriting.

Ex: URL;JSESSIONID=12345

i.e URL Rewriting = URL + Session Info

→ HttpServletResponse Defines the following methods to append Session-ID to the url.

① Public String encodeURL(String URL);
- Returns url by appending JSessionID.

2) public String encodeRedirectURL (String url)
- Returns url by appending session-ID.
- This can be used as an Argument to send-Redirect().

⇒ The above 2 methods will append JSession-ID if & only if cookies are Disable.

⇒ If the cookies are Enable, these methods returns the same url's with out appending JSession-ID.

⇒ At server side we can Identify whether SessionID is coming as the part of url or from the cookie Request Header by using the following methods of HttpServletRequest.

1. boolean isRequestedSessionIdFromURL();
2. boolean isRequestedSessionIdFromCookie();

By using these methods we can identify underlying session management Technique.

Advantages of URL-Rewriting

- There is no chance of Disabling URL-Rewriting Technique. Hence it is universally supported.

Limitations of URL-Rewriting

- It is very Difficult to Rewrite all urls to append Session Information.
- URL-Rewriting works only for Dynamic Documents.
- It is common trend to use both cookies & URL-Rewriting for session management.
- Cookies will work if they are Enable otherwise immediately. Url-Rewriting will work. so that we can maintain client state always across multiple requests.

Demo program for URL-Rewriting

login.html

```
<html>
<body>
<form action="session3/test1">
<h1> Enter session Information
name : <input type="text" name="uname">
<input type="submit"/> </h1>
</form>
</body>
</html>
```

Session servlet1.java → Retrieve the name & adding to the second servlet.

```
Public class SS1 extends HS
{
   P V doGET(--)
   {
      PW out = resp.getWriter();
   String name = req.getParameter("uname");
   o.pln("<h1> welcome to Durga for SCJCD </h1>");
   o.pln("<a href=" /session3/test2/name + '+name+' >
         click here to get as name </a>");
   }
}
```

SessionServlet2.java — Accessing that name & Point the name

```
Public class SS2 extends HS
{
   P V doGET( -- ) --
   {
      PW out = resp.getWriter();
      String name1 = req.getParameter("name");
      out.pln("<h1> hi +"name+" Good Morning </h1>");
   }
}
```

blank space

Listeners 2 to 3 Ques

1)) Describe the web Container Event Life-cycle model for the Request, Session & web-Application.

2) Create & Configure Listener class for each scope.

3) Create & Configure Attribute Listeners for each scope.

) For the given Scenario Identify Proper Attribute listener.

> In the web-Application there may be a chance of occuring several Events like

1. Request Object Creation
2. Session Object Destruction
3. Context Object Creation
4. Attribute addition in Request Scope
5. Removal of Attribute in SessionScope.

> we can Configure Listeners classes to listen these Events & they can do appropriate things whenever that Event occurs.

> All the Listeners are Divided in to 3 Groups

1. Request Listeners – These listen the Events Related to Request. There are 2 types of Request Listeners.
   1. Servlet Request Listeners.
   2. Servlet Request Attribute Listeners.

2) Session Listeners – Listens the events related to session.

These are 4 types of Session Listeners. (90)

1. HttpSessionListener.
2. HttpSessionAttributeListener
3. HttpSessionBindingListener
4. HttpSessionActivationListener.

Context Listeners – Listens the Events Related to Context.

These are 2 types of Context Listeners

1. ServletContextListener
2. ServletContextAttributeListener.

Servlet Request Listener

This Listener listens the Life cycle Events of Request Object like creation & Destruction.

This interface Defines the following 2 methods.

1. Public void requestInitialized(ServletRequestEvent e)
   – This method will be executed automatically by the web Container at the time of Request Object creation. i.e Just before Starting Service method.

2. Public void requestDestroyed(ServletRequestEvent e)
   – This method will be executed automatically by the web Container at the time of Request Object Destruction i.e Just After completing Service method.

Listeners

- Request listeners
  - Servlet Request Listener
  - Servlet Request Attribute Listener
- Context Listeners
  - ServletContext Listener
  - Servlet Context Attribute Listener
- Session Listeners
  - HttpSession Listener
  - HttpSession Attribute Listene
  - HttpSession Binding Listener
  - HttpSession Activation Listener

- This class contains the following methods to return Request & Context Objects.

① Public ServletRequest getServletRequest();

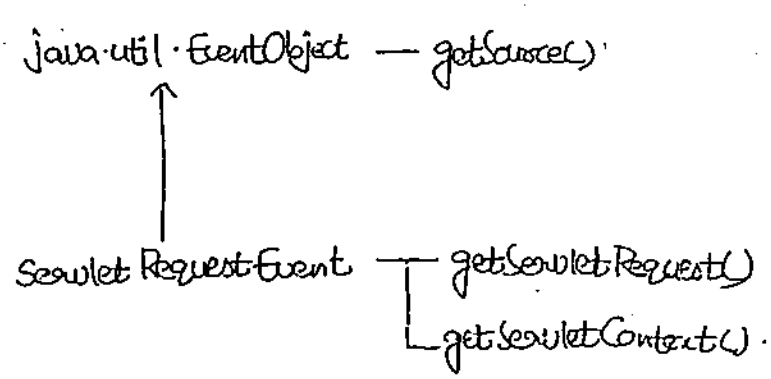
② Public ServletContext getServletContext();

→ ServletRequestEvent is the child class of java.util.EventObject.

→ EventObject class contains one method getSource();

Public Object getSource();

- It returns the source which causes the Event. In this case web-Application is the source of Event & hence we will get Servlet Context object.



Demo Program for ServletRequestListener

→ This listener maintains counter variable to represent hit count of the web-Application.

ServletRequestDemoListener.java

```
Public class SRDL implements ServletRequestListener
{
    Static int count=0;

  Public void requestInitialized (SREvent e)
  {
        count++;

      S.o.pln (" A new Request came at:"+new Date());
  }
  public void requestDestroyed (SREvent e)
  {
    S.o.pln (" A Request Destroyed at:"+new Date());
  }
}
```

5\09 Configuring a Listener in web.xml

→ we can configure a listener in web.xml by using <listener>.

```
<web-app>
    <listener>
        <listener-class> SRDL </listener-class>
    </listener>
</web-app>
```

-> <listener> is the direct child tag of <web-app> & we can place anywhere with in <web-app>

-> we can configure more than one listener of same type, the order of execution of these listeners is depends on the order of <listener> in web.xml.

First Servlet.java

```
Public class FirstServlet extends HttpServlet
{
    Public void doGET(--)---
    {
        PrintWriter out = resp.getWriter();
        o.pln("The hit count of the webApplication is:" +
                                        SRDL.count);
    }
}
```

static void so we can call by class Name

=> web container is responsible for the creation of listener class obj. For this web-container always calls public no-Argument constructor. Hence every listener class should compulsory contain public no Argument Constructor.

=> Instantiation of the listener should happen at either server start-up or At the time of Application Deployment.

Servlet Request Attribute Listener

→ This Listener listens the Events Related to Request Scoped Attributes like Attribute Addition, Attribute Replacement & Attribute Removal.

→ This interface Defines the following 3 methods.

1. Public void attributeAdded (ServletRequestAttributeEvent e);
   - This method will execute automatically by the web-Container whenever we are adding an Attribute in the Request Scope.

2. Public void attributeRemoved (SRAE e);

3. Public void attributeReplaced (SRAE e);

ServletRequestAttributeEvent

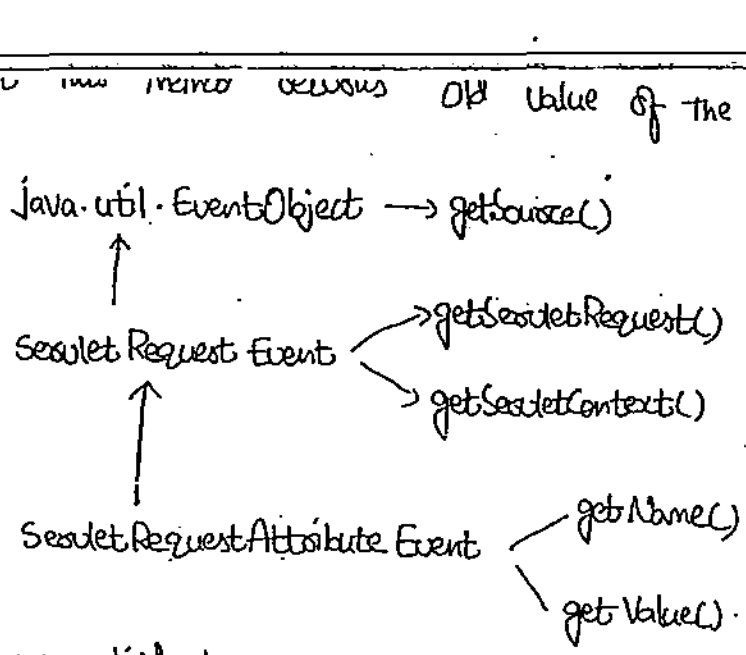
→ This class contains the following 2 methods.

1. Public String getName();
   - Returns the Name of the Attribute which is added (or) Replaced (or) Removed in the Request scope.

2. Public Object getValue();
   - Returns the Value of the Attribute which is added (or) Replaced (or) Removed in the Request scope.

⇒ In the case of Attribute Addition & Removal this method returns the corresponding Attribute Value. But in the case

of replacement has method returns old value of the Attribute.



⇒ SRAE is the child class of SRE

Demo Program

```
Public class SRADL implements SRAL
{
  Public void attributeAdded (ServletRequestAttributeEvent e)
  {
    S.o.pln (e.getName() + "-- Attribute Added");
  }
  Public void attributeRemoved (SRAE e)
  {
    S.o.pln (e.getName() + "--Attribute Removed");
  }
  Public void attributeReplaced (SRAE e)
  {
    S.o.pln (e.getName() + '-- Attribute Replaced');
  }
}
```

```
<web-app>
    <listener>
        <listener-class> SRADL </listener-class>
    </listener>
    FirstServlet
        ↑
        /test
</web-app>
```

FirstServlet.java

```
public class FS extends HttpServlet
{
    P V doGET( -- ) ---
    {
        PW out = req.getWriter();
        o.pln("This is SRADL");
        req.setAttribute("att1", "SCJP");
        req.setAttribute("att2", "SCWCD");
        req.setAttribute("att1", "SCBCD");
        req.removeAttribute("att1");
    }
}
```

ServletContext Listeners

→ This listeners listens the life cycle Events of Servlet Context like creation & Destruction.

→ This interface Defines the following 2 methods.

① public void contextInitialized (ServletContextEvent e)

– This method will be executed automatically by the web Containes at the time of Context Object creation. i.e At the time of Application Deployment.

② public void contextDestroyed (ServletContextEvent e)

– This method will be executed automatically by the web Container at the time of Context Object destruction i.e At the time of Application unDeployment.

5/09 Servlet Context Event.

→ It is the child class of java.util.Event Object & contains only one method getServletContext();

```
Public ServletContext getServletContext();
```

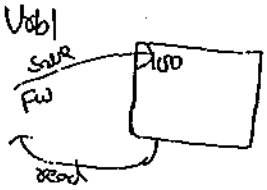
java.util.EventObject – getSource();

↑

ServletContextEvent – getServletContext();

Persistant Counter Vbl

Static int Count



To save the count value permanently (i.e At the time Restart, Deployment, unDeployment, Reload the value will gone

ServletContextAttributeListener

→ This listener listens the Events Related to Context scoped Attributes. like Addition, Removal, Replacement.

→ This Interface Defines the following 3 methods.

1) Public void AttributeAdded (ServletContextAttributeEvent e);
2) public void attributeRemoved (SCAE e);
3) Public void attributeReplaced (SCAE e);

Servlet Context Attribute Event

→ It is the child class of ServletContextEvent.

→ It contains the following 2 method.

1) Public String getName();
2) Public Object getValue();

```
java.util.EventObject  — getSource();
        ↑
ServletContextEvent  → getServletContext();
        ↑
ServletContextAttributeEvent  — getName();
                              — getValue();
```

HttpSessionListener

→ This listener listens the lifecycle Events of HttpSession Object like creation & Destruction.

→ This interface Defines the following 2 methods.

1. Public void sessionCreated(HttpSessionEvent e);
2. Public void sessionDestroyed(HttpSessionEvent e);

HttpSessionEvent

→ This class Defines only 1 method getSession();

```
Public HttpSession getSession();
```

java.util.EventObject — getSource();

↑

HttpSessionEvent — getSession();

Demo Program to Display the number of users currently online. (i.e to Display the no. of active Sessions at the Server Side).

SessionCounter.java

```
Public class SessionCounter implements HttpSessionListener
{
    Static int Count=0;

    Public void sessionCreated (HttpSessionEvent e)
    {
        count++;
        S.o.p("A new session created with id :"+e.getSession().getId());
    }
```

```
    public void sessionDestroyed (HSE e)
    {
        count--
        s.out.pln ("An existing session Destroyed with Id:" +
                                e.getSession().getId());
    }
}
```

FirstServlet.java

```
public class FS extends Hs
{
    P v doGET (---) ---
    {
        PW out = resp.getWriter();
        HttpSession session = req.getSession();
        session.setMaxInactiveInterval (120);   session Obj expires after 2 minutes
        o.pln ("The no. of users online is:" + SessionCounter.count);
    }
}
```

web.xml

```
<web-app>
    <listener>
        <listener-class> SessionCounter </listener-class>
    </listener>
</web-app>
```

FS
↑
test

HttpSessionAttributeListener

→ This listener listens the Events Related to session Scoped Attributes. like Addition | Removal | Replacement.

→ This interface Defines the following 3 methods.

1. public void attributeAdded (HttpSessionBindingEvent e)
2. public void attributeRemoved (HSBE e)
3. public void attributeReplaced (HSBE e)

→ There is no Event class Named with HttpSessionAttributeEvent For this Equivalent Event class is HttpSessionBindingEvent.

HttpSessionBindingEvent

→ It is the child class of HttpSessionEvent & contains 2 methods.

1. public String getName();
2. public Object getValue();

```
java.util.EventObject  — getSource();
        ↑
HttpSessionEvent  — getSession()
        ↑
HttpSessionBindingEvent  ┬ getName()
                         └ getValue()
```

05/01

HttpSession Binding Listener

→ whenever a particular type of Object Adding / Removing / Replacing in Session Scope. If we want to perform certain activities then we should go for HttP Session Binding listener.

→ This listeners list defines the following 2 methods.

① Public void valueBound (HttpSessionBindingEvent e)
→ Adding an Attribute

② Public void valueUnbound (HSBE e)
→ Remove an Attribute

For Replacement
Both will be executed
① Value Bound()
↓
② ValueUnbound()

Note

→ when ever It is not required to configure HttpSessionBinding listener in web.xml. whenever we are adding an Attribute the web Container will check the corresponding class implements HttpSessionBinding Listener or not. If it implements, the corresponding method will be executed Automatically.

→ If both Attribute & Binding listeners are configured then Binding Listener will be executed first followed by Attribute Listener.

Attribute Listener (attributeAdded())
← Session.setAttribute("attr", "sdp")

HSAL → General Attribute

← Session.setAttribute("attr2", new DOG());

HSBL → DOG
Value Bound()
attributeAdded();

```
Public class FS extends HS
{
    Public void doGET(--)---
    {
        PW out = resp.getWriter();
        HttpSession session = req.getSession();
        session.setAttribute("a1", "SCJP");-attributeAdded()
            "       "       ("a1", "SCWCD");-attributeReplaced()
            "       "       ("a2", new Dog());-valueBound()
                                                attributeAdded()
            "       "       ("a3", new Dog());-valueBound()
                                                attributeAdded()
            "       "       ("a2", newDog());-valueBound()
                                                valueUnBound()
                                                attributeReplaced()
        session.removeAttribute("a3");
        O.pln("<h1> This is SessionBindingListener Demo </h1>");
    }
}
```

See on Server Console

Session.Attribute listener.java (For Attribute Listener)

```
Public class SAL implements HSAL -
{
    Public void attributeAdded(HttpSessionBindingEvent e)
    {
        S.o.pln("Attribute Added");
    }
    Public void attributeRemoved(HSBE e)
    {
        S.o.pln("Attribute Removed");
    }
```

```
    public void attributeReplaced (HSBE e)
    {
        s.o.pln ("Attribute Replaced");
    }
}
```

Dog.java (For Binding listener)

```
public class Dog implements HttpSessionBindingListener
{
    public void valueBound (HttpSessionBindingEvent e)
    {
    s.o.pln("Dog object has added to session scope");
    }
    public void valueUnBound (HSBE e)
    {
        s.o.pln("Dog objects has removed from SessionScope");
    }
}
```

web.xml

```
<w-a>
    <listener>
        <listener-class>SessionAttributeListener</l-c>
    <listener>
    FirstServlet
        ↑
        /test.
</w-a>
```

→ It is not Required to configure HTTP sessionBinding Listener in web.xml.

s.setAttribute("a2", new Dog());

⇒ webcontainer automatically checks whether the Dog class implements HttpSessionBindingListener or not.

HttpSessionActivationListener

→ If a web Application is distributed across several JVM's such type of web Application is called Distributed web Applications.

→ The main Advantages of Distributed web Application are.

(1) By Load Balancing we can improve Performance of Application.

(2) By Handling Fail over Situations, we can keep our web Application Robust. (Fail of clients are less).

→ In the Distributed web Application the Session Object will be migrated from one JVM to another.

→ when ever a session Object is migrated from one JVM to another JVM, compulsory the corresponding Session Attributes should be migrated across the network. ~~Here~~ Hence every Session Attribute compulsory should be Serializable.

→ At the time of Session Object migration if we want to perform certain Activities then we should go for

HttpSessionActivationListener.

→ This interface Defines the following 2 methods.

① Public void sessionWillPassivate (HttpSessionEvent e)

– This method is called on each implementing Object Bound to the Session Just before Serialization

② Public void SessionDidActivate (HttpSessionEvent e)

– This is called on each implementing Object Bound to the Session Just after Deserialization.

Web Security (6 to 8 Ques)

1. Terminology
2. Types of Authentication
3. web.xml Declarations
4. Demo program → Basic, → Form
5. Programatic Security.

6) Based on Servlet Specification compare & contrast the following security mechanisms.

1. Authentication
2. Authorization
3. Data Integrety
4. Confidentiality.

Authentication

→ It is the Process of Validating the user. usually we can implement Authentication by using UserName & Password combination.

Ex:- Providing UserName & Pwd to Login in to the Bank site is called Authentication.

Authorisation

→ It is the Process of validating Access Permissions of a user i.e The Process of checking whether the user is allow to Access particular Resource or not.

→ After Authentication we have to Perform Authorisation.

→ usually we can implement Authorization by using AccessControl List (ACL)

Ex:- we are not allow to Access others Account Info even though we are member of the Bank. i.e. we are not authorized to Access others Account Info.

Data Integrity

→ It is the Process of Ensuring That, Data should not be changed in Trans (Transformation) from client to Server

→ we can implement Data Integrity by using Secure Socket Layer (SSL)

Ex:- If we are Sending a Request to transfer 1000 Rs from our Account to other, the bank Should get Transfer

Request for 100rs but not for 5000Rs

Confidentiality (Data Privacy)

→ It is the Process of Ensuring that no one except intended users is able to understand our Information.

→ we can achieve Confidentiality by using Encryption Mechanisms

Difference btw Authorization & Confidentiality.

→ Authorization prevents the Information from reaching unintended users in the First place.

→ where as Confidentiality Ensures that, Even the Information falls in the wrong hands it remains unreadable.

Example :- Beer Site web Application.

(1) members Should have username & pwd to Access site — Authentication

(2) Only Premium users can get 10% special Discount - Authorization.

(3) when a user Places an order Some Sort of confirmation is required — Data Integrity.

(4) when a customer makes a purchase no one allow to use our credit card Information — Confidentiality.

Types of Authentication

b) Describe the Authentication Types & Explain their process for the given Design Problem identify appropriate Authentication mechanism.

→ Servlet Specification Defines the following 4 types of Authentication mechanisms.

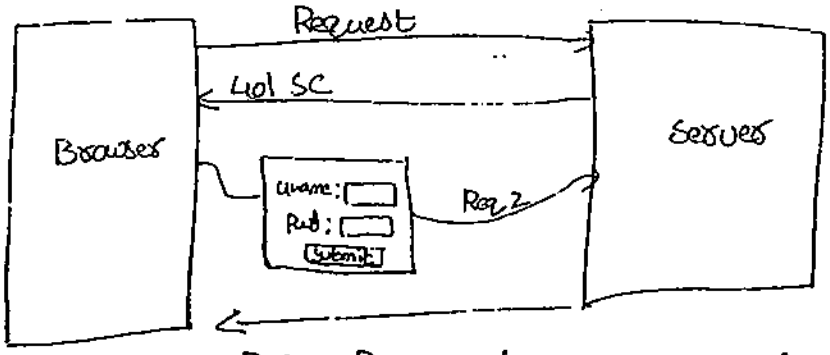
1. HTTP Basic Authentication
2. HTTP Form Based Authentication
3. HTTP Digest Authentication.
4. HTTPS Client-Cert Authentication.

HTTP Basic Authentication

→ It is the simplest & most commonly used Authentication.

→ This basic Authentication has introduced in HTTP 1.1 specification.

Process



Applicable only for 3 times

Process Response (or) 401 Status Code (uname, Pwd for Tomcat)

After 3rd time we will get as 401 status code as — unAthorised person.

→ Browser sends a request to the server. At this time browser don't know whether requested resource is protected or not. Hence it sends a normal HTTP Request.

→ Server observes that the Requested Resource is secure. Hence instead of sending the requested response it will sends 401 Status code saying its requires Authentication.

→ By receiving 401 status code, Browser opens a dialoge box prompting for username & pwd.

→ once the user enters username & pwd, browser resends the request with user credentials.

→ when the server receives the request it validates the username & pwd.

→ If they are valid, server sends proper required Response otherwise server sends again 401 status code.

Advantages of Basic Authentication

1. It is very easy to implement & set-up.
2. All browsers can provide support for this.

Limitations

1. username & pwd is sending in plain text form from client to server. Hence the security is very less in this Authentication. This Authentication follows Base-64 Encoding. (not Encryption)

2) we can't change look & feel of Dialoge Box.

HTTP Digest Authentication

→ This mechanism is exactly similar to basic Authentication, except that pwd is sending in Encrypted form.

→ This Encryption makes more secure.

Advantage

1) when compared with basic Authentication, Digest Authentic is more secure.

Limitations

1) only few browsers provide the support for Digest Authentication. (bcz Browser is responsible to perform Encryption.

2) most of web-servers Doesn't provide support for Digest Authentication bcz Servlet Specification doesn't tell it is mandatory.

3) we can't change look & feel of the Dialogue Box.

HTTP Form Based Authentication.

→ This mechanism is exactly similar to Basic Authentication except, instead of Depending on Browsers Dialogue Box we can provide our own login html form.

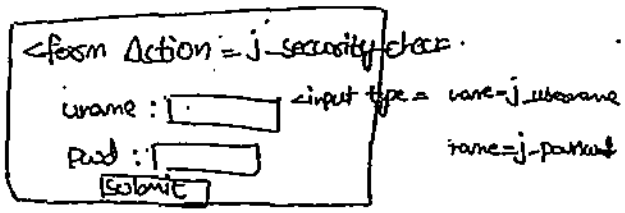
→ Developer is Responsible to provide login & Error pages. So that we can customise look & feel Based on our requirement.

The only requirement of the login form is

a) The value of Action attribute should be j_security_check.

b) The form should compulsory contain 2 text fields with the names j_username & j_Password.

c) Except this all the remaining are customisable (we can add any thing).



Advantages

It is very Easy to Set-up

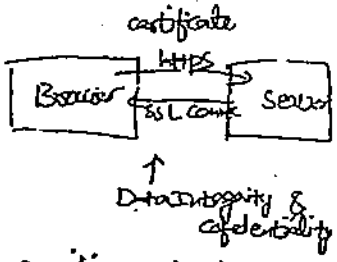
All the Browsers should provide the support.

we can customize the login form.

Limitation

Here also we are sending username & Pwd in plain text form hence it is not secure.

5.09 HTTPS Client-Cert Authentication



HTTPS means Http over Secure Socket Layer.

SSL is a protocol to ensure the privacy of sensitive Data Transmitted over the Internet.

In this mechanism Authentication is performed when SSL connection is established btw Browser & Server.

All the Data is Transmited in Encryption form by using Public key cryptography. which is Handled by both Browser & Server.

Advantage

→ It is the most Secure Type of Authentication.

Disadvantage

→ It is costly to implement & maintain

→ It require a certificate from 3rd party Certificate Authority like Verisign.

b)) Declarative Security. (2 to 3 Ques)

→ In the Deployment Descriptor declare

① <Security-Constraint>

② <web-resource>

③ <transport-guarantee>

④ <login-config>

⑤ <Security-role>

<Security-Constraint>

→ we can implement web-Security mainly by using the following 3 Tags.

① <Security-Constraint>

- It defines the resources which have to be Protected, which roles are allow to access & how the resource is transported across the network.

② <login-Config>

- It defines the Type of Authentication what we are using.

(3) <security-role>

- It defines the security roles which are allowed in the web Application.

⇒ All the above 3 Tags are Direct child Tags of web-APP & Here we can place any where with in <web-APP>

<Security-Constraint>

→ This Tag defines the following 3 child Tags.

(1) <web-resource-collection>

- Defines the resource which is to be protected.

(2) <auth-constraint>

- Authorization Constraints which determines what roles are allow to Access the resource.

(3) <user-data-Constraint>

- It specifies what type of Protection is required when transporting the resource across the network.

<web-resource-collection>

It contains the following 4 child Tags:

<web-resource-name>

<description>

<url-pattern>.

<http-method>

GET
POST
PS

- It specifies the http-method to which the security

constraint is applicable.

→ If we are not specifying this Tag, then security constraint is applicable for all http-methods.

<auth-constraint>

→ It specifies which security roles are allow to Access the protected resource.

→ It contains the following 2 child Tags.

① <description>

② <role-name>

– If the security constraint is applicable for all the roles we have to specify as follows.

<role-name> * </role-name>

<user-data-constraint>

→ This Tag contains the following 2 child Tags.

① <description>

② <transport-guarantee>

– This Tag specifies what type of guarantee we are providing while transporting the resource across the n/w.

⇒ The allowed values for this Tag are

① NONE – It means the Data is transported in plain text form. It is default value.

(2) Integral – It means the Data should not be changed in Trans.

(104)

(3) CONFIDENTIAL – It means the Data is Transported in Encryption Form.

⇒ The required Priority order is Confidential, Integral & None.

<login-Config>

⇒ This Tag specifies the Type of Authentication we are using.

⇒ It contains the following Sub-Tags.

(1) <auth-method>

- It specifies Authentication method.
- The allowed Values are
  1. BASIC
  2. DIGEST
  3. FORM
  4. CLIENT-CERT

(2) <realm-name>

- It specifies the location where we are storing Authentication Information.
- It is required only for basic Authentication.

(3) <form-login-Config>

- This Tag required to specify login Page url & Errors Page url in the case of FORM-Based Authentication.

- This Tag contains the following 2 child Tags.

① <form-login-page> /login.html <|f-l-p>

② <form-~~login~~ Error - page> <form-~~JSP~~ Error - page>
/error.jsp

<Security-role>

→ It can be used to define Security roles in the web Application.

→ This Tag contains the following 2 child Tags.

① <description>

② <role-name>

Demo Program for Basic Authentication.

FirstServlet.java

```
public class FS extends HS
{
   public void doGET( -- ) ---
   {
      Pw out = resp.getWriter();
      o.pln ("GET : After Authentication only we can Access this
                                                  Servlet");
   }

   public void doPOST( -- ) --
   {
      Pw out = resp.getWriter();
      o.pln ("POST : After Authentication only we can Access this
                                                  Servlet");
   }
}
```

I

<security-constraint>

<web-resource-collection>

<web-resource-name>

<description>

<url-pattern>

<http-method>

</web-resource-collection>

<auth-constraint>

<description>

<role-name>

</auth-constraint>

<user-data-constraint>

<description>

<transport-guarantee>

</user-data-constraint>

</security-constraint>

II

<login-config>

<auth-method>

<realm-name>

<form-login-config>

<form-login-page>

<form-error-page>

</form-login-config>

</login-config>

III

<security-role>

<description>

<role-name>

</security-role>

```
<web-app>
    FirstServlet
        ↑
      /test

<security-constraint>
    <web-resource-collection>
        <web-resource-name> check Servlet <
        <url-pattern> /test
        <http-method> POST
        <http-method> GET
    </web-resource-collection>

    <auth-constraint>
        <role-name> durgarole
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method> BASIC </a-m>
</login-config>

<security-role>
    <role-name> durgarole </r-n>
</s-r>
</web-app>
```

tomcatusers.xml

→ Add the following Entries in the tomcatusers.xml.

```
<tomcat-users>
  <role rolename="durgarole" />
  <user  username="durga" Password="ds" role="durgarole" />
  <user  username="divu" Pwd="Pop  role="durgarole" />
<Tomcat-users>
```

Demo Program for Form-Based

① FirstServlet.java

② <web.xml>

```
<web-app>
  <login-config>
      <auth-method> FORM <|auth-method>
      <form-login-config> /login.html  <|f-l-c>
      <form-error-page> /error.jsp  <|f-e-p>
  <|login-config>
<|web-app>
```

③ tomcat-users.xml

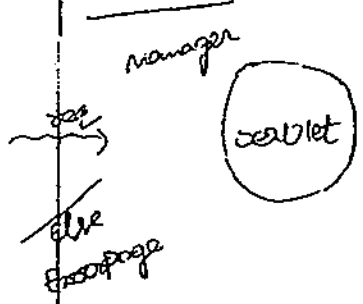
(4) login.html

```
<html>
<body> <h1> welcome to DSS </h1>
<h1>Plz login to avail the facilities</h1>
<form action="j_security-check">
Enter Name : <input type=text name="j_username">
Enter Pwd : <  "  "  Pwd="j_Password">
<input type=submit>
</body>
</html>
```

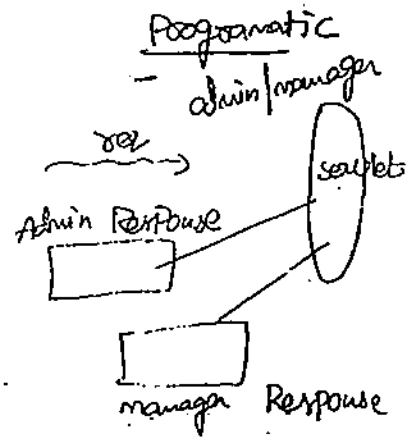
(5) Error.html

```
<h1> Your credentials are not correct. Plz provide valid
credentials </h1>
```

Declarative



Programatic



→ Some times Declarative Security is not enough compulsory we should go for Programatic Security.

→ Based on the users role we have to provide the corresponding Response. If the user is Admin then Admin related response & if the user is manager then manager related response we have to generate.

→ For this type of requirement compulsory we should go for programmatic Security.

→ we can achieve Programmatic Security by using the following methods of HttpServletRequest.

① boolean isUserInRole(String rolename)

— If the Authenticated user belongs to the specified role then this method returns "true".

— If the Authenticated user not belongs to specified role or if the user not Authenticated then this method returns "false".

admin
user: durga

② String getRemoteUser();

— Returns the authenticated username (login-Name).

— If the user has not been Authenticated then this method returns "Null".

3) Principal getUserPrincipal();

— Returns java.security.Principal object which contains Authenticated user name.

— Returns null if the user has not been Authenticated.

Sample

```
if (req.isUserInRole("admin"))
{
    o.println("This is Admins Home Page");
}
else
{
    o.println("This is general Home page");
}
```

⇒ The main problem in this approach is we are Hard-coding the role names in the Servlet.

→ If there is any change in the role-name modifying Servlet code is costly & creates maintainance problems.

→ To resolve this we have <security-role-ref>. By using this Tag we can map Hard-coded role names with original Role name.

```
<servlet>
    :
    <security-role-ref>
        <role-name> Admin </role-name>
        <role-link> DurgaAdmin </role-link>
    </security-role-ref>
</servlet>
```

(Dummy Name) This is logical role name which is Hard coded in servlet

original Role name.

FirstServlet.java


```
Public class FirstServlet extends HS
{
  v doGET(--)---
  {
      PW out = resp.getWriter();
      String name = req.getRemoteUser();
      o.pln("hi" + name);
      if(req.isUserInRole("admin"))
      {
        o.pln("This is Admins Home page");
      }
      else
      {
        o.pln("This is Others Home page");
      }
  }

    P v doPOST(--)---
    {
        doGET(req, resp);
    }
}
```

web.xml

```
<web-app>
   <servlet>
      <s-n> FS
      <s-c> FS
```

```
<security-role-ref>
        <role-name> admin </role-name>
        <role-link> DurgaAdmin </r-l>
  </security-role-ref>

</servlet>

    FirstServlet
        ↑
      /test

<security-Constraint>
  <web-resource-Collection>
        <web-resource-name> check Servlet </w-r-n>
        <url-pattern> /test
        <HTTP-method > GET

    </w-r-c>
<login-config>
  <auth-Constraint>
        <role-name > DurgaAdmin
        <role-name > Durga manager
    </a-c>
</s-c>
<login-Config >
    <auth-method > BASIC
  </login-config>

  <Security-role>
        <role-name > Durga Admin
    </s-r >
```

```
<security-role>
    <role-name> DurgaManager
</s-r>
```

tomcatuser.xml

```
<tomcat-users>
    <role rolename="durgaAdmin"/>
    <role rolename="durgaManager" />

    <user username="durga" Password="SCJP" role="durgaAdmin"
    />
    <user username="pallan" Pwd="SCWCD" role="durgaManager"
    />.
</tomcat-users>
```

0
0
0

13/05/11

JSP

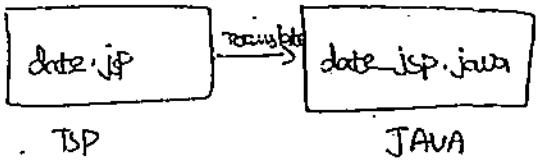
UNIT-I

Obj)) Describe the purpose & Event Sequence of JSP Page Life cycle

(a) Translation phase
(b) JSP Page compilation
(c) Load the class
(d) Create an Instance.
(e) call JSP init()
(f) call -JSP Service()
(g) call -JSP Destroy()

→ The Process of Translating JSP Page (.jsp file) in to corresponding Servlet (.java file) is called Translation phase

→ This can be done by the JSP Container. In Tomcat this Component is called "JASPER".



The Server Time is:
<%= new java.util.Date() %>

→ The generated .java file is available in the following location in Tomcat.

C:\Tomcat\work\catalina\localhost\JSP1\org\apache\jsp\

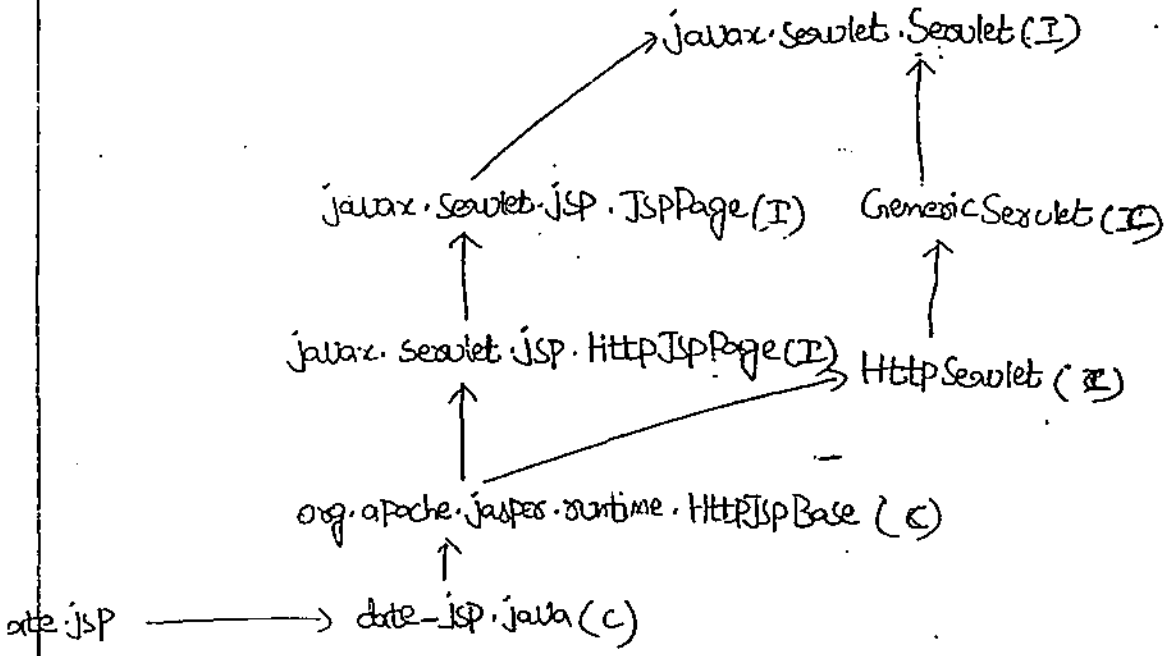
Name of the Application.

date_jsp.java

JSP API

- Every class which is generated for the JSP must implement javax.servlet.jsp.JSPPage (I) or javax.servlet.jsp.HttpJspPage either directly or Indirectly.

- Tomcat People provided a Base class org.apache.jaspes.runtime.HttpJspBase for implementing the above 2 interfaces. Hence any servlet which is generated by Tomcat is extending this HttpJspBase class.



javax.servlet.JSP.JSPPage (I)

②

→ This interface Defines the following 2 Life Cycle methods.

① jspInit()

② jspDestroy()

jspInit()

```
Public void jspInit()
```

→ This method will be executed whenever we are sending first Request to the JSP to perform Initialization Activities.

→ web Containers always calls init(ServletConfig config) of HttpJspBase class which intern calls jspInit();

→ If we have any Initialization Activities we have to overRide jspInit() in the JspPage.

```
class HttpJspBase {
final method init(ServletConfig config)
    {
        P V jspInit()
        {
            Empty Implementation
        }
    }
}
```

<%= new java.util.Date() %> How to overside jspInit() in our class.

```
<%!
  Public void jspInit()
  {
     S.o.ln ("my Init method");
  }
%>
```

→ we can't place init(ServletConfig config) in the JSP. Because it is declared as final in HttpJspBase class.

jspDestroy()

```
Public void jspDestroy()
```

→ This method will be executed only once to perform clean-up activities Just before taking jsp from out of service.

→ web container always calls destroy() available in HttpJspBase which intrans calls jspDestroy().

→ we can overside jspDestroy() in our JSP.

→ we can't overside destroy() in the JSP bcz it is Declared as final in HttpJspBase class.

```
class HttpJspBase
{
final class method destroy()
   {
      jspDestroy();
   }
}
```

14/05/07 _jspService() · [illegible] defines only one method

Public void _jspService(HtsR req, HtsR resp) Throws SE, IE

→ web containers always calls service() available in HttpJspBase class (3) which intern calls _JspService() method.

→ At the time of Translation JSP Engine always generates _JspService() in generated Servlet class.

→ If we are placing _jspService() Explicitly in Jsp the generated Servlet Contains 2 _jspservice() · If a class Contain more than one method with the same signature we will get Compile-time Error. Hence if we are placing _jspservice() explicitly we will also get Compile Time Error.

→ In the Jsp we are not allow to override Service() explicitly. bcz it is declared as final in Parent class (i.e HttpJspBase).

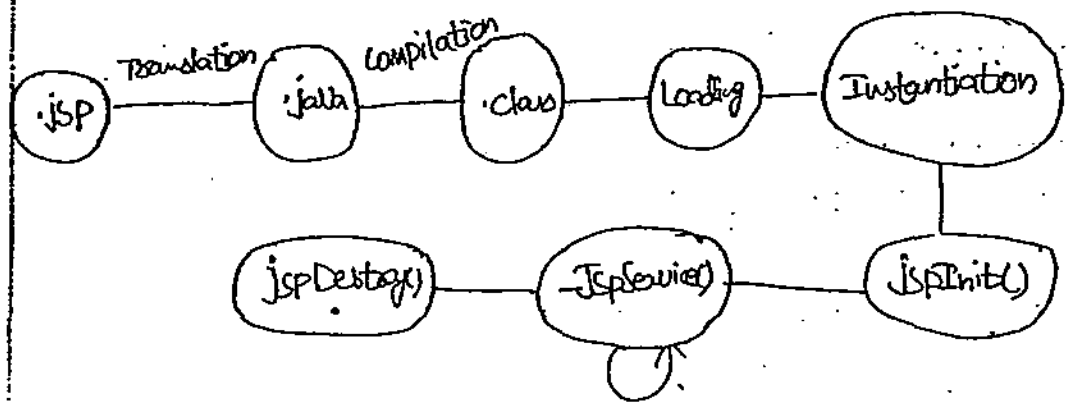
→ In the Jsp which of the following methods are allow to place explicitly.

(1) JspInit()
(2) init()
(3) JspDestroy()
(4) destroy()
(5) _jspservice()
(6) service() ·

(2), (4) (6) are final class methods

_jspservice() is automatically generated by JspEngine.

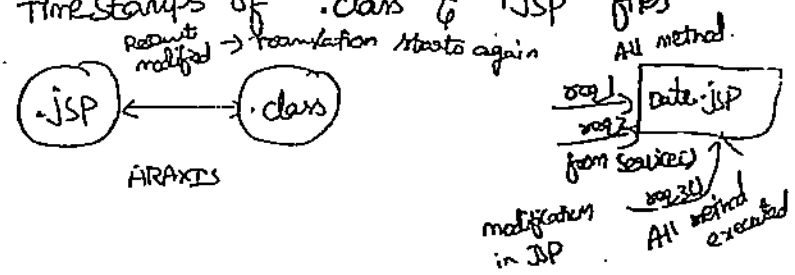
Life Cycle of the JSP



The JSP Page will participate in Translation phase in following cases-

1) At the Time of First Request.

2) If the source code of JSP got modified when compared with earlier request. For this JSPEngine use ARAXIS Tool to compare TimeStamps of .class & .jsp files



Pre Compilation of JSP

We can Initiate Translation Phase Explicitly by using Pre-Compilation process.

JSP specification Provides a special request Parameter jsp_precompiler. This parameter can be used to hit the JSP for compilation without processing any request.

⇒ we can invoke pre-compilation has follows.

http:\\ localhost :8080\ date.jsp ? jsp_Precompiles =true .

[⇒ till jspInit() will be executed.

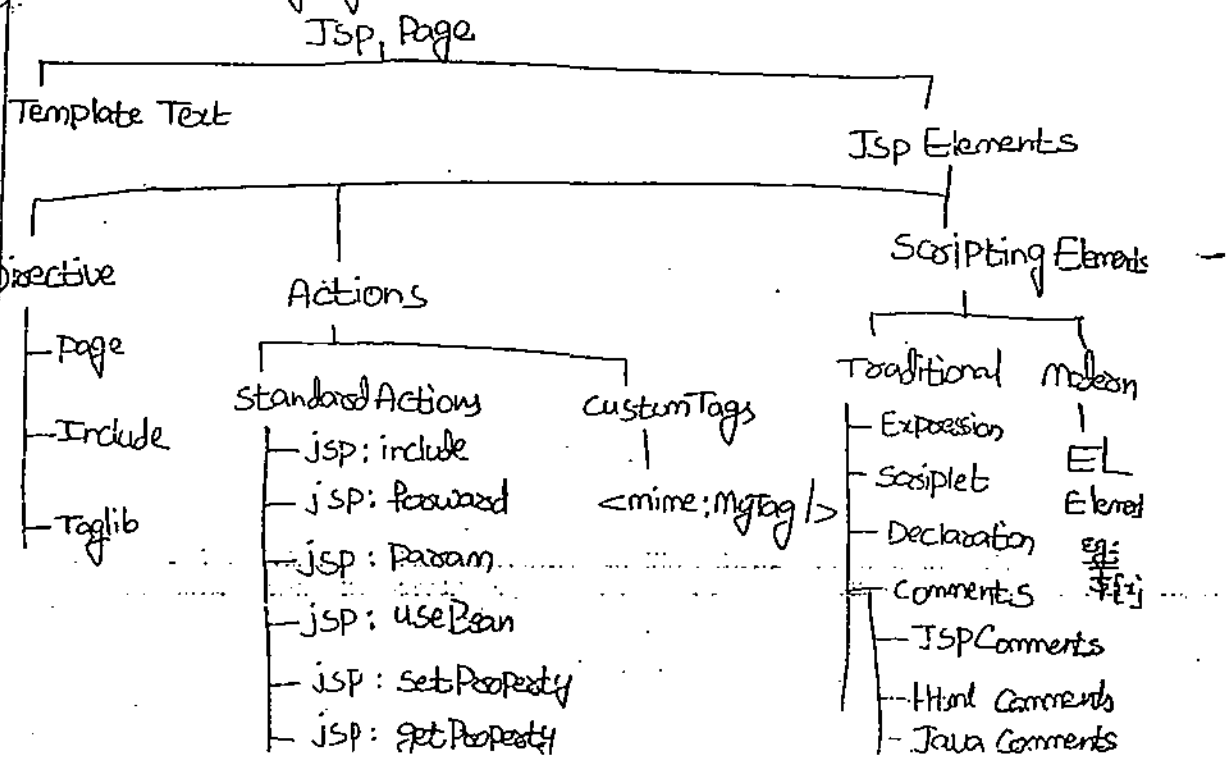
If we remove ~~ then service() will be executed

Example Date.jsp → P V jspInit() { }]

JSP Scripting Elements

6) Identify, Describe, write JSP Code for following Elements.

1) Template Text
2) Scripting Elements
3) Standard & Custom Actions
4) Expression Language Elements.



Template Text

It contains Html & xml Tags and raw Data.

For the Template Text no Translation is required & it will Pass Directly as Argument to out.write() in _jspService()

```
<h1>
   The Server Time is : <%= new java.util.Date() %>
<|h1>
```

Template Text

It will become argument to out.write()

JSP Expression.

→ It will become argument to out.print().

The generated servlet is

```
Public final class date_JSP ___
{
   Public void _jspService( --)
   {
      out.write ("<h1> The Serve Time is ");
      out.print(new. java.util.Date());
      out.write ("<|h1>");
   }
}
```

no processing is Required.

write() can take only character Data as Argument and Template Text is always character Data. Hence Template Text will become argument to the write().

-> Expression Value can be any Data Type Hence we can't use write() compulsory we should go for print() which can take any Type of Argument.

Directives

-> These can provide general Information about JSP Page to the JSP Engine like Session Tracking is required or not, is current JSP is ErrorPage or not, which scripting language used in the JSP etc...

<%@ name attributename = attributeValue %>

page | include | taglib

-> These are 3 types of Directives are Possible.

1. Page Directive

<%@ page attributename = attribute Value %>

-> can be used to define page specific Attributes.

2. Include Directive.

<%@ include file="header.html" %>

-> To include the Content of header.html in current JSP at Translation time.

3. Taglib

www.java2s.com

<%@ taglib Prefix="mine" uri="www.java2share.com" %>

-> To make custom Tags available to the JSP.

5) <%= 24*36 %>

6) <%= 5>3 %>

7) <%= new Student() %> → toString() will be called.

⇒ Expression values will become argument to outPrint() inside _jspService().

Scriplet

⇒ we can use Scriplet to place java code in the JSP.

```
<%
    Any Java code ;   ⇒   Public void _jspService(--)
                          {
%>
                          }
```

→ Java code inside Scriplet will be placed Directly inside _jspService()

→ All Java Stmts inside Scriplet should compulsory Ends with Semicolon (;).

⇒ write a program to Display Hit Count of the JSP

```
<%
  int count=0;
  count ++;
  o.pln(count);
%>
```

It is Invalid & Generates always '1' as Response.

→ Local variable in _jspService()

```
<%!
    int count=0;
%>
<%
    count++;
    open(count);
%>
```

Valid & Generates 1, 2, 3 --- as output

Instance Variable ← int count=0;

```
public _jspService(--)
{
    count++;
    open(count);
}
```

Difference btw

```
<%
    int i=0;
%>
```

→ Local Variable in _jspService()

```
<%!
    int i=0;
%>
```

→ Instance Variable in _jspService();

→ Even we can take static Vbls also;

⇒ Its not recommended to use Scriplets in the JSP.

Declaration Tags

6/05/09

⇒ This can be used for Declaring Instance members, static members, Instance or static Blocks, Inner classes etc.

```
<%!
    Declarations.
%>
```

⇒ These Declarations will be placed Directly in the generated Servlet but outside of _jspService().

```
<%!
    int i=10;
    Static int j=20;
    int[] a={10,20,30};
    public void m1()
    {
        S.O.pln("Hi");
    }
%>
```

⇒ Every Java Stmt inside Declaration Tag should compulsory Ends with semicolon (;).

<u>Note</u>

→ All the Implicit objects available in JSP are Declared as Local variables Inside _jspservice().

→ Declaration Tag Code will be placed outside of _JSPService() in the generated servlet. Hence implicit objects we are not allow to use inside Declaration Tag otherwise we will get CompileTime Error.

<u>hello.jsp</u>

<u>CTError</u>

out cannot be resolved.

```
<%!
    Public void m1()
    {
        out.pln("Hi");*
        S.O.pln();✓
    }
%>
<%
    m1();
%>
```

```
_jspservice()
{
    out is local to this method.
}
```

→ If we use out of this method we will get CTError.

Comments

→ There are 3 Types of Comments allow to use in the JSP

1) JSP Comment.

```
<%-- This is JSPComment --%>
```

→ Also known as Hidden Comments bcz these are not visible in the Remaining phases of JSP Execution.

→ It is Highly Recommended to use JSP Comments in the JSP.

2) Html Comments.

```
<!-- This is Html Comment -->
```

Generated Servlet .class Response.

→ Also known as Template Text Comments.

→ These are visible to the End user as the part of generated Response Source Code.

→ Hence these are not recommended to use in the JSP.

3) Java Comments.

```
<%
    // single line Java Comment
    /* multiple lines Java Comment
%>
```

Generated Servlet

→ Also known as Scripting Comments.

→ Also visible in the generated Servlet Source Code but in all other phases these are not visible.

Comment Type	In JSP page	Generated Servlet	To user Response Source code
① JSP Comments	Visible	NO	NO
② HTML Comments	visible	visible	visible
③ Java Comments	visible	visible	NO

⇒ Among Expressions, Scriplets, Declarations & JSP Comments we can't use one Inside Another i.e Nesting of these Scripting Elements is not possible. otherwise we will get CompileTime Errors.

Sample:-

① <%
out.println(<%= new java.util.Date() %>);
%>

unable to understand by Java compiler.

(1) <%
<%
out.println("Hi");
%>

② <%
<%-- This is very Important Line -->
out.println("Hi");
%>

③ <%!
<%-- This is JSPComment -->
public double getBalance()
{
return 10,000;
}
%>

Comparision table for JSP Scripting Elements.

Scripting Elements	Syntax	Ends with ;	Is code generated Inside _jspService()
(1) Expression	<%= %>	No	yes
(2) Scriplet	<% %>	yes	yes
(3) Declaration	<%! %>	yes	No
(4) comments	<%-- -->	NA	Not Applicable.

JSP Directives

Q) write JSP Code that uses the following Directives.

(1) Page (2) Include (3) Taglib

-> Directives provides general Information about our Jsp to the JSP Engine. These are Translation Time Instructions to the JSP Engine.

<%@ directiveName attributname = attributeValue %>

directiveName ↓ page | include | Taglib

attributeValue ↓ Compulsory we Should Take Single or Double ~~Codes~~ Quotes.

Page Directive

→ page Directive specifies the overall properties of JSP Page to the JSP Engine.

ex:

① <%@ page language="java" %>

- specifies to the JSP Engine the Scripting Language used in current JSP is java.

⇒ we can use Page Directive any no. of Times, Any where in the JSP.

② <%@ page session="true" %>

⇒ The following is the List of all Possible 13 Attributes in the page Directive.

① import	⑧ language
② session	⑨ Extends
③ contentType	⑩ buffer
④ isELIgnored	⑪ AutoFlush
⑤ isThreadSafe	⑫ Info
⑥ isErrorPage	⑬ PageEncoding.
⑦ errorpage	

import

→ we can use import attribute for importing classes & interfaces in the JSP.

→ This is similar to import stmt in core java.

with out import Attribute

```
<h1> The Server Time is:
  <%= new java.util.Date() %>
<|h1>
```

with import Attribute

```
<%@ Page import ="java.util.*" %>
  <h1> The Server Time is :
    <%= new Date() %>
  <|h1>
```

→ To import multiple packages the following are various possibilities.

1. `<%@ Page import="java.util.*" %>`
   `<%@ Page import ="java.sql.*" %>`
2. `<%@ Page import ="java.util.*" import="java.sql.*" %>`
3. `<%@ Page import ="java.util.*, java.sql.*" %>`

* : with in the same JSP we are not allow to take any Attribute except import of page directive multiple times with diff values.

→ But we can take same Attribute multiple times with same value.

→ which of the following page Directives are valid.

(1) <%@ page session="true" session="true" %>

(2) <%@ page session="true" session="false" %>

(3) <%@ page session="true" %>

(4) <%@ page import="java.util.*" import="java.sql.*" %>

(5) <%@ page import = java.util.* %> code Quote Expected

In (2) we will get Translation Time Error saying Page Directive: illegal to have multiple occurances of session with diff values (old : true, new : false).

⇒ Inside JSP there is no need to import the following packages bcz these are available by default.

```
javax.servlet.*;
javax.servlet.http.*;
javax.servlet.jsp.*;
java.lang.*;
```

Session

→ we have to use this attribute to make session object unavailable to the JSP.

→ By default session object is available in every JSP. If we don't want session object we have to declare page directive as follows.

```
<%@ Page Session="false" %>
```

→ In this case session implicit object not available to the JSP.

→ If we are not declaring session attribute explicitly or declaring session attribute with "true" value, the equivalent generated servlet code is

```
HttpSession session = null;

session = pageContext.getSession();
```

if false
→ These stmts will not be present in Generated Servlet Code.

→ The allowed values for the session attribute are.

"true"	TRUE	True	Durga X
"false"	FALSE	FALSE	

i.e case insensitive string of True or false.

→ If we are taking any other we will get Translation Time Error.

```
<%@ Page Session="durga" %> X Invalid value for session.
```

ContentType

→ we have to use this attribute to specify content Type (MIME Type of the response).

```
<%@ Page contentType = "image/jpeg" %>
```

→ The Default value of the ContentType attribute is text/html.

isELIgnored

→

	EL
`<%= session.getId() %>`	`${ session.id }`
`<%= req.getParameter("uname"); %>`	`${ Param.uname }`

→ In JSP 1.2V Expression Language has introduced the main objective of EL is to eliminate Java Code from the JSP.

→ isELIgnored = "true"

EL Syntax Just Treated as Template Text (plain Text) with out any procession.

→ isELIgnored = "false"

EL Syntax Should be Evaluated & Point its value.

Example

```
<%@ Page isELIgnored = "true" %>
<h1> Session ID : ${ se PageContext.session.id } .
O/p- Session ID : ${ PageContext.session.id }
```

<%@ page isELIgnored = "false" %>

<h1> Session ID : ${pageContext.session.id}

SessionID : 1456789

→ The Default Value in JSP 1.2 V is true. But from JSP 2.0 onwards the Default Value is false.

18/08/09

isThreadSafe

→ we can use this Attribute to provide Thread Safety to the JSP.

(a) isThreadSafe = "true";

It means the JSP Page is already ThreadSafe & it is not required to generated Servlet to implement Single Thread model.

→ JSP Page can process any no of client Request simultaneously.

(b) isThreadSafe = "false".

The current JSP is not ThreadSafe. Hence to provide ThreadSafety Generated Servlet will implement SingleThreadModel.

→ In this case JSP can Process only one client Request at a Time.

⇒ The Default Value of isThreadSafe attribute is "true".

→ To provide Thread Safety which of the arrangement we have to take in the JSP.

(1) <%@ Page isThreadSafe = "true" %>

(2) <%@ Page isThreadSafe = "false" %>

3) Not Required any arrangement bcz every JSP is default Thread Safe.

Summary of Page Attributes

Attribute Name	Purpose	Default Value.
1) import	To imports classes & Interface.	javax.servlet.*; javax.servlet.http.*; javax.servlet.jsp.*; javax.servlet.lang.*;
2) session	To make sessionObject unavailable To JSP Page	true
3) contentType	To specify mime Type of the Response	text/html
4) isELIgnored	To Disable expression language in the JSP	false.
5) isThreadSafe	To Provide Thread Safety to the JSP for resolving Data Inconsistancy Problems.	true.

Include Directive

→ Inclusion — Static ⇒ Include Directive
— Dynamic ⇒ Include Action → Dynamic Inclusion.

→ If several JSP contains same code it is recommended to seperate that common code in seperate file. wherever that common file is required Just we have to Include that file.

→ The main Advantages of this approach are

(1) Promotes Reusability.
(2) It improves maintainability.
(3) Enhancement will become very Easy.

⇒ we can achieve this inclusion either by using include Directive or Include Action.

Include Directive

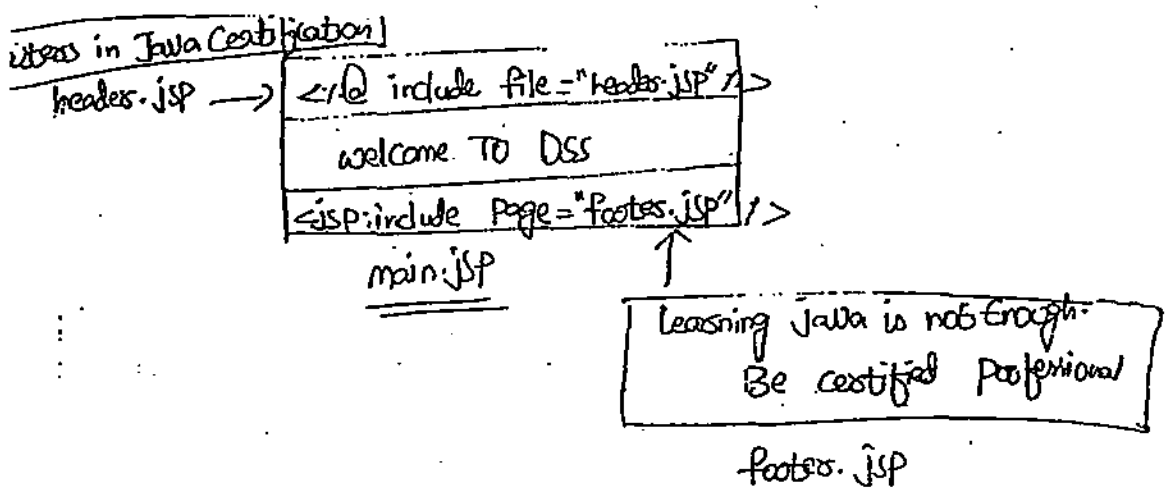
```
<%@ include file="header.html" %>
```

→ The inclusion will happen at Translation Time Only. Hence this Inclusion is also called Static Include or Translation time Inclusion.

Include Action

```
<jsp:include page="header.html" flush="true" />
```

→ The Inclusion will happen at Request Processing Time. Hence this inclusion also called Dynamic Include or RunTime Inclusion.


ites in Java Certification
header.jsp
<%@ include file="header.jsp" %>
welcome TO DSS
<jsp:include page="footer.jsp" />
main.jsp
Learning Java is not Enough.
Be certified Professional
footer.jsp

Diff btw Include Directive & Include Action.

(14)

Include Directive	Include Action
① <%@ include file="header.jsp" Contains only one Attribute file	① <%@. <jsp:include page = header.jsp flush="true"/> Contains 2 Attributes Page, flush.
② Inclusion will happen at Translation Time Hence it is considered as Static Include	② Inclusion will happen at Request Processing Time. Hence it is considered as Dynamic Inclusion.
③ For both Including & Included JSP's a single Servlet will be generated. Hence code sharing btw the components is possible.	③ Seperate Servlets will be generated for both Including & Included JSP's. Hence code sharing btw the components is not possible.
④ Performance is High	④ Performance is low.
⑤ There is no guarantee for Inclusion of latest version of Included JSP. It is vendor Dependent.	⑤ Always Latest Version of Included Page will be Included.
⑥ If the Target Resource won't change frequently then it is Recommended to use Static Include.	⑥ If the Target Resource will change frequently then it is recommended to use Dynamic Include.

⇒ To include header.jsp at Translation Time the required code is.

```
<%@ include file="header.jsp" %>.
```

⇒ To Include header.jsp at Request Processing Time. the required code is

```
<jsp:include page="header.jsp" />.
```

⇒ which of the following inclusions are valid.

1) ✗ `<%@ include page="header.jsp" %>`
2) ✓ `<%@ include file="header.jsp" %>`
3) `<jsp:include file="header.jsp" />`
4) ✓ `<jsp:include page="header.jsp" />`
5) ✓ `<jsp:include page="header.jsp" flush="true" />`
6) ✗ `<%@ include fil="header.jsp" flush="true" %>`

⇒ Page is the attribute applicable for Include Action where as file Attribute is applicable for Include Directive.

⇒ Flush Attribute is applicable only for Include Action but not for Include Directive.

Taglib Directive

-> It makes custom Tags available to the Jsp.

<%@ taglib prefix="MIME" uri="www.durgasoft.com" %>

-> The Taglib Directive Contains 2 Attributes prefix & Uri.

-> Every custom Tag we are using in the Jsp must have a separate Taglib Directive.

-> uri represents the location of the tld file.

Jsp Implicit Objects

Obj) Given a design goal write the Jsp Code using appropriate implicit objects.

-> when compared with servlet Programming Developing Jsp's is very easy by the required mandatory stuff automatically available in every Jsp. Implicit Objects also one of such areas which are by default available for every Jsp.

=> The following is the list of Jsp Implicit Objects.

Implicit Object	Declared Type
(1) request	javax.servlet.http.HttpServletRequest (I)
(2) response	javax.servlet.http.HttpServletResponse (I)
(3) Config	javax.servlet.ServletConfig (I)
(4) application	javax.servlet.ServletContext (I)

5) session	javax.servlet.HttpSession (I)
6) out	javax.servlet.jsp.JspWriter (Abstract class)
7) page	java.lang.object (class)
8) pageContext	javax.servlet.jsp.PageContext (Abstract class)
9) exception	java.lang.Throwable (class)

05/09

Request & Response Object.

→ Request & Response Implicit objects are available as arguments to service method.

→ All the methods of HttpServletRequest & HttpServletResponse can be applicable on these request & response implicits objects.

```
<h1>
  The Request method is : <%= request.getMethod() %>
  The username is : <%= request.getParameter("uname"); %>
  The cookie array is : <%= request.getCookies();
  The Request Host is : <%= request.getRemoteAddr();
  response.setContentType("text/html");
  response.setHeader("durga","SCJP");
%>
</h1>
```

→ In browser if we give uname=45

Application

⇒ This implicit object is of type ServletContext which represents the Environment of the application.

```
<h1>
  The application name is : <%= application.getServletContextName() %>
  The Init Parameter uname : <%= application.getInitParameter("uname") ; %>
</h1>
```

Session

⇒ In Every JSP by default Session Object is available. It is of type HttpSession.

Session is alread available by default

```
<h1>
  The Session ID is : <%= Session.getID() %>
</h1>
```

⇒ If we can make explicitly Session Object unavailable by using Page Directive.

Session is not available

```
<%@ Page Session="false" %>
  <h1>
    The SessionID is : <%= Session.getId() %>
  </h1>
```

CTE - Session Cannot be Resolved.

⇒ Which of the following makes Session Object available to the JSP

<%@ Page Session="true" %>

<%@ Page Session="false" %>

<%@ Page Session="false" Session="true" %> Translation Time Error

9) <%@ Page contentType="text/html" %> available.

Page

→ The Page implicit variable is always pointing to current Servlet Object.

→ In the Generated Servlet this var is declared as follows.

Object page=this;

Parent class ref can be used for holding child class obj.

⇒ which of the following are valid.

→ with parent ref we can call only parent class methods but not child class methods.

① <%= page.getServletInfo() %>

② <% = this.getServletInfo() %>

③ <% = getServletInfo() %>

④ <% = ((HttpServlet) page).getServletInfo() %>

① CTE – The method getServletInfo() is undefined for the type object.

→ Page implicit object is declared as object type. Hence we are allow to call object class methods only but not servlet specific methods. Hence Page Implicit var is the most rarely used var to the JSP.

Config

→ Config Implicit Object is of Type ServletConfig.

→ whatever the methods we can apply on the ServletConfig object, the Same we can apply on config also.

→ The following is the list of applicable methods.

1. getInitParameter()
2. getInitParametersNames()
3. getServletName() servlet Logical Name.
4. getServletContext()

date.jsp

```
<h1>
  The Init Parameter is : <%= config.getInitParameter("uname") %>
  The Logical Name is : <%= config.getServletName() %>
</h1>
```

web.xml

```
<web-app>
  <servlet>
    <s-n> DemoServlet </s-n>
    <jsp-file> /date.jsp </j-f>
    <init-param>
        <Param-name> uname </p-n>
        <Param-value> SCOTT </p-v>
```

```
<|Servlet>

<Servlet-mapping >
      <s-n> DemoServlet  <|s-n>
      <url-pattern > |test <|u-p>

S| s-m>

<|w-a>
```

JSP
- date.jsp
- WEB-INF
  - web.xml

http:// localhost : 8080 / JSP / date.jsp → Directly calling JSP.

The Init Parameters is : NOIL

The Logical Name is : JSP (Default Value)

http:// localhost : 8080 / JSP / test → Through url-pattern.

The Init Parameters is : SCOTT

The Logical Name is : DemoServlet.

⇒ To reflect Servlet level web.xml configurations in the JSP compulsory we should access by using url-pattern only.

PageContext

⇒ The Page Context Implicit object is of Type javax.servlet.jsp.PageContext.

→ It is an abstract class & Vendors is responsible to Provide implementation.

→ PageContext is the child class of JSP Context.

JspContext → Simple Custom Tag JSP 2.0V

↑

PageContext → Classic Custom · Custom Tag JSP 1.1V.

→ we can use PageContext Object for the following purposes.

1. To get All other JSP Implicit Objects
2. For Request Dispatching Purposes.
3. For Attribute Management in any Scope.

Getting JSP Implicit Objects from PageContext

→ By using PageContext Implicit Object we can get all other JSP Implicit Objects i.e PageContext acts as Single STOP for all other Implicit Objects.

→ Page Context class contains the following methods for this

1. request — getRequest();
2. response — getResponse();
3. config — getServletConfig();
4. application — getServletContext();
5. session — getSession();
6. out — getOut();
7. Page — getPage();
8. Exception — getException();

⇒ These methods are not usefull with in the JSP but these are useful outside of JSP mostly in Custom Tag handlers.

Request Dispatching by using PageContext

→ we can perform Request Dispatching by using the following methods of PageContext.

① Public void forward (String Target)

② Public void include (String Target)

⇒ The Target Resource can be specified either by Absolute Path or by Relative Path.

Example

First.JSP

```
<h1>
 This is First JSP
</h1>

<%
  PageContext.forward ("second.jsp");

 %>
```

Second.JSP

```
<h1> This is Second JSP </h1>
```

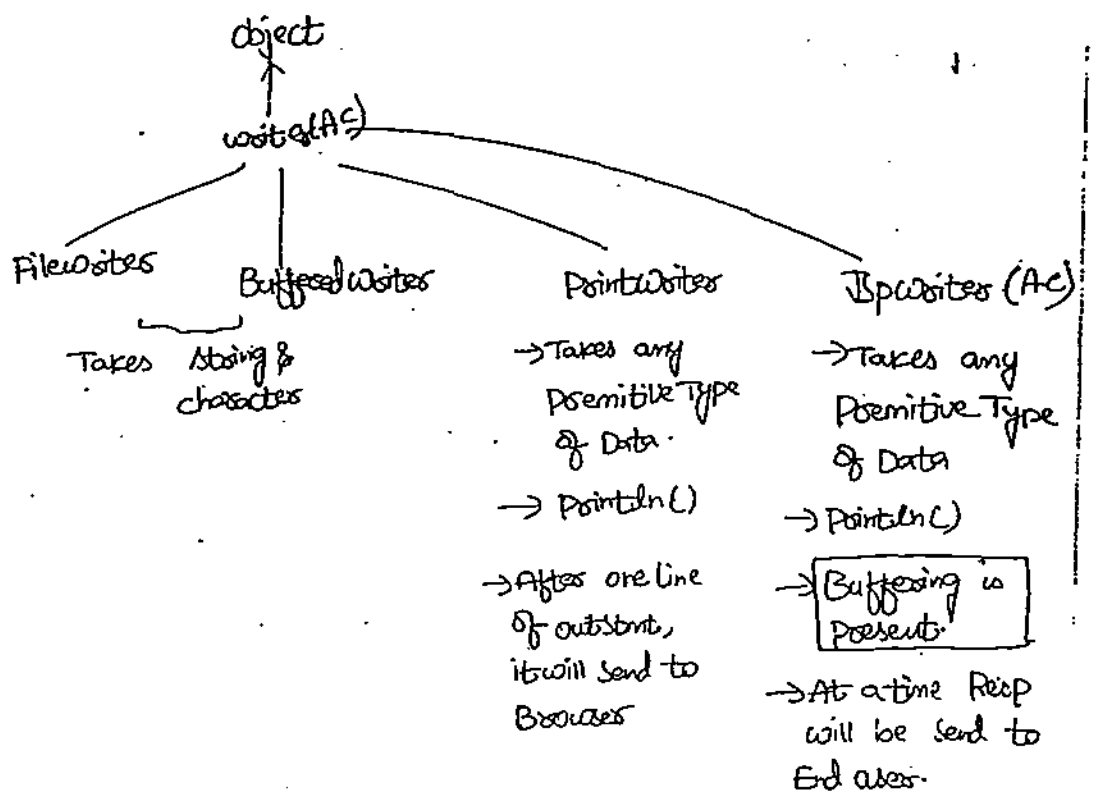
O/P forward — This is Second JSP

include — This is First JSP
This is Second JSP

Attribute Management in Any Scope

→ we can use PageContext Object to perform Attribute management in any scope. (will be covered in JSP Scopes).

21/05/09



Out

→ This implicit object is of Type javax.servlet.jsp.JspWriter. This class is specially Designed for JSP's to write Character Data to the Response.

→ The main Diff btw JspWriter & PrintWriter is, in the case of PrintWriter no buffering is available. Hence whenever we are adding Character Data to the response it will be delivered immediately to the browser with out waiting until commiting the Response.

→ But in the case of buffering concept is available.

→ wherever we are adding Data to the response, it will store in the buffer without Delivering immediately. once the response is committed then only entire Response will be delivered to the Browser.

→ Except this Buffering Diff there is no other Diff btw JspWriter & PrintWriter.

JspWriter = PrintWriter + Buffering

Example

Demo.jsp

```
<%
    PrintWriter out1 = response.getWriter();
    out.println("Hello YSR Good Morning");
    out1.println("Hello Babu Good Morning");
    out.println("Hello YSR Good Morning");
    out1.println("Hello Babu Good Morning");
    out.println("Hello YSR Good Morning");
    out1.println("Hello Babu Good Morning");
%>
```

PrintWriter — out1

JspWriter — out

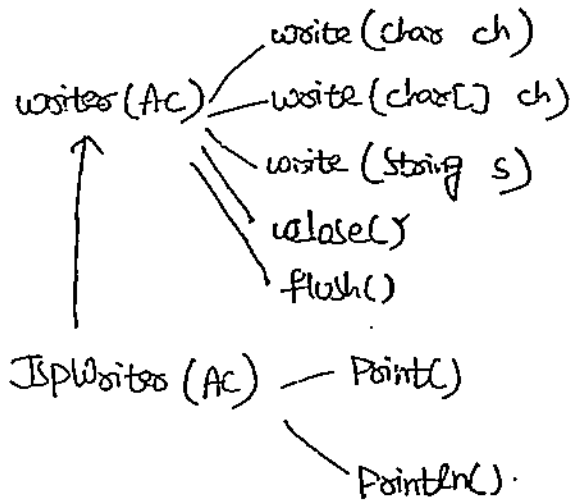
op: Hello Baby Good Morning
Hello Baby " "
Hello Baby " " } PointWriter

Hello USR Good Morning
Hello USR " "
Hello USR " " } JspWriter

→ With in the JSP we can use either PointWriter (or) JspWriter but not recommended to use both simultaneously

→ JspWriter is the child class of writer (AC). Hence all the methods available in writer as by default available to the JspWriter through Inheritance.

→ In addition to these methods, JspWriter contains its own set of Point() & Pointln() to add any type of Data to the Response.



Example

```
<%
    out.pln("The PI Value is"); String
    out.pln(3.1415); float
    out.pln("This is"); characters
    out.pln(true); Boolean
%>
```

→ How we can write Response in JSP's.

A) By using JspWriter.

→ what is the diff btw PrintWriter & JspWriter.

JspWriter = PrintWriter + Buffering

Configuring ErrorPages in JSP's (Exception)

→ we can Configure Error Pages in JSP's by using the following 2 approaches.

① Declarative Approach

② Programmatic Approach:

Declarative Approach

→ we can Configure Error Pages according to a particular exception or according to error-code in web.xml as follows.

```
<web-app>
    <error-page>
        <exception-type> java.lang.AE </e-t>
        <location> /error.jsp </l>
    <error-page>
    <error-page>
        <error-code> 500 </e-c>
        <location> /error.jsp </l>
    </e-p>
</w-a>
```

→ The error pages configured in this approach are applicable to the entire web application.

Programatic Approach

→ we can configure ErrorPage for a particular JSP by using errorPage Attribute of Page Directive.

Demo.jsp

```
<%@ page errorPage="error.jsp" %>
<h1> The Result is : <%= 10/0 %> </h1>
```

→ In Demo.jsp if any Exception or Error occurs then error.jsp is responsible to report this Error.

→ This way of Configuring Error Page is applicable for a particular JSP.

→ we can declare a JSP as errorPage by using isErrorPage attribute of Page Directive.

→ In the ErrorPages only exception implicit object is available.

<u>Error.JSP</u>

```
<%@ page isErrorPage="true" %>
  <h1> we are facing some Technical Problems. Plz Try
  Later.
    The Problem we are facing is : <%= exception %>
```

→ If JSP is not declared as errorPage but if we are trying to access Exception Implicit object we will get CompileTimeError Saying – Exception can not be resolved.

→ If we are accessing errorPage Directly without any exception, then exception implicit object refers "null".

which approach is recommended.

Declaration Approach is recommended bcz we can customise Error Pages based on Type of Exception or error-code.

<u>Note</u>

① All JSP Implicit Objects are available as local Vars of _JSPservice() in Generated Servlet. Hence we are allow to use these implicit objects in Scriplet & Expression Tag.

Bcz this code will be placed in _JspService().

But we are not allowed to use in Declarative Tag bcz its code will be placed outside of service().

Among all Jsp Implicit Objects except Session & Exception all the remaining Objects always available in every Jsp. we can't make them unavailable.

Session Object by Default available in every Jsp. If we don't want we can make it unavailable by using PageDirective

```
<%@ Page Session="false" %>
```

Exception Implicit Object is by default not available in every Jsp. we can make it available by using PageDirective

```
<%@ Page isErrorPage="true" %>
```

S(C)

JSP Scopes

→ In Servlets we have the following 3 scopes for storing Information in the form of Attributes.

1. Request Scope
2. Session Scope
3. Application Scope.

→ In addition to these 3 scopes in JSP's we have pageScope also.

Request Scope

→ In Servlets it is maintained by Servlet Request Object but in JSP's it is maintained by request Implicit Object.

→ The Information stored in request Scope is available for all the components which are processing that request.

→ request Scope will be start at the time of request object creation i.e. before starting Service method & will be lost at the time request object Destruction i.e Just after completing Service method.

→ we can perform attribute Management in request Scope by using the following methods:

1. Public void setAttribute (String name, Object value);
2. public Object getAttribute (String name);

3. public void removeAttribute (String name);
4. public Enumeration getAttributeNames();

Session Scope

→ In Servlets Session Scope is maintained by HttpSession Object but in JSP's it is maintained by session implicit Object.

→ Session Scope will be started at the time of Session Object creation (i.e at the time of executing get session method): & will be lost once session Expires either by invalidate() or TimeOut.

→ The information stored in the Session Scope will be available for all components which are participating in that Session.

→ HttpSession Contains the following methods for Attribute Mangmt

1. public void SetAttribute (String name, Object value)
2. public Object getAttribute (String name)
3. public void removeAttribute (String name)
4. public Enumeration getAttribute

Application Scope :-

→ We can maintain application Scope by using ServletContext Objects in Servlets & by application implicit Object in JSPs

→ The Information stored in application Scope is available for

all components of the web application.

→ Application Scope will be stored at the time of ServletContext Object creation (i.e at the time of application deployment) & will be lost at the time of Context Object Destruction. (i.e at the time of application undeployment).

→ ServletContext defines the following methods for application scope attribute mangmt.

setAttribute, getAttribute, removeAttribute, getAttributeNames()

Page Scope

→ This scope is applicable only for JSP's but not for Servlets.

→ This scope is maintained by Page Context implicit Object.

→ Information stored in PageScope is by default available with in the Translation unit only.

→ page scope is most commonly used scope in custom Tags to share Information btw Tag Handler classes.

→ PageContext Class Defines the following methods for attribute mgmt in PageScope.

1) Public void SetAttribute (String name, Object Value);

2) Public void removeAttribute (String name);

3) Public Object getAttribute (String name);

Extra methods in PageContext

→ PageContext class contain the following methods to perform Attribute Mgmt in any Scope.

(1) Public void setAttribute(String name, Object value, int Scope);

intScope
- PageContext.APPLICATION-SCOPE (or) ⇒ 4
- PageContext.SESSION-SCOPE ⇒ 3
- PageContext.REQUEST-SCOPE ⇒ 2
- PageContext.PAGE-SCOPE ⇒ 1

(2) Public Object getAttribute(String name, int Scope)

(3) Public void removeAttribute(String name, int Scope).

⇒ which of the following is valid way to store an attribute in Page Scope.

(1) ~~Page~~ pageContext.setAttribute("durga", "SCJP");

(2) PageContext.setAttribute("durga", "SCJP");

(3) PageContext.setAttribute("durga", "SCJP", 4); Appli SCP but we need PageScope

(4) PageContext.setAttribute("durga", "SCJP", 1);

(5) PageContext.setAttribute("durga", "SCJP", PageContext.PAGE-SCOPE)

→ PageContext defines the following extra methods also

1) public Enumeration getAttributeNamesInScope (int Scope)

2) public Object findAttribute (String name);

→ First it will searches in the pageScope If it is not available then it will search request, Session & application Scopes respectively. If it is not available it will say that the attribute is not available.

→ getAttribute(String name) → searches in only page Scope. If not it will say that, attribute is not available.

→ what is the diff btw getAttribute(String name) & getAttribute(String name, int Scope)

→ what is the diff btw getAttribute(String name) & findAttribute(String name)

write a JSP to display the no. of Session Objects Created at server side.

⇒ <%@ Page isELIgnored="false" %>

<%

Integer count = (Integer) application.getAttribute("sessionCount");

if(Session.isNew())

{ if(count==null)

count=1;

```
else
    count++;

PageContext.setAttribute("sessionScope", count);
}
```

The no. of Session Objects : ${ session.count }.
(or) <%= count %>.

24/05/09

Jsp Standard Actions.

Obj) for the given Design Problem write code by using the following Standard actions.

1. <jsp:useBean> with Attributes - Id, Type, Name, Scope.
2. <jsp:getProperty>
3. <jsp:setProperty>

⇒ we are generally using Scripting Elements in the Jsp. This approach is very easy approach at beginers level.

Demo.jsp

```
<%!  public int squareIt(int i)
     {
        return i*i;
     }
%>

<h1> The square of 4 is : <%= squareIt(4) %>
the square of 5 is : <%= squareIt(5) %>
</h1>
```

→ This approach has several serious Disadvantages.

(1) There is no clear seperation of presentation & Business Logic. The person who is writing this JSP should have compulsory the knowledge of both java & html. which may not possible always.

(2) This approach doesn't promote Reusability.

(3) It reduces Readability of code also.

⇒ we can resolve these problems by Encapsulating entire Business logic inside JavaBean.

CalculatorsBean.java

```
package pack1;
public class CalculatorsBean
{   public int squareIt(int i)
    {
        return i*i;
    }
}
```

BeanEx

- demo.jsp
- WEB-INF
  - classes
    - pack1
      - CalculatorsBean.java
      - .class

Demo.jsp

```
<jsp:useBean id="calc" class="pack1.CalculatorsBean" />

<h1> The square of 4 is <%= calc.squareIt(4) %>
     The square of 5 is <%= calc.squareIt(5) %>
</h1>
```

→ The advantages of this approach are

(1) Seperation of Presentation & Business Logic (Presentation logic is available in JSP & Business logic is availble in Bean class). so that readability of code has improved.

(2) Seperation of Respousabilities. Java Developer can concentrate on business logic where as Html Page Designer can concentrate on Presentation Logic. As both can work simultaneously so that we can reduce Project Development Time.

(3) It promotes Reusability of the code. wherever squcing functionality is required we can use the same Bean without rewriting.

⇒ we can purchase a bean for File Uploading & we can start uploading of files with in minutes.

Java Bean

→ Java Bean is a simple Java class. to use Bean inside JSP the Bean class has to follow the following rules.

(1) The class should contain Public no argument Constructor. Bcz JSP Engine always calling this constructor only for Instantiating the Bean.

(2) For every property Bean class should contain Public getter & setter methods. JSP Engine calls these methods to access or mutate (changing) Bean Properties.

① <jsp:useBean />

→ we have to use this Tag to make Bean Object available to the Jsp.

→ There are 2 forms of JSP useBean.

① without Body

```
<jsp:useBean id="calc" class="Pack1.CalculatorBean" />
```

② with Body

```
<jsp:useBean attribute list >
    ≡ Body
</jsp:useBean>.
```

→ The main objective of Body is to perform initialization for the newly created Bean Object.

→ If the Bean Object is already available then jsp:useBean won't create any new object, it will use existing object only. At this time Body won't be executed.

Attribute List of <jsp:useBean />

→ <jsp:useBean > can contain the following 5 Attributes.

① Id
② Class
③ Type
④ Scope
⑤ beanName.

(1) Id – This attribute is the name of the Reference Var of the Bean Object.

→ By means of this Id only we can access Bean in rest of JSP.

→ The Id attribute is Mandatory.

```
<jsp:useBean id="calc" class="Pack1.CalculatorBean" />
```

The equivalent Java Code is

```
CalculatorBean calc = new CalculatorBean();
```

Id → refers Bean Object.

(2) Class – This attribute specifies the FullyQualified Name for the Java Bean class.

→ JSP Engine, to perform Instantiation the Value of class Attribute should be Concrete class. Abstract classes & Interfaces are not allowed (bcz creation of obj is not Possible). new CBC();

→ The Bean Class should compulsory contain public No-Argument constructor otherwise we will get Instantiation Problem.

→ This attribute is optional & whenever we are not using class Attribute compulsory we should use type Attribute.

(3) type – This attribute can be used to specify the type of Reference Var.

→ The value of type Attribute can be Concrete class or Abstract class or Interface.

→ The type Attribute is optional but at that time compulsory we should specify class Attribute.

```
<jsp:useBean id="P" class="Employee" type="Person" />
```

⇓

```
Person P = new Employee();
```

type · Id · class

(4) Scope — This attribute specifies in which scope the JSP Engine has to search for the required Bean Object.

→ In that scope if the bean object is not already available then JSP Engine search will create a new bean object & stores that object in the scope for the future purpose.

→ The allowed values for the scope attribute are page, request, session & application.

→ The scope Attribute is optional & default value is page-scope.

```
<jsp:useBean id="calc" class="CustomersBean"
                                scope="request" />

CustomersBean Calc = null;
Calc=(CustomersBean) PageContext.getAttribute("calc", Calc
                                PageContex.REQUEST-SCOPE);
if(Calc==null)
        Calc = new CustomersBean();
PageContext.setAttribute("calc", Calc, PageContext.REQUEST-SCOPE
```

Synchronized method

→ when one is performing one changes others should not use this obj at the same time.

→ To use session scope compulsory session Object should be available in the JSP. (28)

<%@ Page session="false" %>

<jsp:useBean id="calc" class="CustomerBean" scope="session"/>

CTE → HTTP Status Code 500

— Illegal for useBean to use session scope when JSP page declare (via page Directive) that it does not participate in session.

(5) beanName — There may be a chance of using Serialized bean object from local file system. In that case we have to use beanName Attribute.

Various Possible Combinations

(1) Id Attribute is mandatory.

(2) scope is Optional & Default scope → PAGE-SCOPE.

(3) The Attributes class, type, beanName can be used in the following combinations.

(a) class

(b) type

(c) class, type

(d) type, beanName (beanName associate with type but not class)

→ If class Attribute is used whether we are using type Attribute or not, it should be Concrete class & should contain Public no-argument Constructor.

→ If we are using only type Attribute with out class Attribute Compulsory Bean Object should be available in specified scope. otherwise this Tag won't create any new Bean object & raises Instantiation Exception.

Ex: <jsp:useBean id="Calc" type="java.lang.Thread" />
— In this case, compulsory Thread Object should be available in page Scope otherwise we will get
java.lang.InstantiationException, bean calc not found with in scope.

⇒ Consider the class Package PACK1;
public class CustomersBean
{
}

→ Assume that no customersBean Object is already created. which of the following Standard Action creates a new Instance of this & Store in the RequestScope.

① <jsp:useBean name="cust" type="Pack1.customersBean" />
② <jsp:makeBean name="cust" type="Pack1.CustomersBean" />
③ <jsp:useBean id="cust" Class="Pack1.customersBean
Scope="request" />

<jsp:useBean id="cust" type="Pack1.CustomerBean" scope="request" />
object should be available already (Ques → obj not present)

JSP: getProperty

This Tag can be used to get the Properties of bean Object.

```
<%
Customer c=new Customer();
  out.println(c.getName());  Accessing & Printing
%>
```

⇓

```
<jsp:useBean id="c" class="CustomerBean" />
<jsp:getProperty name="c" Property="name" />  getting & Printing
```

<jsp:getProperty> contains the following 2 Attributes.

name – The name of the bean Instance from which the required property is obtained.

It is exactly equal to Id Attribute value of the <jsp:useBean>

Property – The name of the Java Bean Property which has to retrieve.

Both attributes are mandatory.

CustomerBean.java

```
Package Pack1;
Public class CustomerBean
{
    Private String name = "Durga";
    Private String mail = "info@durgasoft.com";
    Public String getName()
    {   return name;
    }
    Public String getMail()
    {   return mail;
    }
}
```

Demo.jsp

```
<jsp:useBean id="c" class="Pack1.CustomerBean" />
<h1>
  <jsp:getProperty name="c" Property="name" /> <br>
  <jsp:getProperty name="c" Property="mail" />
</h1>
```

o/p Durga
info@durgasoft.com.

JSP: SetProperty

→ This standard Action can be used to set properties of a bean object.

→ we can use <jsp:setProperty> in the following forms.

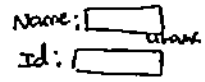
① <jsp:useBean id="c" class="Customer" />

<jsp:setProperty name="c" property="name" value="durga" />

⇓

c.setName("durga");

② <jsp:setProperty name="c" property="name" param="uname" />

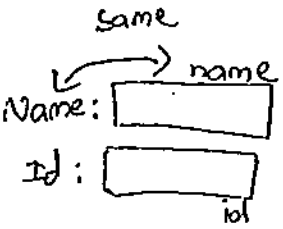


⇓

c.setName(request.getParameter("uname"));

→ It retrieves the value of specified request parameter & assign its value to the specified Bean property.

③ If the request parameter name matches with bean property name then no need to use param attribute.

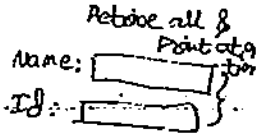


<jsp:setProperty name="c" property="name" />

⇓

c.setName(request.getParameter("name"));

④ <jsp:setProperty name="c" property="*" />



→ * specifies all properties of Bean.

→ It iterates through all request parameters & if any parameter

- name matches with bean property name then it assigns request parameter value to the bean property.

Attributes of <jsp:setProperty>

1. name — It refers name of the Bean Object whose property has to set.

→ This is exactly same as Id Attribute of <jsp:useBean>

→ It is mandatory Attribute.

2. Property — The name of the Java Bean Property which has to be set.

→ It is mandatory Attribute

3. Value — It specifies the value which has to set to the Java Bean Property.

→ It is optional Attribute & should not come in conjunction (combination) with "Param" Attribute.

4. Param — This attribute specifies the name of request parameter whose value has to set to the Bean Property.

→ It is optional Attribute & should not come in conjunction with 'value' Attribute.

login.html

```
<html>
<form action="test.jsp">
Name: <input type="text" name="name"> <br>
mail : <input type="text" name="mail"> <br>
age : <input type="text" name="age"> <br>
<input type= submit >
</form>
</html>
```

test.jsp

page Scope
→ changes Reflected

```
<jsp:useBean id="c1" class="Pack1.customerBean" scope="session">
<jsp:setProperty name="c1" Property="*" />
<jsp:setProperty name="c1" Property="name" />
</jsp:useBean>
```

→ no reflecti in same Browser with reload

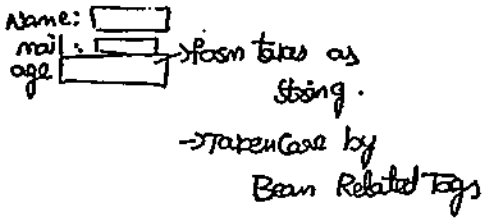
overriding with old value

```
<h1> Please check your values & confirm <br>
<jsp:getProperty name="c1" Property="name" /> <br>
<jsp:getProperty name="c1" Property="mail" /> <br>
<jsp:getProperty name="c1" Property="age" /> <br>
</h1>
```

```
package Pack1;
public class CustomersBean
{
    Private String name;
    Private String mail;
    Private String/int age;

    Public String getName()
    {
        return name;
    }
    Public String getMail()
    {
        return mail;
    }
    Public int getAge();
    {
        return age;
    }
    Public void SetName(String name)
    {
        this.name = name;
    }
    Public void setMail(String mail)
    {
        this.mail = mail;
    }
    Public void SetAge(int age)
    {
        this.age = age;
    }
}
```

Note

→ All the required Type Conversion (String → int or int → String) will take care by Bean Related Tags.



Developing Reusable Web Components

→ we can Develop reusable web components by using the following standard Actions.

(1) <jsp:include>

(2) <jsp:forward>

(3) <jsp:Param>

<jsp:include>

```
<jsp:include  Page="header.jsp"  flush="true" />
```

→ The response of included page will be placed in including page at Request Processing Time. Hence it is Dynamic Include.

→ This is the Tag Representation of PageContext.include()

→ This standard Action contains the following 2 Attributes.

(1) Page - Represents name of the included page & mandatory

(2) Flush - It determines whether the response will be flushed before Inclusion or not.

→ It is optional Attribute. & default value is "false."

header.jsp

masters in JCT

```
<jsp:include  Page="header.jsp" />
welcome to DSS
<jsp:Include  Page="footer.jsp" />
```

footer.jsp

Leaving Java is not enough

make use of that Subject is important.

→ In JSP's we can perform Include by following ways.

① Include Directive

```
<%@ include file="header.jsp" %>
```

② Include Action

```
<jsp:include page="header.jsp" flush="true" />
```

③ scriplet

```
<%
PageContext.include("header.jsp");
%>
```

④ By RequestDispatcher

scriplet

```
<%
    RD rd = request.getRD("header.jsp");
    rd.include(req, resp);
%>
```

10/09 <jsp:forward>

⇒ If first JSP is Responsible for some Preliminary Process & second JSP is responsible for providing complete response, then we should go for forward mechanism.

```
<jsp:forward page="second.jsp" />
```

first JSP

```
<h1> This is first JSP </h1>
<jsp:forward page="second.jsp" />
```

second.jsp

```
<h1> This is Second JSP </h1>
```

→ <jsp:Param>

→ while forwarding or Including we are allow to send parameters to the Target JSP we can achieve this by using <jsp:Param>

```
<jsp:Param name="c1" value="SCJP" />
```

→ <jsp:Param> contains the following 2 mandatory Attributes

① name – Its represents name of the parameter

② value – " " value " " "

→ The Parameters which are sending by using JSP:Param Tag are available as form Parameters in the Target JSP.

First JSP

```
<h1> This is first JSP </h1>
<jsp:forward page="second.jsp" >
    <jsp:Param name="c1" value="SCJP" />
    <jsp:Param name="c2" value="SCWCD" />
</jsp:forward>
```

⇒ send as Query String
second.jsp?c1=SCJP&c2=SCWCD

Second.jsp

```
<%@ Page IsELIgnored="false" %>
<h1> This is Second JSP . The Offered Courses are
    ${Param.c1} and ${Param.c2}
```

(or)

```
<%= request.getParameter("c1") %>
<%= request.getParameter("c2") %>
```

⇒ In JSP's the following are Various Techniques for forwarding a request from one JSP to another JSP.

① forward Standard Action

```
<jsp:forward  Page="Second.jsp" />
```

② By PageContext implicit Object

```
<%
  PageContext.forward("Second.jsp");
%>
```

③ By Request Dispatcher

```
<%
  RD rd = request.getRD("Second.jsp");
  rd.forward(request, response);
%>
```

Conclusions

case-1

```
<jsp:forward  Page="http:\\localhost:8080\JSP\Second.jsp" />
<jsp:Include  "          "          />
<%@ include file="http ---  \Second.jsp %>
```

Valid
X

→ The Value of file & Page Attributes should be Relative Paths only. we are not allow to use Server Port, Protocol etc. Otherwise we will get 404 - Status Code.

Case-2

✓ <jsp:forward page="/test" />

✓ <jsp:include page="/test" />

✗ <%@ include file="/test" %>

→ In the case of forward & Include Actions the attribute page can pointing to a servlet.

→ But in case of include Directive file attribute can't pointing to a servlet. It can point to JSP, Html, xml, xhtml etc.

Case-3

✓ <jsp:forward page="second.jsp?c1=SCJP" />

✓ <jsp:include page="second.jsp?c1=SCJP" />

✗ <%@ include file="second.jsp?c1=SCJP" %>

→ In the case of forward & Include Actions we are allow to pass query string.

→ But in the case of Include Directive we are not allow to pass query string.

Summary of JSP Standard Action

Standard Action	Description	Attribute List
1) <jsp:useBean>	To make Bean Object available to the JSP	id, class, type, scope beanName.
2) <jsp:getProperty>	To get & Print property of Bean	name, property
3) <jsp:setProperty>	To set Properties of the Bean	name, property, value, param
4) <jsp:include>	For including the response of Target JSP in current JSP Response at request processing time.	Page, flush
5) <jsp:forward>	To forward a request from one JSP to another JSP	Page
6) <jsp:param>	To send Parameters to the target JSP while Forwarding or Including	name, value.
7) <jsp:plugin>		
8) <jsp:Params>		
9) <jsp:fallback>		

JSP Document

→ There are 2 types of Syntaxs are possible to write JSP

① JSP Standard Syntax

② xml Based Syntax

→ If we are writing JSP by using JSP Standard Syntax, such type of JSP's are called JSP-Pages.

→ If we are Developing JSP's Based on xml Syntax such type of JSP's are called JSP-Documents.

① Scriplet

Standard	xml Syntax
<%	<jsp:scriplet>
≡	≡
%>	</jsp:scriplet>

② Expression

<%=	<jsp:Expression>	Part of Expression only X <j:e> ⊖ 2+3 </j:e>
≡	≡	
%>	</jsp:expression>	✓ <j:e> 2+3 </j:e>
✓ <%= 2+3 %>		

③ Declaration

<%!	<jsp:declaration>
≡	≡
%>	</jsp:declaration>

Directives

① <%@ page import="java.util.*" %>

→ xml Syntax <jsp:directive.page import="java.util.*"/>

② <%@ include file="header.jsp" %>

xml <jsp:directive.include file="header.jsp" />

③ <%@ taglib prefix="mime" uri="www.durgasoft.com" %>

X xml <jsp:directive.taglib prefix="mime" uri="www.durgasoft.com"/>

→ For the taglib directive there is no equivalent syntax in xml. we can provide its purpose by using <jsp:root> Tag

<jsp:root xmlns:mime="www.durgasoft.com" version="2.1">

</jsp:root>

Standard Actions

→ There is no difference in standard Actions representation btw standard & xml based syntaxes

Ex:- <jsp:useBean id="calc" class="Pack1.CalculatorBean"/>

Comments

<!-- xml Comment -->

→ JSP Specification does not provide any specific syntax for writing comment. Hence we can use normal xml comment syntax.

Template Text - JSP standard syntax doesn't provide any specific way to write template text. But xml based syntax provides a std tag <jsp:text> Tag <jsp:text> Template Text </jsp:text>

Element	Standard Syntax	xml Based Syntax
1) Scriplet	<% Any Java code %>	<jsp:scriplet> Any Java code </jsp:scriplet>
2) Expression	<%= Any Java Expression %>	<jsp:expression> 2+3 </jsp:expression>
3) Declaration	<%! Any Java Declaration eg:- int i=10; %>	<jsp:declaration> int i=10; </jsp:declaration>
4) Directives	Page <%@ Page import=java.util.*%>	<jsp:directive. page import=java.util.* />
	Include <%@ include file="header.jsp" %>	<jsp:directive.include file="header.jsp" />
	Taglib <%@ taglib Prefix="mine" uri="durgasoft.com" %>	No Equivalent Tag =>But we can achieve this by using <jsp:root>

5) Standard Actions <jsp:forward page="second.jsp"/> <jsp:forward page="second.jsp"/>

⇒ No Difference in syntax

6) Comments <%-- JSP comments --%> <!-- xml comment -->

7) Template Text No specific Syntax <jsp:text> Template Data </jsp:text>

How web Container Identifies JSP Document

1) Save the JSP file with .jspx extension

2) Enclose entire JSP in <jsp:root> Tag

3) use jsp-Config Element in web.xml -

```
<web-app>
    <jsp-config>
        <jsp-property-group>
            <url-pattern> /jspx/* </u-p>
            <jsp-xml> true </j-x>
        </j-p-g>
    </j-c>
```

Note

⇒ From servlet 2.4 version onwards web container can identify jsp Document automatically. No special arrangement is required.

⇒ write a Jsp-Document to point Hit count of JSP

JSP Page

```
<%!
int count=0;
%>
<%
 count++;
%>
The Hit Count is:
<%= count %>
```

.jsp
(or)
.jspx

JSP Document

```
<jsp:root version="2.1">
 <jsp:declaration>
      int count=0;
 </jsp-declaration>
 <jsp-scriplet>
      count++;
 </jsp-scriplet>
 <jsp-text>
      The hit count is:
 </jsp-text>
 <jsp-expression> count
      </jsp-expression>
</jsp:root>
```

→ Its not recommended to use both standard & xml based syntaxes simultaneously.

→ The main Advantage of xml-based syntax is, we can use xml editors like xml-spy for writing & debugging JSP's.

→ In xml-based syntax all Tags, Attributes are case-sensitive. Attributes values must be enclosed either in single quotes or in Double quotes.

⇒ which of the following is the valid way of importing util package in the JSP.

(1) <jsp:page import="java.util.*" />

(2) <jsp:directive page import=" " />

(3) <jsp:page.directive import=" " />

(4) <jsp:directive.page import=" " />

Elimination of Java Code ⟹

- EL
- JSTL
- Custom Tag
- Standard Actions

0
0
0

UNIT-3

Expression Language (EL)

→ It has Introduced in JSP 2.0 version. The main objective of EL is to eliminate Java code from the JSP.

→ In general we can use EL with JSTL & custom Tags for complete Elimination of Java code.

Examples

(1) To Point the value of request Parameter "uname"

Standard : <%= request.getParameter("uname"); %>

EL : ${ Param.uname }.

(2) To point the value of x Attribute

Standard : <%= PageContext.findAttribute("x") %>

EL : ${x}

→ To use any variable 'x' in EL compulsory it should be an attribute of some scope.

${x} => It Points the value of 'x' attribute & if there is no such Attribute, we will get blank space (in Standard Syntax Tags we will get NULL but EL supresses NULL & does not Point anything (i.e blank space).

EL Implicit Objects
EL operators
EL functions

10 to 12 Questions.

EL Implicit Objects

EL contains 11 Implicit Objects. The power of EL is just because of these implicit objects only.

The following is the list of all EL Implicit Objects

String	String or String[ ]
key	value
x	SCJP
y	{SCJP, SCWCD}

1) pageScope
2) Application Scope
3) Request Scope
4) Session Scope

} A map of Scoped Attributes

5) param
6) paramValues

} Map of request parameters

7) header
8) headerValues

} Map of request headers

9) cookie — map of cookies.

10) initParam → A map of Context Initialization parameters (but not servlet Initialization parameters)

11) pageContext

Page Scope, request Scope, Session Scope, Application Scope (39)

→ By using these implicit Objects we can retrive the attributes of a particular Scope.

→ PageScope ⇒ PageContext.getAttribute()

requestScope ⇒ request.getAttribute()

SessionScope ⇒ Session.getAttribute()

applicationScope ⇒ application.getAttribute()

(1) To Access the value of Session Scoped Attribute x.

${sessionscope.x}

(2) ${requestScope.x} – It Points the value of request Scoped attribute 'x'. In request scope if there is no such Attribute we will get blank space

(3) ${x}

→ JSP Containes first checks in PageScope for Attribute "x". If it is available it Points its value. If it is not available it will check in request Scope followed by session & application Scopes. It simply acts as PageContext.findAttribute().

```
<%@ page isELIgnored="false" %>
<%
    PageContext.setAttribute("x", "SCJP", 3);
    Page Context.setAttribute("x", "SCWCD", 1);
%>
```

The value of attribute is : ${x} SCWCD

param & paramValues

By using these implicit Object we can retrive request Parameters.

param => request.getParameter()

paramValues => request.getParameterValues()

${ param.x } - It Points the value of formParameter "x".

If the specified parameter is not already available then we will get blank Space.

If Parameter associated with mutiple values then we will get first value.

param

String	String
Pname	PValue
uname	Durga
uadd	SRNbgar
⋮	

paramValues

String	String[ ]
Pname	PValue
uname	{durga}
uadd	[SRNbgar, Ameerpet]
⋮	

${ paramValues.x }

on String[] to String() will be called.

${paramValues.x[0]} — Points first value of x

${paramValues.x[1]} — Points second value of x.

${paramValues.x[10]} Points blank space

Name: [durga] name

mail : [durgatech@gmail.com] mail

food1 : [Dosha] food

food2 : [Pori] food

[Submit]

login.html

```
<html>
  <form action="test.jsp">
    Enter Name : <input type=text name="name"> <br>
    Enter mail : <input type=text name="mail"> <br>
    Enter Food1 : <input type=text name="food"> <br>
    Enter Food2 : <input type=text name="food"> <br>

    <input type=submit>
  </form>
</html>
```

test.jsp

<%@ page isELIgnored = "false" %>

${Param.name} durga

${Param.mail} durgatoc@gmail.com

${Param.food} Dosha

${Param.age} blank space

${ParamValues.name} [java.lang.@12345]

${ParamValues.name[0]} durga

${ParamValues.name[1]} blank space

${ParamValues.food[0]} Dosha

${ParamValues.food[1]} puri

${ParamValues.food} java.lang@15698

EL handles "null" & ArrayIndexOutofBounds Exceptions very nicely. It supresses them & prints blank space

for the QueryString

?fname = pawan & lname = kalyan. what is the O/P for the following code.

${Param.fname} Pawan

${Param.mname} blank space

${Param.lname} Kalyan

${paramValues.lname[0]} Kalyan

${paramValues.lname[1]} blankSpace

${paramValues.lname} toString() will be called
java.lang @ 156789.

headers & headersValues

These are exactly similar to param & paramValues. except these are for retriving requestHeaders.

headers => request.getHeaders();

headersValues => request.getHeaders();

${headers.accept} */*
${headersValues.accept} toString()
${headers.accept[2]} blankSpace

cookie

By using this implicit object we can access cookies associated with the request object.

cookie => request.getCookies();

name	Value
JSESSIONID	

${cookie.JSESSIONID.name}

${cookie.JSESSIONID.value}

${cookie.JSESSIONID} toString() get called
javax.servlet.http.Cookie @ 15947

initParam

- By using this implicit object we can access ServletContext initParameters.

  initParam => getServletContext.getInitParameters();

- ${initParam.mail}

  ${initParam.name}

<web.xml>

```
<web-app>
    <context-Param>
        <P        mail
                  durgasoft.com
```

- If the specified initParameters is not available then we will get blank space bcz EL supresses 'null'.

pageContext

- This is the only one EL implicit object which matches with JSP Implicit objects.
- This is the only one EL Implicit object which is a non-map object.
- By using this implicit object we can access all other JSP implicit objects in EL.

request.getMethod();

request, session implicit Objects are not available in EL

→ ✗ ${request.method}

✓ ${pageContext.request.method} GET

✓ ${pageContext.session.id} 1234Sabc E8

✗ ${session.id}

EL Operators

→ EL contains its own specific operators. The following is the list of all possible operators.

1. Property Access operator (.)
2. collection Access operator ([ ])
3. Arithmatic operators
4. Relational operators
5. logical operators.

Property Access operator (.)

${ left variable . right variable }

↓ map (or) Bean

↓ mapkey (or) Property } should be a valid Java Identifier.

${custom.name}

${pageContext.request.method}

Bean

${initParam.mail}
map — name of map

X${customer.1234}
should be identifier

Collection Access Operator ([])

key	value
Student	Roll-no.

${leftVariable[rightVariable]}

map ———— mapkey
Bean ———— Property
} should be enclosed either in single or Double Quotes

Array ———— Index
List ———— Index
} Quotes are optional.

→ Map

${initParam['mail']}

${initParam["mail"]}

${initParam[mail]} blank space.

→ If we are not keeping Quotes EL assumes that the key is an attribute stored in some scope. If that attribute is not available then we will get blank space as o/p.

→ which of the following is the valid way of retriving Accept Request Header.

1) ${ header.accept }

2) ${ header['accept'] } blank space as o/p

3) ${ header ["accept"] }

4) ${ header ['accept'] }

Bean

1) ${ customer['name'] }

2) ${ customer ["name"] }

3) ${ customer [name] } blank space as o/p.

Arrays

```
<%
    String[] s = {"A", "B", "C"};
    PageContext.setAttribute ("s", s);
%>
```

${ s[0] } A ${ s[1] } B ${ s[2] } C

${ s[10] } blank space.

List

```
<%@ page import="java.util.*" %>
<%
    LinkedList l = new LinkedList();
```

```
l.add("congress");
l.add("LS");
l.add("tdp");
l.add("pop");

PageContext.setAttribute("list", l, 3);
%> PageContext.setAttribute("index", 3);
```

${ list[2] } tdp ${ list['0'] } congress

${ list['3'] } pop ${ list[index] } pop.

⇒ wherever property Access operator is required there we can use collection Access operator. But wherever collection Access operator is required we may not use property Access operator.

Arithmatic Operators

① + operator

→ EL doesn't support overloading & hence there is no string concatenation operator. "+" always acts as addition operator only.

→ ${ 2+3 } 5 ${ "abc" + "3" } NumberFormatException

${ "2" + 3 } 5 ${ " " + "3" } NumberFormatException

${ "2" + '3' } 5 ${ null + "3" } 3

0+3=3

${ abc + "3" } 3

2) "−" operator ${a-b}

→ All rules are exactly similar to '+' operator.

${"10"-3}

${"abc"-3}

3) "*" operator ${a*b}

→ All rules are exactly similar to "+" operator.

${"10"*3} ⇒ 30

4) "/" operator (or) div

→ All rules are exactly similar to "+" operator & Division operator always follows floating point Arithmatic.

${10/0} Infinity

${0/0} NAN

${10/2} or ${10 div 2} 5.0

5) % operator (or) mod

→ In the case of mod operator both floating point & integral arithmatic are possible.

${10%0} ArithmaticException.

${10%3} 1

${10.0%0} NaN

Relational Operators

> or gt == (or) eq <= (or) le

< or lt >= (or) ge != (or) ne

${ 3<4 } True

${ abcd ==abcd } True

Logical Operators

→ These are 3 Logical operators.

& (or) && and ${ T && True } True

| (or) || or ${ !true } false.

! (or) ! not

Conditional Operators

${ (3<4) ? "yes" : "no" } yes

${ (true==false) ? right : wrong } wrong

Empty Operator

${ empty object }

true → if object doesn't exist (or)

object is an Empty Array (or)

" " " " String (or)

" " " " Collection

Otherwise returns false.

${empty test} true

${empty "durga"} false

${empty null} true

EL Operator Precedence

1) Unary operators (! & empty)

2) *, /, %

+, -

3) Relational operators

4) logical operators - &&, ||

5) conditional operators.

EL Reserved words

true	instanceOf	and
false	gt	or
null	lt	mod
empty	ge	div
	le	
	ne	not
	eq	

EL treats null

EL behaves very nicely with null.

In Arithmatic operations null is treated as "zero".

with in String Evaluations null is treated as "empty string".

→ with in logical operator null is evaluated as false.

EL Functions

→ The main objective of EL is to seperate Java code from the JSP. If we are having any business functionality we can seperate it to a Java class & we can invoke its functionality through EL syntax. The page designer has to known only function name & tld file uri to get its functionality. Hence EL is almost considered as alternative to customTags.

Designing EL function Application contains the following stages

1. writing a Java class with required functionality.
2. writing tld file to map Java class to JSP.
3. write a taglib Directive to make business functionality available to JSP.
4. write EL function call.

Writing a Java class

→ Any Java class can simply acts as repository for EL Functions. The only requirement of a method that acts as EL-function is it should be declared as Public & static.

→ method can take parameters.

→ void return Type methods are not recommended to use as EL Functions eventhough it is legal. (for closing DBase operations).

```
Package Pack1;

Public class DiceRoller
{
    Public static int rollDice()
    {
        return (int) (Math.Random() *6 +1);
    }
}
```

Writing Taglibrary Descriptor File.

→ For EL functions Tld File provides mapping btw Java class (whose functionality is available) & JSP (which calls the functionality

→ we can configure EL function by using <function> Tag in the tld.

→ It contains 4 SubTags.

1. <description>
2. <name> – By means of this name only we can call EL functionality in JSP.
3. <function-class> The fully Qualified name of Java class where EL function is defined.
4. <function-Signature> The Signature of EL Function which is defined in Java class.

→ Here we have to provide return type, method-name & Argument List.

tld file Myfunction.tld

```
<taglib version="2.1" >
    <tlib-version> 1.2 </tlib-version>
    <uri> DiceFunctions </uri>
    <function>
        <name> rollIt </name>
        <function-class> Pack1.DiceRoller <
        <function-signature> int rollDice() <
    </function>
</taglib>
```

JSP Version

Developers Version (we can give any version)

→ By means of uri, we can make EL functionality available in JSP.

Writing Taglib Directive.

```
<%@ taglib prefix="mine" uri="DiceFunctions" %>
```

Invoking EL Function

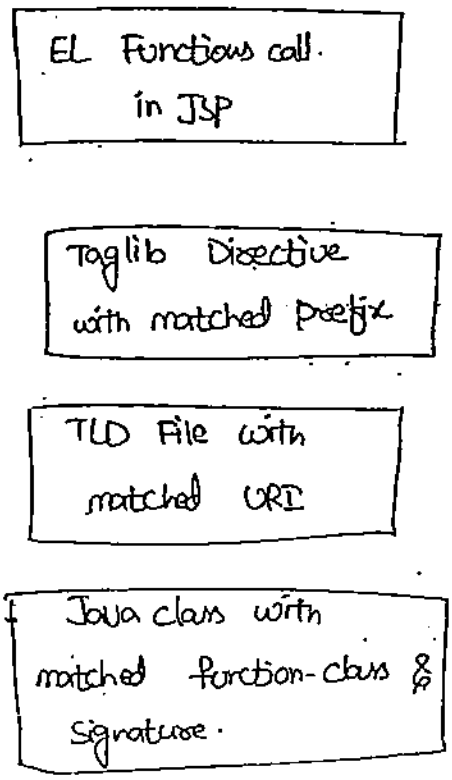
```
${ mine : rollIt() }
```

test.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib prefix="mine" uri="DiceFunctions" %>
<h1> Your 3-digit Lucky Number is :
```

${mine:roll()} ${mine:roll()} ${mine:roll()} . (47)

Work Flow of EL Function Execution.



whenever JSP Engine encounters EL function call with prefix & EL function name then it checks for corresponding Taglib Directive with matched prefix.

From Taglib Directive JSP Engine identifies uri & checks for tld file with matched uri.

From the tld file JSP Engine checks the corresponding class & required method.

JSP Engine executes that method & return its result to JSP.

write a program to invoke math class square root() as EL Function call with method name m1().

test.jsp

```
<%@ Page isELIgnored="false" %>

<%@ taglib prefix="mime" uri = mathFunctions %>

The Square Root of 4 is : ${ mime : m1(4) }
The Square Root of 5 is : ${ mime : m1(5) }
```

myFunction.tld

```
<taglib version="2.1" >
  <tlib-version> 1.9 </tlib-version>
  <uri > mathFunctions </uri>
  <function>
      <name> m1 </name>
      <function-class > java.lang.math </function-class>
      <function-Signature> double sqrt(double) </f-s>
  </function>
</taglib>
```

```
EL1
 |- test.jsp
 |- WEB-INF
    |- myFunction.tld
```

11/5/04

JSTL (JSP Standard Tag Library) 1.2 & V

(48)

→ Sun People Encapsulates the core Functionality which is common to many web Applications in the form of JSTL. Programmer can use this predefined Library with out writing on his own.

→ The main objective of EL is removing Java code from the JSP. But it fails to replace Java code which processes some functionality. we can recover this by using JSTL. Hence the main objective of JSTL also for removing Java Code from the JSP.

→ Entire JSTL Library Dividing in to 5 sub parts.

1) Core Library – It defines some standard Actions to perform Programming general stuff like Conditions & loops. It can also perform JSP Fundamental Tasks such as setting Attributes, writing output & ReDirecting the request to other Pages.

2) xml – Defines several standard Actions useful for writing & Formating xml Data.

3) Fmt (formating) – Defines several standard Actions which can be used for Formating Numbers & Date in the Process of Internationalization.

4) Sql – Defines several standard Actions can be used for

Data Base operations.

3) Functional – Defines several standard Actions can be used for manipulating collection & string object

Core Library

⇒ JSTL Core Library Divided in to the following 4 parts based on Functionality.

1) General purpose Tags

<c:out>

<c:set>

<c:remove>

<c:catch>

2) Conditional Tags

<c:if>

<c:choose>

<c:when>

<c:otherwise>

③ Iteration

<c:foreach>

<c:forTokens>

④ Url Related Tags

<c:import>

<c:url>

<c:redirect>

<c:param>

Installing JSTL

⇒ By default JSTL Functionality is not available to the JSP. we can provide JSTL Functionality by placing the following jar Files in web Application Lib folder.

1. jstl.jar – Defines several API classes which are Defined by sun.

2. standard.jar – Provides library implementation classes by vendor.

⇒ It is recommended to place these Jar Files At Server Level (c:\Tomcat\lib folder) instead of Application Level.

⇒ To make core Library available to JSP we have to Declare taglib directive as follows.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

.. General Purpose Tags

<c:out>

```
<c:out>
 prefix   name of Tag
```

This Tag can be used for writing Template Data & Expressions to the JSP.

Form-1

```
< c:out value="durga" />
```

It Points Durga to JSP.

Form-1

<c:out value="${param.uname}" />

→ It Points the value of request parameter uname to the JSP.

→ If the main value is not available or it evaluates to null then we can provide default value by using default attribute.

Example → If we give both req param & default ⇒ req param will be placed in JSP

JSP/date.jsp ?uname=durga Query String

<c:out value="${param.uname}" default="Guest" />

→ If we are not passing request parameter uname then we will get default value guest.

→ <c:out> defines the following 3 Attributes

① value – It is mandatory attribute & it is for providing the value.

→ It should be either literal or RunTime Expression

② default – It is optional Attribute & it is to provide default value.

→ JSP Engine considers its value if & only if the value Attribute evaluates to "NULL".

③ escapeXml.

<c:set>

Form-1

we can use <c:set> to set Attribute in any scope.

```
<c:set var="name of Attribute" value="AttributeValue"
                                scope="page" />.
```

Scope Attribute is optional & default scope is pageScope.

```
<%@ taglib prefix="c" uri="http://java.sun.com/JSP/jstl/core"%>

<c:set var="x" value="10" scope="session" />
<c:set var="y" value="20" scope="session" />
<c:set var="result" value="${x+y}" scope="application" />
<h1>The result is : <c:out value="${result}" />
```

The result is : 30

form-2

we can set map or Bean property by using <c:set>
we can specify Bean or map Object by using Target Attribute & property name by using property Attribute.

```
<c:set target="customer" property="name" value="pavan" />
```

The attributes for the <c:set> are

var ③ Scope ⑤ property.
value ④ Target

<c:remove>

- This Tag can be used for Removing Attributes in the specified scope.
- This Tag contains the following 2 Attributes.

) Var – name of Attribute.

) scope – The scope in which Attribute present.

```
<c:remove var="x" scope="session" />

<c:remove var="x" />
```

- If the scope is not specified then the containers will search in the page scope first for the required Attribute. If the Attribute is not available then it will search request Scope followed by session & application scopes.

```
<%@ taglib prefix="c" uri="
<%@ page isELIgnored="false" %>

<c:set var="x" value="10" scope="session" />
<c:set var="y" value="20" scope="session" />
<c:set var="result value="${x*y}" scope="application"/>
<h1> The result is : <c:out value="${result}" />
<c:remove var="x" />
<c:remove var="y" />
```

```
<c:remove  value="result" />

<h1> The result is : <c:out value="${result}" default="8888" />
```

The result is : 20

The result is : 8888

<c:catch>

This can be used to catch an Exception with in the JSP instead of forwarding to the Error page.

The Risky code we have to place as the body of <c:catch>

```
<c:catch>
    =
  Risky code
    =
</c:catch>
```

If an Exception raised in the Risky code this Tag supresses that exception & rest of the JSP will be executed normally.

If an Exception is raised we can hold that by using var attribute which is page scoped Attribute.

```
<%@ page  isELIgnored="false" %>

<%@ taglib  prefix="c"  uri="

<h1> UserName is : ${ param.uname } <br>

<c:catch  var="e" >
```

```
<%.
  int age=Integer.parseInt(request.getParameter("uname"));
%>                                  -> only one catch Tag
  Age :${Param.uage} <br>             is Enough.
</c:catch>
<c:if test="${e !=null}" >
   oops ---Exception raised : ${e} <br>
   Height : ${Param.uheight}.
```

http://localhost:8080/jsp1/catch1.jsp?uname=durga&uage=20&uheight=5.3

UsersName : durga

Age : 20

Height : 5.3

http://localhost ---- uname=durga & uage=ten & uheight=5.3

usersName : durga

oops ---Exception raised : java.lang.NumberFormatException;
For input string="ten".

Height= 5.3.

Summary of General Purpose Tag.

Tag	Description	Attribute
1) <c:out>	For writing Expression & Template Text to JSP	Value, default, escapeXml.
2) <c:set>	To set Attributes, Bean or map properties	Var, Value, Scope target, property
3) <c:remove>	For removing Attributes	Var, Scope
4) <c:catch>	For supressing an Exception & to continue rest of JSP Normally.	Var.

Conditional Tags

1) <c:if>

→ This is for implementing case Java if Statement.

→ There are 2 forms available for <c:if>

<u>without Body</u>

<c:if test = test-condition var="x" scope="request" />

→ In this case test-condition will be evaluated & store its result into variable 'x'. In the rest of the page where ever this test

Condition is required we can use directly its value without evaluating once again.

→ In this case both test & var attributes are mandatory. Scope Attribute is optional & Default value is page.

with body

for future purpose we can save.

```
<c:if test=test-condition var="x" scope="request">
    ≡ Body
</c:if>
```

→ If test-condition is true then body will be executed otherwise without executing body rest of the JSP will be continued.

→ In this case also we can store test result inside var Attribute

→ Here both var & scope Attributes are optional but Test attribute is mandatory.

```
<%@ Page isELIgnored
<%@ taglib Prefix="c" uri= —
<c:set var="x" Value="10" scope="request" />
```

compulsory single quote

```
<c:if test="${x eq '10'}" var="y" scope="Session"/>
<h1> X value is : ${x}        XValue is : 10
     Test Result is : ${y}    Test Result is : true
```

<c:choose> & <c:when> & <c:otherwise>

We can use these Tags for implementing if-else & switch stmts.

Implementing if-else

JSTL doesn't contain any Tag for else. We can implement if-else stmt by using above Tags.

```
<c:choose>
    if { <c:when test="test-condition">
             ≡ Action 1
         </c:when>

    else { <c:otherwise>
             ≡ Action 2
         </c:otherwise>

</c:choose>
```

If test-condition is true, Action-1 will be executed otherwise Action-2 will be executed.

Implementing Switch Stmt

```
<c:choose>
    <c:when test="test-condition 1">
        Action 1          implicit break Stmt; available.
    </c:when>
    <c:when test="test-condition 2">
        Action 2
    </c:when>
    <c:when test="test-condition 3">
        Action 3
    </c:when>
    <c:otherwise>
        default-Action
    </c:otherwise>
</c:choose>
```

→ <c:choose> should compulsory contain atleast one <c:when>, <c:otherwise> is optional.

→ ~~when~~ Every <c:when> implicitly contains break Stmt. Hence there is no chance of fall through inside switch.

→ we have to Place <c:otherwise> as last case only.

→ <c:choose> & <c:otherwise> won't take any Attributes.

⇒ <c:when> contains only one mandatory Attribute which is test.

Switch.jsp

```
<%@ page isELIgnored
<%@ taglib prefix="c" uri=
<h1> Select the Number :
<form action="/jsp1/switch.jsp">
<select name="numbo">
    <option value="1"> 1 </option>
    <  "  value="2"> 2  "
    <  "  value="3"> 3  "
    <  "  value="4"> 4  "
    <  "  value="5"> 5  "
    <  "  value="6"> 6  "
    <  "  value="7"> 7  "
</select>
<input type="submit" />
</form>
```

```
<c:set var="s" value="${param.combo}" />
Today is:
<c:choose>
  <c:when test="${s==1}">Sunday</c:when>
  <  "  test="${s==2}">Monday  "
  <  "  test="${s==3}">Tuesday  "
  <  "  test="${s==4}">Wednesday  "
  <  "  test="${s==5}">Thursday  "
  <c:otherwise> Select between 1 & 5 </c:otherwise>
</c:choose>
</h1>
```

Summary of Conditional Tag

	Tag	Description	Attributes.
1	<c:if>	To implement Core Java if-Stmt.	test, var, scope
2	<c:choose>, <c:when>, <c:otherwise>	To implement if-else & Switch Stmt	<c:choose> & <c:otherwise> doesn't contain any Attribute but <c:when> should contain only one Attribute i.e test.

Iteration Tags

<c:forEach>

There are 2 Iteration Actions available in JSTL.

<c:forEach> – General Purpose for loop.

<c:forTokens> – Specialized for spliting strings similar to string Tokenizer.

<c:forEach>

Form-1

```
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri=
<c:forEach begin="0" end="4" step="1" >
  <h1> Learning JSTL is very Easy </h1>
</c:forEach>
```

o/p – Learning JSTL is very Easy will be printed 5 times

→ begin="0" end="10" step="2" 6 Times will be executed

→ begin="4" end="0" step="-1"

The begin Attribute Specifies the Index where the loop has to Start.

End Index Specifies the index where the loop has to Terminate

This loop internally maintain counter variable which is incremented by step Attribute Value.

The Default value for Step Attribute is "1" & it is optional Attribute.

Form-2 <c:forEach> with Var Attribute

→ <c:forEach> internally maintains a counter variable which can be Accessed by using Var Attribute.

→ This variable is local variable, from outside of forloop we can't Access.

```
<c:forEach begin="0" end="7" step="2" var="counter">
  <h1> Learning JSTL is very Easy --- ${counter} </h1>
</c:forEach>
```

O/p-Learning JSTL is very Easy ---- 0
---- 2
---- 4
---- 6

Form-3 <c:forEach> for Iterating through Arrays & Collections.

→ <c:forEach items="collection Object" var="current Object">

≡ Body

</ c:forEach>

→ items Attribute should contain either collection Object or Array

This Action will iterate over each item in the collection until all elements completion.

→ we can represent current collection Object by using Var.

Type of items Attribute	corresponding Var Attribute
① Primitive Array	corresponding Wrapper class
Ex: int[]	Integer
double[]	Double.
② object type Array	corresponding Declared Type
Ex: student[]	student
③ collection	Object
④ Map	map. Entry
⑤ List of Strings seperated by ,	String.

Example

```
<%
    String[] s={"A","B","C","D"};
    PageContext.setAttribute ("s", s);
%>
<c:forEach items="${s}" var="obj" >
  <h1> The Current Object is : ${obj} </h1>
</c:forEach >
```

The current Object is : A
" " : B
" " : C
" " : D

~~Request~~

write a Program to Display all request headers by using <c:forEach>

```
<h1>
 <table border='2' >

 <c:forEach  items="${headers}"  var="hdr" >
  <tr>
   <td> ${hdr.key }  </td>
   <td> ${hdr.value} </td>
  </tr>
 </c:forEach >
 </table  </h1>
```

K	V

map.Entry → getKey
getValue
setValue.

→ To Display all Cookies Associated with the Request

```
<h1>
<table border ="2" >

<c:forEach  items ="${cookie}"
                    var="x" >
 <tr>
  <td> ${ x.value.name } </td>
  <td> ${ x.value.value } </td>
 </tr>
</c:forEach> </table > </h1>
```

key	Value
cookie name	cookie object

on this particular object we have to get cookie name & cookie value.

→ if we call

x.key	x.value
JSessionID	toString() will be called.

Form-4 <c:forEach> with varStatus Attribute.

-> This Attribute describes the status of Iteration like current iteration Number, is it first Iteration or not etc.

-> This Attribute is of type javax.servlet.jsp.core.TagLoopStatus

-> This class contains several methods which are useful during Iteration.

1. Object getCurrent()
   - Returns the current Item.
2. int getIndex()
   - Returns the current Index (counter value)
3. int getCount()
   - Returns the no. of iterations that have already performed.
4. boolean isFirst()
   - Returns true if the current Iteration is First Iteration.
5. boolean isLast()
   - Returns true if the current Iteration is last Iteration.
6. Integer getBegin()
   - Returns Start Index.
7. Integer getEnd()
   - Returns last Index.
8. Integer getStep()
   - Returns Incremented value.

```
<c:forEach items="durga, pavan, ravi, shiva" varStatus="status">

<h1> Is it The First Iteration : ${status.first} <br>
  The current Object is : ${status.current} </h1>

</c:forEach>
```

```
o/p:  Is it First Iteration : True          -> false
      The current Object is : durga          ravi
      Is it First Iteration : false          -> false
      The current Object is : pavan          shiva
```

5/09 <c:forTokens> It is a specialised version of forEach to perform string Tokenization based on some delimeters. It behaves exactly same as string Tokenization.

Syntax <c:forTokens items="list of strings" delims="seperators sequence">
≡ Body.
</c:forTokens>

```
<c:forTokens items="List of Strings" delims="seperators sequence">

<c:forTokens items="one, two, three, four" delims=","  var="x">

  <h1> The current Token is : ${x} </h1>

</c:forTokens>
```

we can store the current Token by using var Attribute.

<c:forTokens> can Access (or) Accept the following extra Attributes also.

1) begin – Specifies the Index at which the iteration should start
The Index of first Token is zero.

2) End – Specifies the Index where the iteration should Terminate.

3) step – counter Increment Value btw iterations.

4) varStatus – to specify the status of Iteration.

Eg-1

<c:forTokens items="Congress ; TDP ; PR ; TRS" delims=";" >
No Party is good.
o/p – No Party is good → 4 times
</c:forTokens>

<c:forTokens items="chiru, nts, Pawan, roja" delims="," var="x">
Cine Industry is too good than Politics : ${x}
</c:forTokens>

<c:forTokens items="one, two, three, four, five" delims="," var="x"
begin="0" end="3" step="2" >
currentToken is : ${x}
.. o/p: currentToken is : one
current Token is : Three.
</c:forTokens>

In the case of <c:forTokens> items Attribute should be String only. But in the case of <c:forEach> the items Attribute can be collection, Array, Map or String. Hence <c:forTokens> is considered as Specialized Version of <c:forEach>

Summary of Iteration Tags

Tag	Description	Attributes
1) <c:forEach>	General purpose for loop	items, begin, end, step, var, varStatus.
2) <c:forTokens>	Specialized for string Tokenization	items, delims, begin, end, step, var, varStatus.

URL Related Tags

1) <c:import>

→ This is used for importing the response of other pages in to current page at Request processing Time (i.e Dynamic Include)

Form-1

<c:import url="second.jsp" />

↓

absolute | Relative.

c: import ⇒ c:include.

first.jsp

```
<%@ taglib prefix="c" uri=  %>
<%@ page isELIgnored = "false" %>

<c:import url="second.jsp" />
<h1> This is First JSP </h1>
```

O/P - This is Second JSP
This is First JSP

Second.JSP

```
<h1> This is Second JSP </h1>
```

Form-2

→ we can import the resources from outside of current Application also (i.e cross context import is also possible).

```
<c:import  url="/second.jsp"  context="/webapplic2" />
```

Not possible in Tomcat server

↓ Absolute path

↓ Absolute path.

first.jsp test <c:import url="/test.jsp" context="/JSP2" /> JSP1 second.jsp JSP2 context.jsp.

Form-3

→ we can store the result of imported page in to a variable specified by var Attribute. In the rest of the page where ever that result is required we can use directly the variable with out performing import once again.

```
<c:import  url="second.JSP"  var="x"  scope="page" />
<%
    String s=(String)PageContext.getAttribute("x");
    String s1 = s.subString (0,10);
%>
<h1> The Result is :${x};
    The part of subString is : <%= s1 %>
```

This is Second JSP

This

Form-4

→ The more convinent way to store the result of <c:import> is to use Reader Object. It is alternative to var Attribute.

```
<c:import    url="second.jsp"  varReader="myReader"
                                  scope="page" />

<%
Reader r=(Reader) pageContext.getAttribute("myReader");
    int i = r.read();
while (i != -1)
{
   // perform some operation on i          o.print(i);

    i = r.read();
}
%>
```

Form-5

while performing import we can send Form Parameters also to the se Target JSP. For this we have to use <c:Param> Tag. These parameters are available as Form parameters in the Target JSP.

first.jsp

```
<%@ taglib prefix="c" uri="  ---  " %>
<h1> welcome to DSS </h1>
<c:import  url="second.jsp" >
    <c:param  name="p1"  value="SCJP" />
    <c:param  name="p2"  value="SCWCD" />
</c:import>
```

Second.jsp

```
<%@ page isELIgnored="false" %>
<h1> The courses offered are : ${Param.P1} and ${Param.P2}
```

<c:redirect>

→ This Action can be used to redirect the request to another page. This is exactly similar to sendRedirect() of ServletResponse.

Form-1

```
<c:redirect  url="second.jsp" />
```

↓ Absolute or relative

Form-2

→ we can redirect the request to some other web Application resource also.

```
<c:redirect  url="/second.jsp"  context="/webapplic2" />
```

↓ Compulsory Absolute path ↓ Compulsory Absolute path

Form-3

→ while performing redirection we can pass parameters also to Target Resource.

```
<c:redirect  url="second.jsp" />
    <c:Param  name="x"  value="10"/>
    <c:Param  name="y"  value="20"/>
</c:redirect>
```

<c:url>

we can use this standard Action to rewrite url's to append session Information & form parameters.

Form-1

```
<c:url value="second.jsp" var="x" scope="request" />
```

→ modified url will be stored in x.

form-2

```
<c:url value="/Secon.jsp" context="/webapp2" var="x"
                                scope="request" />
```

form-3

```
<c:url value="second.jsp" var="x" scope="request" >
    <c:param name="p1" value="10" />
    <c:param name="p2" value="20" />
</c:url>
```

first.JSP

```
<c:url value="second.jsp" var="x" >
    <c:param name="p1" value="10" />
    <c:param name="p2" value="20" />
</c:url>
<h1> The modified url is : ${x}
<a href="${x}" > Click here to go to next page </a>
```

Second.jsp → Disable cookies.


```
<%@ Page isELIgnored="false" %>
<h1> This is second JSP
The Parameters are : ${ param.p1 } and ${ param.p2 }.
```

Summary of url related Tags.

	Tag	Description	Attributes
1	<c:import>	For importing the Response of other Pages at Request processing Time.	url, var, scope, varReader, context
2	<c:redirect>	To Redirect the request to other web components.	url, context,
3	<c:url>	To Rewrite url for appending session Information & Parameters (from).	value, var, scope context.
4	<c:param>	To send Parameters while importing & redirecting	name, value.

Design Pattern

A software Design Pattern is a repetable Solution for a commonly occuring software problem. It Provides a standard Process to Design Application.

The following are most commonly used Design Patterns.

1) Transfer Object Design Pattern.
2) Intercepting Filter " "
3) Front - Controller " "
4) MVC " "
5) Business Deligate " "
6) Service Locator " "

Data Transfer Object Design Pattern

Context - In Distributed Applications client & Servers are locating at remote locations and they have to communicate via network.

problem - Every call btw client & Server is a Remote method call. If client Application calls individual getter & setter methods to retrive & update Attribute values. then the no. of Remote calls required is as many as the no. of Attributes.

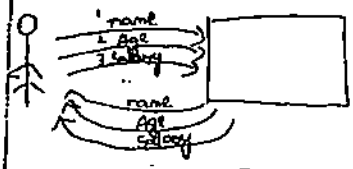
→ These individual remote calls generate lot of network Traffic & Degrade Performance of the System.

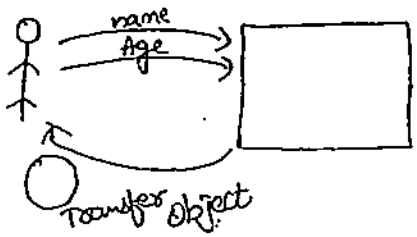
solution - create an Object to encapsulate all Attribute Values. that are required by client Application. This Object is called Data Transfer Object.

→ when the client request Data from Server, Server side Component gathers Data Values & construct Transfer Object by setting its Data Values. This Object is then sent to the client.

→ client use this Object to query all the required Data Values.

→ The Transfer Object acts as proxy for the properties of Remote Object.





Consequences & Implications

→ Transfer Object DP reduces the network Traffic b/w client & Server.

→ Its minimises coupling (dependencies)

→ The main Advantage of this Design pattern is we can prevent network overhead problems. & we can improve performance of the system.

→ The main limitation of this Design pattern is upto Date Information may not be available at client side.

Keywords

) Reduce Network Traffic

) Improve Response Time

Transfer Data Access Times

Group Information.

Intercepting Filter Design Pattern

Context — client Request may have many diff Processing units & System must be able to meet them.

) The System has to receive the request by multiple protocols like http, ftp, SMTP ---

) The System must Authorize & Authenticate each request.

) The System needs to Add or Remove Info to request before further Processing.

Problem — Request & Response needs to be Pre & Post Processed in a simple organised manner.

) This Preprocessing needs to be performed with out effecting rest of the System i.e it should be pluggable.

Solution — use Filter to Pre-Processes the request & Post Processing of Response.

Configuring or Removing Filter takes places with out effecting rest of the system.

(63)

Consequences & Implications

1) Intercepting Filters is based on cohesion, loose coupling, increase declarative control.

2) Declarative Control allows Filters to be easily implemented either on temporary or permanent bases.

3) Declarative Control allows the sequence of invocations to be easily updatable.

Key Words

1) Central Object for preprocessing Request & Post-processing Response.

2) Authentication of each request.

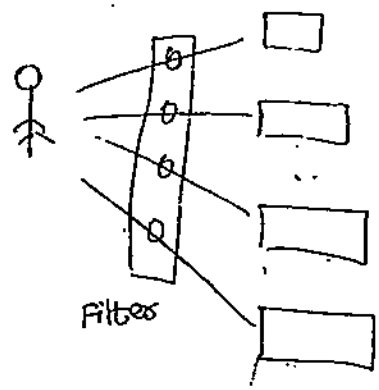
3) Authorization of each request.

4) logging Request Information.

5) Compression of Response.

6) Encryption of Response.

7) Altering Request & Response Information.



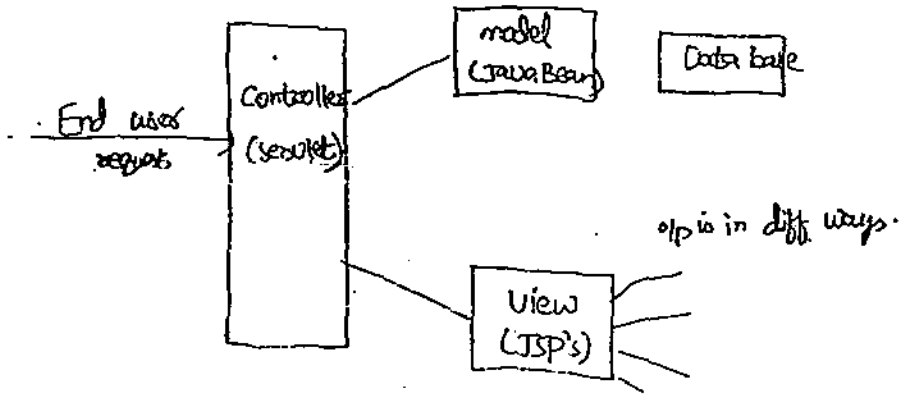
109 MVC Design Pattern (model View controller Design Pattern)

Context – In the systems involving users Interfaces, the following situation typically arises.

1) The system has to accept the Data from the users
2) Update Database and return the same Data to the user at later point of time. There are several ways in which the Data can be accepted and several ways to present Data to the users.
3) Data stored in the system in one form should be retrivable in another form.

problem – If the system Deploys a single component that interacts with users and maintains Database, then a requirement to support a new Type of Display or View will requires Re-Design of entire component.

solution – The solution is to seperate Data Presentation from Data maintainance and have a third component that co-ordinate first two. These 3 components are called model, View & controller.



model – It is responsible to keeping the Data or State of the application. It also manages the storage and retrival of Data from Data Source.

View – It contains presentation logic. It displays Data from the model to the end user. It also allows the user to interact with the system.

Controller – It manages whole show. It intantiates the model & view, and associates view with the model. Depending on application requirements it may instantiate multiple views for the same model.

Consequences or implications

It provides Job seperation. Hence with out effecting remaining components we can modify any component.

It improves code maintainability.

It supports loose coupling & high cohesion.

for the same Data we can provide multiple views.

It promotes Re-usability of model components.

The main limitation of this Design Pattern is it increases the overall complexity of the application & it is not suitable for small scale applications.

Key words

Seperation of Data Presentation from Data maintainance.

Provide services to different clients, web clients, WAP client

) multiple views such as html, xml

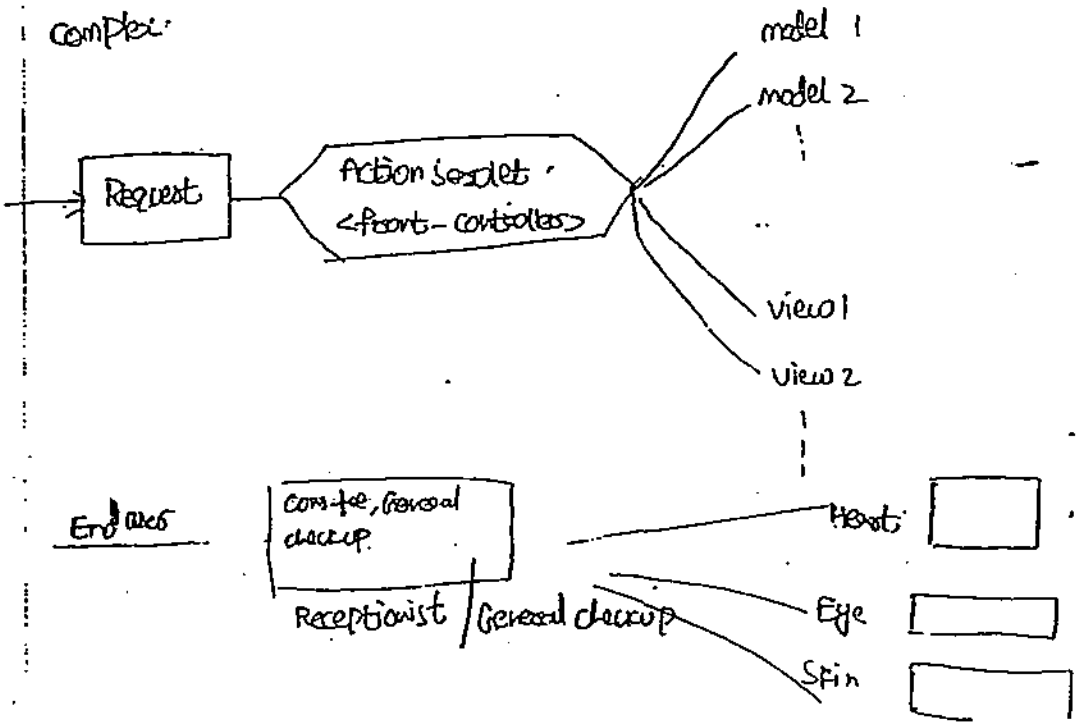
) Single controller

Front Controller Design Pattern

Context — The presentation tier request handling mechanism must control and coordinate processing of each user across multiple request. such control mechanism may be managed either in centralized or De-centralized manner.

Problem — when the users Accesing view component directly with out going through a centralized mechanism each view is required to provide its own system services which may result duplicate code.

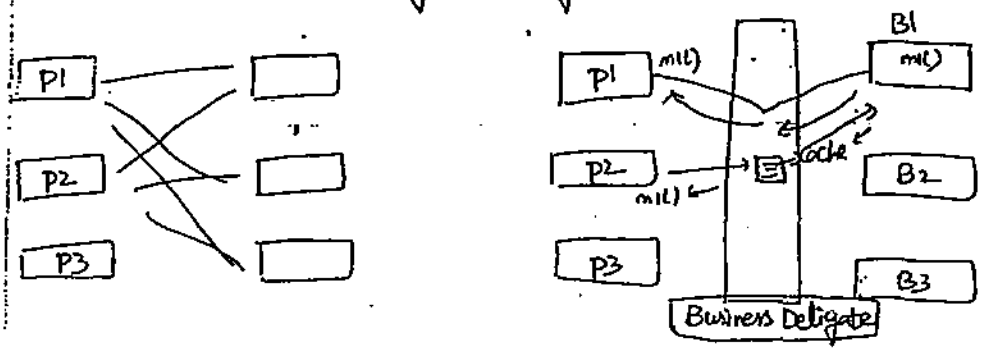
Solution — use front controller Design Pattern to gather common redundant request processing code in to a single component. This allows the application to be more cohesive and less complex.



Consequences, implications keywords.

1. Centralized control of request processing.
2. improves maintainability
3. Promotes Reusability.

Business Deligate Design Pattern



Context – A multi-tier Distributed System requires remote method Invocations to send and receive Data across Tiers. clients are exposed to the complexity of deling with distributed components.

Problem – Presentation Tier components interacts directly with Business Services. This Direct interaction exposes the underlying implementation details of Business service to the presentation Tier. As a result if there is any change in business service it requires to modify all presentation tier components.

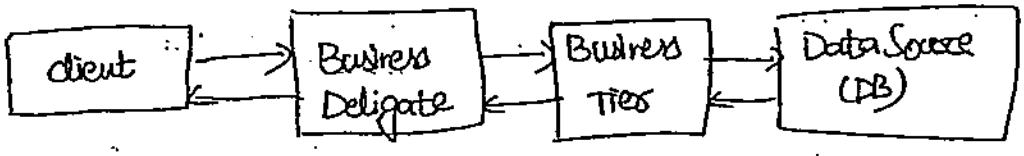
→ There may be network overheads. bcz Presentation Tier Components make to many invocations over the network and Presentation Tier Components won't maintain any caching.

→ Presentation Tier components are responsible for network issues also.

plation – use a business Deligate to reduce coupling btw presentation Tier and business Tier.

Business Deligate hides implementation Details of Business Service

) Business Deligate acts as client-side business proxy.



) If there is any change in business service its not require to perform any change in presentation – Tier components. we have to perform that change only in Business Deligate.

) Presentation Tier components are not responsible to handle any network issues. Bcz it is the responsibility of Business Deligate.

Business – Deligate can maintain caching so that we can reduce network traffic and improves (i.e increases) performances.

) It adds complexity bcz of extra Tier Business Deligate

) It decreases flexibility.

) The client is restricted to the options provided by Business Deligate.

Service Locator Design Pattern.

Context – Service look up and creation involves complex interfaces and network operations.

→ EJB clients needs to use JNDI API to look-up EJB Home object

→ Jms clients needs to use JNDI API to look-up Jms Components such as connection factories, Queues and Topics.

→ look-up and creation of service components could be complex and may use repeatedly in multiple client components.

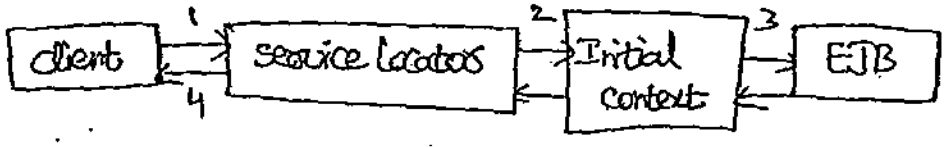
Problem – Every client component has to contain code for look-up and creation of service component. As the same code present in several places it is difficult to change every where if there is any change in service component.

→ Every call to look-up process is an individual network call. It may increases network traffic and effects performance. big at client side we are not maintaining any caching.

Solutions – use a service locator object to abstract all JNDI usage and to hide complexities of initial context creation, EJB Home object look-up and EJB object creation. multiple clients can reuse the same service locator object to reduce complexity.

→ Providing a single point of control for look-up processes is possible through service locator object.

It can reduce network overhead and improve performance by maintaining "cache".



) client invokes singleton instance of locator.

) locator creates and initial context for service look-up

Initial context obtain EJB service object.

Locator returns that service object to client.

Consequences or implications and keywords.

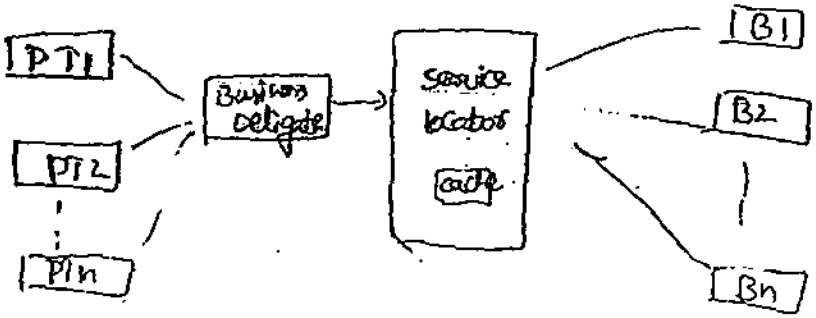
It Abstracts complexity

Provides uniform service Access for all clients.

facilitates adding new business components.

improves performance and reduces network over head :-

) It supports loosely coupling by reducing the dependency betw presentation Tier and Business Tier components.



Design Pattern	Application Tier
1. Data Transfer Object	Business Tier (Remote Hot-sources)
2. Intercepting Filter	Presentation Tier (Servlets, JSP)
3. Front Controller	"
4. MVC	Presentation & Business
5. Business Delegate	Business Tier
6. Service Locator	Business Tier

web.xml Configurations

memorising EJB Related Tags:

1. Reference to a local Bean

```
<ejb-local-ref>
    <ejb-ref-name> ejb/customer <|ejb-ref-name>
    <ejb-ref-type> Entity <|ejb-ref-type>
local <local-home> com.durga.CustomerHome <|local-home>
Remote <local> com.durga.Customer <|local>
<|ejb-local-ref>
```

JNDI look-up name

These must be fully qualified names of beans exposed interfaces.

<ejb-ref>

<ejb-ref-name>

<ejb-ref-type>

<home>

<remote>

<ejb-ref> but not <ejb-remote-ref>

<home> but not <remote-home>

</ejb-ref>

memorising JNDI <env-entry> Tag

look-up name we will use in code

<env-entry>

<env-entry-name> rates / discountRate </env-entry-name>

<env-entry-type> java.lang.Integer </env-entry-type>

<env-entry-value> 10 </env-entry-value>

</env-entry>

) Standard Actions ① <jsp:useBean>
<jsp:include> RD rd = req.getRD()
<jsp:forward> rd.forward();

) EL

) JSTL

② <%
ServletContext s = getServletContext().getInitParameter("pwd")
%>
$${initParam.pwd}

) custom Tags.
To write our own code in Java and make that functionality available to JSP.

Custom Tags

-> Standard Actions, EL & JSTL are not succeeded completely to seperate java code from the JSP.

Ex: Some times our requirement has to communicate with DB or with EJB. There is no standard Action define for this requirement. We can provide our own Custom Tags to meet our requirement from JSP 1.1 version onwards. i.e we can define our own Tags to provide required functionality which are nothing but custom Tags.

=> All custom Actions are divided in to 3 categories

Custom Tag
- Classic Tag (JSP 1.1)
- Simple Tags (JSP 2.0)
- Tag Files (JSP 2.0)

Components of CustomTag Application

=> custom Tag Application Contains the following 3 components.

1. Tag Handler – It is a simple java class which defines entire required functionality.

-> Every Tag Handler class should compulsory implement Tag interface either Directly or indirectly.

-> web containers is responsible for creation of Tag Handler Object. For this it always invokes Public no-Argument Constructor. Hence every Tag Handler class should compulsory

contain public no-Argument constructor.

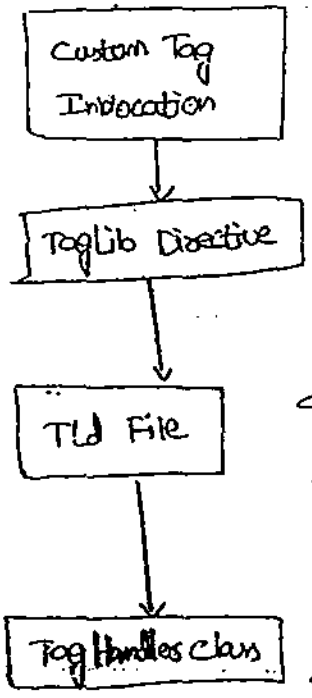
) TLD File. (Tag Library Descriptor File)

It is an xml File which provides mapping btw JSP (where customTag Functionality is required) and Tag Handler class (where customTag Functionality is available)

) Taglib Directive

It makes customTag Functionality available to the JSP. It defines the location of the TLD File.

Work-Flow of Custom Tag Application



<mine: myTag />

<%@ taglib prefix="mine" uri="/WEB-INF/tld/myTaglib.tld" %>

<taglib version="2.1" />
<tlib-version="1.2.3"> </t-v>
<tag>
<name> myTag </n>
<tag-class> myTagHandler </t-c>
</tag>
</taglib>

(1) whenever JSP Engine encounters a customTag, it identifies Prefix and checks for the corresponding Taglib Directive with matched prefix.

(2) From the Taglib Directive JSP Engine Identifies the location of TLD File.

(3) From TLD File JSP Engine identifies the corresponding Tag-Handler class.

(4) JSP Engine Executes Tag Handler class and provides required Functionality to the JSP.

Tag Extension API

→ we can build customTags by using only one Package javax.servlet.jsp.tagext.

→ This Package contains the following Interfaces and classes to build customTags.

(1) Tag :- It is the base Interface for all customTag Handlers. Every Tag Handler class should compulsory implement this interface either directly or indirectly.

→ This interface Defines 6 methods which can be applicable for any Tag Handler Object.

→ we should go for this interface if we are not manipulating Tag body and iteration is not required.

Iteration Tag

- It is the child Interface of Tag. we should go for this interface whenever iteration is required.
- It contains one extra method i.e doAfterBody().

Body Tag

- It is the child Interface of Iteration Tag. we should go for Body Tag whenever Tag Body manipulation is required.
- This interface defines the following 2 extra methods. setBodyContent() and doInitBody().

1) TagSupport Class

- It implements Iteration Tag Interface and provides default implementation for all its methods.
- It acts as a base class to provide (or) Develop Iteration & Simple Tags.
- It acts as Adapter for Tag and Iteration Tag Interfaces.

2) BodyTagSupport Class

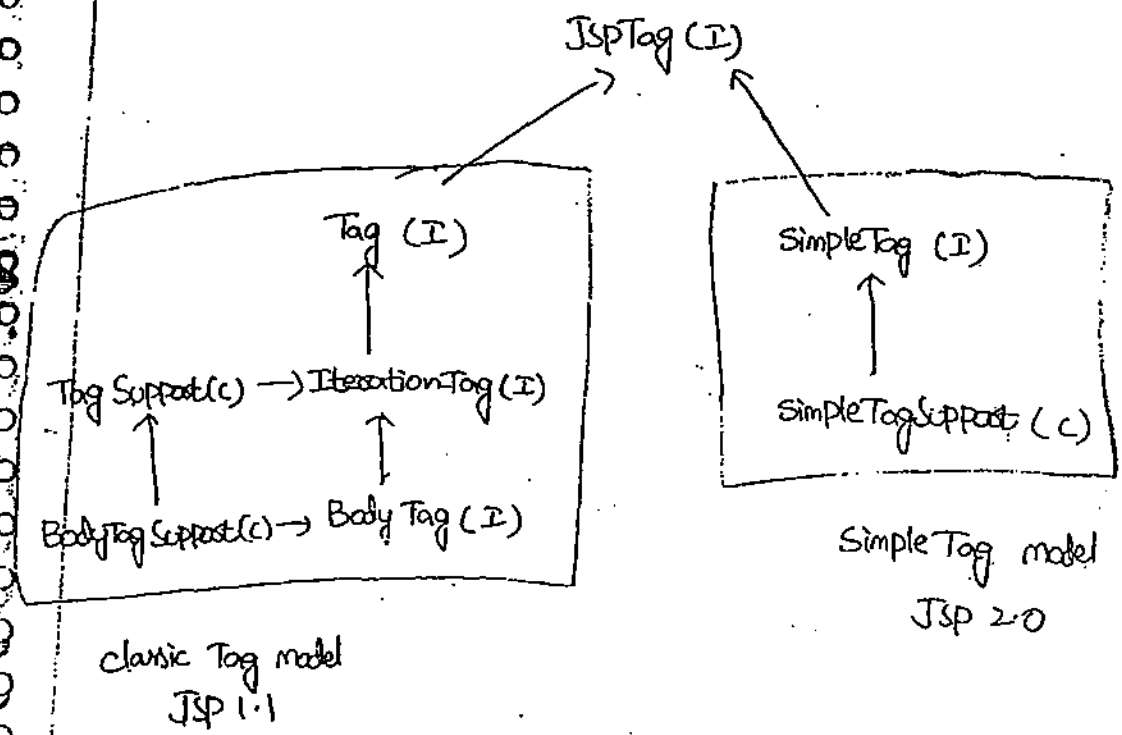
- This class implements BodyTag Interface and provides default implementation for all its methods.
- It acts as a Adapterclass for BodyTag
- we can use this as Base class for implementing CustomTag that processes Tag Body.

(6) BodyContent Class

→ Body Content object acts as a buffer for Tag body.

→ It extends JspWriter.

→ we have to use BodyContent class only in BodyTag Interface and BodyTagSupport class.



→ JspTag Interface is Just for Polymorphism purpose and doesn't contain any methods.

Implementing Tag (I)

It defines the following 6 methods

setPageContext()

setParent()

doStartTag()

doEndTag()

release()

getParent().

Tag Interface Defines the following Constants also

EVAL_BODY_INCLUDE

SKIP_BODY

EVAL_PAGE

SKIP_PAGE

LifeCycle of TagHandler that implements Tag Interface.

whenever JSP Engine Encounters a CustomTag it will identify the corresponding Tag Handler class through Tag lib directive and tld Files.

web Containes Creates an Instance of TagHandler by executing Public no argument Constructor (if it is not already available).

JSP Engine calls setPageContext() to make Page Context of JSP available to TagHandler class.

```
Public void setPageContext(PageContext p)
```

-> JSP Engine calls setParent() to make parentTag object available to TagHandler. This method is helpfull in nested Tags.

Public void setParent(Tag t)

-> setting Attributes – Attributes in custom Tags are exactly similar to properties of Java Beans. If a custom Tag has an Attribute then compulsory TagHandler class should contain one Instance variable and the corresponding setters methods.

-> JSP Engine calls setters methods for each Attribute.

-> JSP Engine calls doStartTag()

Public int doStartTag() throws JSPException.

-> we can define entire Tag functionality in this method only.

-> doStartTag() can return either EVAL-BODY-INCLUDE or SKIP-BODY.

-> If it returns EVAL-BODY-INCLUDE, the tag body will be included in result.

-> If it returns SKIP-BODY, JSP Engine doesn't consider Tag Body.

-> JSP Engine calls doEndTag()

Public int doEndTag() throws JSPException.

-> doEndTag() can return either EVAL-PAGE or SKIP-PAGE.

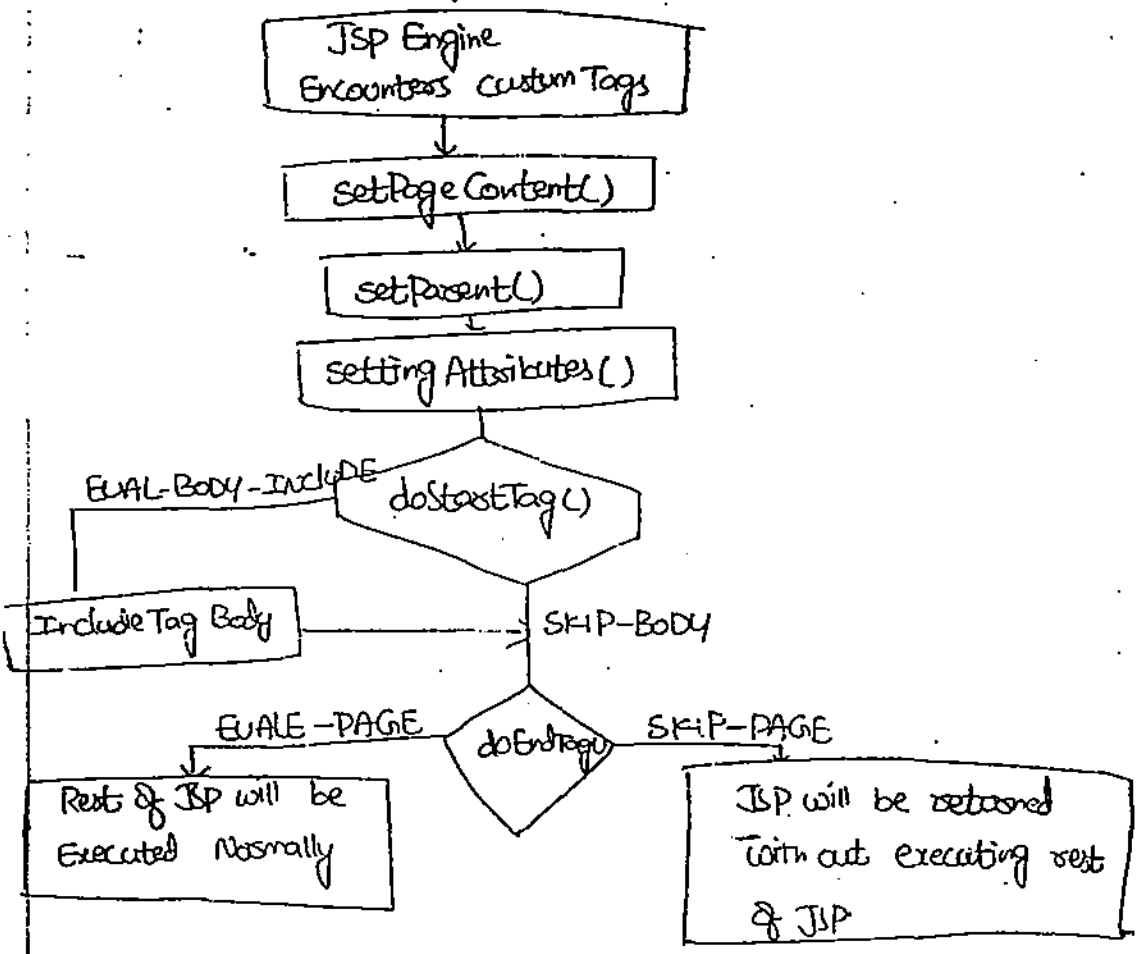
-> If it returns EVAL-PAGE, rest of the JSP will be executed normally.

-> If it returns SKIP-PAGE, JSP Page will be returned with out executing rest of the JSP.

Finally JSP Engine calls release() to perform clean-up Activies when ever Tag Handler Object no longer required.

Public void release()

Flow chart for Tag Handler Life Cycle.



Demo Program

test.jsp

```
<%@ taglib prefix="mine" uri="/WEB-INF/myTld.tld" %>

<h1> Hello ! This is Demo Jsp </h1>

<mine : mytag>
    This is body of custom Tag
</mine : mytag>

<h1> This is After Custom Tag </h1>
```

myTld.tld

```
<taglib version="2.1" >
  <tlib-version> 1.2.3 </tlib-version>
                <uri> www.durgasoft.com </uri>
  <tag>
      <name> myTag </name>
      <tag-class> tags.myTagHandler </tag-class>
  </tag>
</taglib>
```

myTagHandler.java

```
Package tags;
Import javax.Servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
```

```
Public class MyTagHandler implements Tag

    Private PageContext PContext;

  Public void SetPageContext (PageContext PContext)
  {
      this.PContext = PContext;
  }
  Public void SetParent (Tag t) {      }
  Public int doStartTag() throws JSPException
  {
      try
      {
        JspWriter out = PContext.getOut();
        out.pln(" Hello ! This is from Tag Handler");
      }
      catch (IoException e) {   }
      return EVAL-BODY-INCLUDE;
  }
  Public int doEndTag() throws JSPException
  {
      return EVAL-PAGE;
  }
  Public void getParent()
  {   return null;
  }
  Public void release() {    }
}
```

→ If doStartTag() returns SKIP-BODY & doEndTag() returns SKIP-PAGE then the o/p is

Hello ! This is Demo JSP

Hello ! This is from Tag Handler class

⇒ Instead of hard codeing the location of tld file in jsp we can specify through web.xml also

Example

```
<%@ taglib Prefix="mine" uri="www.durgasoft.com" %>
```

web.xml

```
<web-app>
    <jsp-config>
        <taglib>
            <taglib-uri> www.durgasoft.com </taglib-uri>
            <taglib-location> /WEB-INF/myTld.tld </taglib-location>
        </taglib>
    </jsp-config>
</web app>
```

we can map Tag lib directly to the tld File by using uri Attribute.

```
<%@ taglib Prefix="mine" uri="www.durgasoft.com" %>
```

.tld file

```
<taglib version="2.1" >
```

```
<tlib-version > 1.2.3 </t-v>
<uri> www.durgasoft.com </uri>
  <tag>
     <name>                    Directly Place in WEB-INF
     <tag-class>
  </tag>
</taglib>
```

This feature may not be supported by all web servers.

Structure of Tld File

```
<taglib version="2.1" >
<tlib-version> 1.2.3 </tlib-version>
<uri> www.durgasoft.com </uri>
<tag>
    <description>
    <name> mytag </name>
    <tag-class> tags.MyTagHandler </tag-class>
    <body-content> XXXX </body-content>
```

Fully Qualified Name

↳ empty / tag dependent / scriptless / JSP

```
    <attribute>
        <name>
        <required> true/false
        <rtexprvalue>
    </attribute>
</tag>
```

body-content — It describes the type of content allowed inside tag body. The allowed values are

① empty - The body of the Tag should be empty. we can't take any tag body. In this case we can invoke custom Tag as follows.

- <mine : myTag /> or <mine : myTag>
  <|mine : myTag>

② tagdependent - Entire Tag body will be Treated as Plain text. JSP Engine sends Tag Body to the Handler class with out any Processing.

③ scriptless — Tag Body should not Contain any Scripting Elements (i.e scriptlet, Expression) But Standard Actions & EL Expressions are allowed.

④ JSP - No restrictions on Tag Body. whatever allowed in JSP is by default allowed in Tag Body also.

=> The Default Value is JSP

Attributes

-> A Tag can contain any no. of Attributes we can declare these Attributes by using <Attribute> tag in tld.

-> <attribute> Contains the following child Tags.

① <name> — name of the Attribute.

② <required> — true - if Attribute is mandatory
— false - " " " optional

Default value is — false

3) <rtexprvalue> runTime Expression Value

true - RunTimeExpressions also allowed.

Ex: <mine : mytag color="${param.color}" />

Default Value is false

false - RunTimeExpressions are not allowed and we have to Provide only literals.

Ex: <mine : mytag color="red" />

Demo program for empty CustomTag with mandatory Attribute

→ A tag can contain Attributes also. For each Attribute we have to do the following things.

1. we have to declare that Attribute in tld by using <attribute> Tag.
2. For each Attribute in TagHandler class we have to define one Instance var and corresponding setter method.
3. In the case of optional Attributes there may be a chance of null pointer Exceptions. we have to handle carefully.

test.JSP

```
<%@ taglib prefix="mine"  uri="/WEB-INF/myTLD-file.tld" %>
<mine:myTag1  numbers="2" />
<mine:myTag1  numbers="5" />
```


myTld.tld

```
<taglib version="2.1" >
  <tlib-version> 1.2.3 </tlib-version>
    <tag>
        <name> myTag </n>
        <tag-class> tags.MyCustomTags </tag-class>
        <attribute>
            <name> numbers </name>
            <required> true </required>
        </attribute>
    </tag>
</taglib>
```

MyCustomTag.java

```
package tags.
 import javax.servlet.jsp.tagext.*;
 import javax.servlet.jsp.*;
```

```
Public class MyCustomTag implements Tag
{
        Private int numbers;
        PageContext pcontext;
    Public void setPageContext(PageContext pcontext)
    {
        this.pcontext=pcontext;
    }
    Public void setParent(Tag t) {    }
    Public void setNumbers(int numbers)
    {
        this.numbers = numbers;
    }
Public int doStartTag() throws JSPException
{
    try
    {
        JSPWriter out = pcontext.getOut();
        out.println("<h1> Double of" + numbers + "is:" +
                                    (2*numbers) "</h1>");
    }
    catch (IoException e) {    }
    return EVAL_BODY_INCLUDE;
}
Public int doEndTag() throws JSPException
{
    return EVAL_PAGE;
```

```
    Public void getParent()
    {
        return null;
    }
    Public void release()
    {
    }
}
```

Demo program for Empty Tag with optional Attribute

test.jsp

```
<%@ taglib

 <mime:myTag  greetings name="durga" />
 <mime:greetings />
```

.tld

```
 <attribute>
        <name> greetings </name>
        <required> false </required>
 </attribute>
```

MyCustomTag.java

```
Public class MyCustomTag implements Tag
{
    Private String name;

    Public void setName(String name)
    {
        this.name = name;
    }
```

```
Public int doStartTag() throws JspException
{
    try
    {
      JspWriter out = PContext.getOut();
      if(name==null)
      {
        o.pln("Hello Guest Good Morning");
      }
      else
      {
        o.pln(" Good Morning " + name);
      }
    }catch(IoException e) { }
    return EVAL_BODY_INCLUDE;
  }
}
```

Iteration Tag

→ Iteration Tag is the child Interface of Tag. If we want to Include Tag Body multiple times then we should go for Iteration Tag.

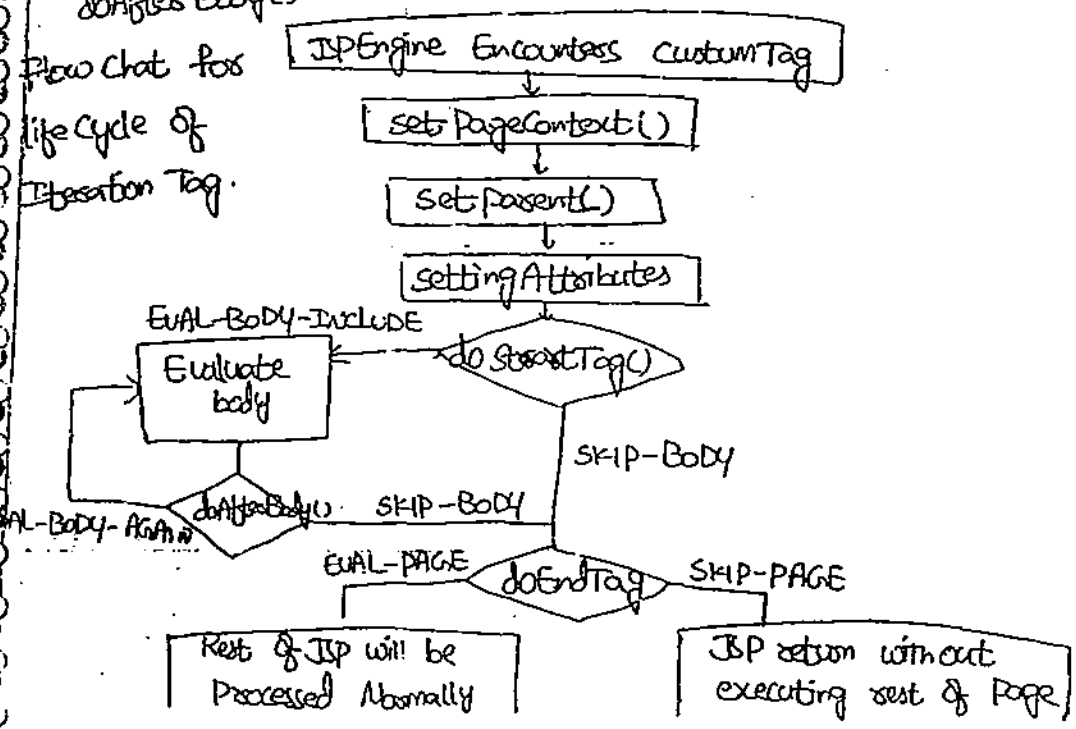
→ It contains only one extra method doAfterBody() and one extra Constant EVAL-BODY-AGAIN.

doAfterBody()

Public int doAfterBody() throws JSPException

1. This method is invoked after doStartTag() and can return either SKIP-BODY or EVAL-BODY-AGAIN
2. If it returns SKIP-BODY the body will be skipped and jsp Engine calls doEndTag().
3. If it returns EVAL-BODY-AGAIN then the body will be considered once again followed by execution of doAfterBody().

Flow Chat for life Cycle of Iteration Tag.



emo Program for Iteration Tag

```
<%@ taglib prefix="mine" uri="/
<mine:loop count="2">
<h1> Learning CustomTags is very Easy -- </h1>
</mine:loop>
```

count=0
count=-2 } no o/p.

tld

```
<taglib version="2.1" >
<tlib-version> 1.2.3 </t-v>
  <tag>
      <name> loop </n>
      <tag-class> tags.myCustomTag </t-c>
      <attribute>
            <name > count </name >
            <required> true </required>
      </attribute>
  </tag>
</taglib>
```

myCustomTag.java

```
Package tags;
import
import
```

```
Public class MyCustomTags implements IterationTag
{
    Private int count;
    PageContext pcontext;
    Public void setCount(int count)
    {
        this.count = count;
    }
    Public void setPageContext(PageContext pcontext)
    {
        this.pageContext = pcontext;
    }
    Public void setParent(Tag t)
    {
    }
    Public int doStartTag() throws JSPException
    {
        if (count > 0)
            return EVAL_BODY_INCLUDE;
        else
            return SKIP_BODY;
    }
    Public int doAfterTag() throws JSPException
    {
        if(--count > 0)
            return EVAL_BODY_AGAIN;
        else
            return SKIP_PAGE BODY;
    }
```

```
Public void doEndTag() Throws JSPException
{
        return EVAL-PAGE;
}
Public void getParent()
{
        return null;
}
Public void release() { }
}
```

Tag Support Class

The main draw back of implementing Tag and Iteration Tag Interfaces is we have to provide implementation for all methods eventhough most of times we have to consider doStartTag(), doEndTag() and doAfterBody().

- we can resolve this problem by using Tag Support class.
- TagSupport class implements Iteration Tag and provides default implementation for all its methods.
- It is easy to extend TagSupport class and override required methods instead of implementing all methods.

Implementation of TagSupport class


```
Public class TagSupport implements IterationTag, Serializable
{
    Public PageContext pagecontext;
Private Tag t;
    Public void setPageContext(PageContext p)
    {
        PageContext = p;
    }
    Public void setParent(Tag t)
    {
        this.t = t;
    }
    Public int doStartTag() Throws JSPException
    {
        return SKIP-BODY;
    }
    Public int doAfterBody() throws JSPException
    {
        return SKIP-BODY;
    }
    Public int doEndTag() throws JSPException.
    {
        return EVAL-PAGE;
    }
    Public Tag getParent()
    {
    }   return t;
```

```
public void release()
{
        t=null;
        pageContext=null;
}
```

Note

1) The default return type of doStartTag() and doAfterBody() is SKIP-BODY.

2) The default return type of doEndTag() is EVAL-PAGE.

3) pageContext variable is by default available to the child class Hence we can use this vbl directly in our Tag Handler class.

Example

```
public class MyCustomTag extends TagSupport
{
    public int doStartTag() throws JSPException
    {  try
       {JSPWriter out =pageContext.getOut();
        out.pen("<h1>This is from CustomTag Handler class" </h1>);
       }
       catch (IoException e) {    }
       return EVAL_BODY_INCLUDE;
    }
```

pageContext is by default available through Inheritance of TagSupport

fast JSP

<%@ taglib prefix="mine" uri=" one Time will be executed

<mine:myTag />

BodyTag Interface

→ It is the child Interface of Iteration Tag. If we need to manipulate Tag Body then we should go for BodyTag Interface.

→ BodyTag Interface defines the following 2 extra methods.

① public void setBodyContent(BodyContent b)

② public void doInitBody() throws JSPException

→ BodyTag Interface defines the following 2 extra constants.

① EVAL_BODY_BUFFERED

② EVAL_BODY_TAG ⇒ Deprecated in JSP 1.2V (JSP 1.1)

Body Content

→ It is an Abstract class and extends JSPWriter

→ BodyContent Object represents the body of the Tag.

→ BodyContent class defines the following methods to retrieve Tag Body

① public String getString() — returns body content as String.

② public Reader getReader() — returns a reader Object to extract Tag Body.

JSPWriter getEnclosingWriter()

→ Returns JSP writer Object of Parent class.

→ If there is no parent then it returns current JSP out object.

Public void clearBody()

→ It clears body Content Object i.e the data present in body content object will be removed.

LifeCycle of BodyTag

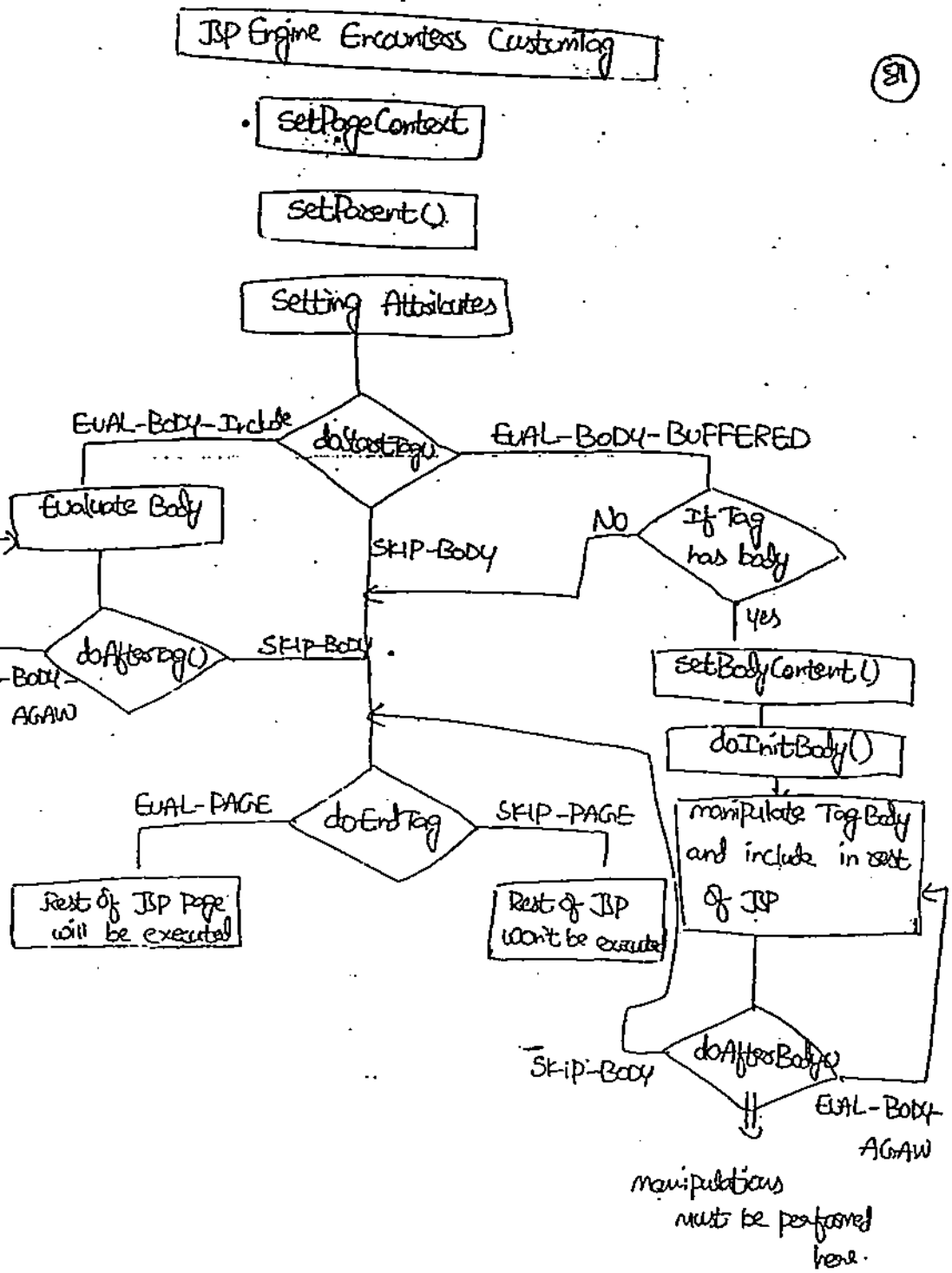
→ The Life Cycle JSP of BodyTag Handler is exactly similar to Iteration Tag Handler. The difference is doStartTag() can return EVAL-BODY-BUFFERED in addition to EVAL-BODY-INCLUDE and SKIP-BODY.

→ If the method returns EVAL-BODY-INCLUDE then body will be evaluated exactly similar to Iteration Tag.

→ If method returns EVAL-BODY-BUFFERED and Tag contains body then JSP Engine creates BodyContent Object and call setBodyContent() followed by doInitBody().

⇒ The setBodyContent() and doInitBody() won't be executed in following cases.

(1) If doStartTag() returns either EVAL-BODY-INCLUDE or SKIP-BODY or if the doesn't have the Body.


JSP Engine Encounters CustomTag
setPageContext
setParent()
Setting Attributes
doStartTag()
EVAL-BODY-Include
EVAL-BODY-BUFFERED
Evaluate Body
SKIP-BODY
No
If Tag has body
Yes
doAfterBody()
SKIP-BODY
EVAL-BODY-AGAIN
setBodyContent()
doInitBody()
manipulate Tag Body and include in rest of JSP
doEndTag
EVAL-PAGE
SKIP-PAGE
Rest of JSP page will be executed
Rest of JSP won't be executed
doAfterBody()
SKIP-BODY
EVAL-BODY-AGAIN
manipulations must be performed here.

BodyTag Support class

- This class extends Tag Support class and implements BodyTag Interface.
- This class provides default implementation for all 9 methods available in Body Tag Interface.
- It is ~~very~~ Easy to implement Tag Handler by Extending BodyTag Support class instead of implementing BodyTag interface directly. In this case we have to overside only required methods instead of implementing all 9 methods.

```
Public class BodyTagSupport extends TagSupport implements
                                        BodyTag
{
    Public PageContext pageContext;
    Public BodyContent bodyContent;
    public Tag t;
    Public void setPageContext (PageContext p)
    {
        PageContext = p;
    }
    Public void setParent (Tag t)
    {
        This.t ;
    }
    Public ~~void~~ int doStartTag () throws JspException
    {
        return EVAL-BODY-BUFFERED;
    }
```

```
Public int doAfterBody() throws JSPException
{    return SKIP-BODY;
}
Public int doEndTag() throws JSPException
{    return EVAL-PAGE;
}
Public void setBodyContent (BodyContent bodycontent)
{    this.bodyContent = bodyContent;
}
Public void doInitBody() throws JSPException {  }
Public Tag getParent()
{    return t;
}
Public void release(){  }
}
```

PageContext P, BodyContent C are by default available to the Tag Handler class.

The default return type of doStartTag() is EVAL-BODY-BUFFERED doAfterBody() is SKIP-BODY, doEndTag() is EVAL-PAGE.

```
<%@ taglib prefix="mine" uri=" /
<mine:body>
 <h1> Learning Custom Tags is very Easy -- <h1>
<|mine:body>

<mine:body>
    <h1> Durga Software Solutions <|h1>
<|mine:body>
```

tld

```
<tag>
    <name>
    <tagclass>
    <body-content> JSP <|b-c>
<|tag>
```

myCustomTag.java

```
Public class MyCustomTag extends BodyTagSupport
{
    public int doAfterTagBody() throws JSPException.
    {
        try
        {
            String s= bodyContent. getEnclosingString();
            s= s.toUpperCase();
            JSPWriter out = bodyContent.getEnclosingWriter();
        }   out. Println(s);
```

```
    catch (IoException e) {    }
    return SKIP-BODY;
}
}
```

7/6/09

Comparision btw TagSupport and BodyTagSupport classes.

method	TagSupport	BodyTagSupport.
(1) doStartTag()		
(a) Possible Return values	EVAL-BODY-INCLUDE SKIP-BODY	EVAL-BODY-INCLUDE SKIP-BODY EVAL-BODY-BUFFERED.
(b) Default value from implementation class	SKIP-BODY	EVAL-BODY-BUFFERED.
(c) no. of times it can be called per Tag	only once	only once
(2) doAfterBody()		
(a) Possible Return values.	EVAL-BODY-AGAIN SKIP-BODY	EVAL-BODY-AGAIN SKIP-BODY
(b) default value from implementation class	SKIP-BODY	SKIP-BODY
(c) no. of times it can be called per Tag	0 or more	0 or more

3) doEndTag()

(a) Possible return values	EVAL-PAGE SKIP-PAGE	EVAL-PAGE SKIP-PAGE
(b) Default value from implementation class	EVAL-PAGE	EVAL-PAGE
(c) no of times it can be called per Tag	once	once

4) doInitBody() & setBodyContent.

→ circumstances under which these can be called and no of times per Tag Invocation	Not Applicable	Executed only once if and only if doStartTag() returns EVAL-BODY-Buffered and Tag contains Body.

Cooperative or Nested Tags

→ Some times a group of Tags work Together to perform required functionality. Such type of Tags are called Co-operative or nested Tags.

Ex: In JSTL <c:choose>, <c:when> and <c:otherwise> Tags work together to implement Java's switch stmt.

These Tags are called Co-operative Tags.

- <c:choose> → Parent Tag

<c:when> → child Tag

!

<c:otherwise>

</c:choose>

Ex: <mime : outerTag >

<mime : InnerTag /></mime : outerTag >

→ From InnerTag (child Tag) we can get Parent Tag reference by using getParent().

Tag Parent = getParent();

Demo Program for nested Tags

first.JSP

```
<%@ taglib prefix="mime"

  <mime: myTag> — 0
    <mime: myTag> — 1
      <mime: MyTag> — 2
        <mime: MyTag /> — 3
      </mime : myTag>
    </mime : myTag>
  </mime : myTag>
```

tld. File

```
<taglib version="2.1" >
    <tlib-version>1.2 </t-v>
    <tag >
        <name> MyTag
        <tag-class> tags. MyCustomTag </t-c>
        <body-content> JSP </b-c>
    </tag>
</taglib>
```

MyCustomTag.Java

```
Public class MyCustomTag extends TagSupport
{
    Public int doStartTag() throws JSPException.
    {
        int nestedLevel =0;                    Nested level is : 0
        Tag t = getParent();                   "   " : 1
        while (t != null)                      "   " : 2
        {                                      "   " : 3
            nestedLevel ++;
            t=t.getParent();
        }
        try
        {
            pageContext.getOut().pln("<h1> Nested level is:"
                                    +nestedLevel + <\h1>");
        }
        catch (IoException e) { }
        return EVAL-BODY-INCLUDE;
    }
}
```

Example-2

test.jsp

```
<%@ taglib prefix = 'mine' uri = " /          %>
<mine : menu>
        <mine : menuItem   item="chicken"  />
        <mine : menuItem   item = "mutton"  />
<\mine:menu> <mine : menuItem   item = "fish" />
```

```
<mine:menu>
        <mine:menuItem   item="SCJP" />
                         item="SCWCD" />
                         item="SCBCD" />
                         item="SCJD" />
</mine:menu>
```

tld File

```
<taglib version="2.1" >
    <tlibversion > 1.2.3 </t-v>
     <tag>
         <name> menu </n>
         <tag-class> tags.menuTag </t-c>
         <body-content> JSP </b-c>
    </tag>
    <tag>
         <name > menuItem </n>
         <tag-class> tags.menuItemTag </t-c>
         <body-content > JSP </b-c>
    ~~</tag>~~ <attribute >
                <name> item </name>
                <required > true </required>
            </attribute >
    </tag>
</taglib>
```

```
public class MenuTag extends TagSupport
{
    Private ArrayList l=null;
    Public int doStartTag() throws JspException
    {
        l=new ArrayList();
        return EVAL_BODY_INCLUDE;
    }
    Public void addMenuItem( String s)
    {
        l.add(s);
    }
    Public int doEndTag() throws JSPException
    {
        try
        {
            JSPWriter out=PageContext.getOut();
            o.pln("<h1> Items are :" +l );
        }
        catch (IoE e) { }
        return EVAL_PAGE;
    }
}
import java.util.*;
Public class MenuItemTag extends TagSupport
{
    Private String item;
    Public void setItem (String item)
    {
        this.item= item;
    }
```

menu items are: chicken, mutton, fish

menu items are : SCJP, SCWCD, SCBCD, SCJD.

```
public int doStartTag()
{
    MenuTag parent = (MenuTag).getParent();
    parent.addMenuItem(item);
    return EVAL_BODY_INCLUDE;
}
```

Getting an Arbitrary Ancestor class

we can get immediate Parent by using getParent().

TagSupport class contains the following method to get an arbitrary Ancestor class.

```
Public Static Tag findAncestorWithClass(Tag t, Class c)
```

(t3, class t1);

Accessing JSP Implicit Objects & Attributes in TagHandler class

→ using PageContext API write Tag Handler code to access JSP Implicit Objects and Access web Application Attributes.

→ TagHandler class can get PageContext Object by setPageContext() By using this PageContext Object Tag Handler can get all other JSP Implicit objects.

	JSP Implicit Object	PageContext class method
1	request	getRequest()
2	response	getResponse()
3	config	getServletConfig()
4	application	getServletContext()

session getSession()

out getOut()

Page getPage()

Exception getException()

If the Enclosing JSP is not ~~Error~~ Page Then getException() returns null

```
package tags
Public class myCustomTag extends TagSupport
{
    public int doStartTag() throws JSPException
    {
      ServletRequest req = PageContext.getRequest();
    String s1= req.getServerName() + ":" + req.getServerPort();

      ServletResponse resp = PageContext.getResponse();
      String s2 = resp.getContentType();
      HttpSession session = PageContext.getSession();
       String s3 = Session.getId();
       Throwable e= PageContext.getException();
      try
      {
        JspWriter out = PageContext.getOut();
        o.pln("<h1>"+s1);
        o.pln(s2);
        o.pln(s3);
```

```
        o.pln(e + "</h1>");
    }
    catch(IOE e) {  }
    return E-B-I
}
```

(87)

Getting and setting Attributes by using PageContext API.

PageContext defines the following methods to perform Attribute management.

```
Public void setAttribute(String name, Object obj);
Public void setAttribute(String name, Object obj, int Scope);
Public Object getAttribute(String name)
Public Object getAttribute(String name, int Scope)
Public void remove(String name)
Public void remove(String name, int Scope)
Public Object findAttribute(String name)
Public Enumeration getAttributeNames(int Scope).
```

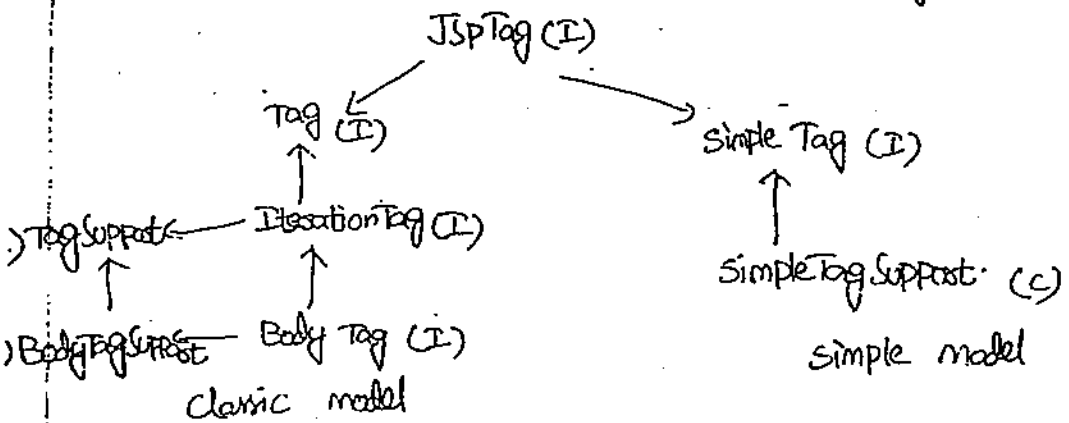
1-4
P, R, S, A

Simple Tag Model

→ implementing Custom Tags by using classic Tag model (Tag, Iteration Tag, Body Tag, TagSupport, BodyTag Support) is very complex bcz each tag has its own life cycle and different possible return types for every method.

→ To Resolve this complexity Sun People introduced simple Tag model in JSP 2.0 V.

→ In this model we can build custom Tags by using SimpleTag Interface and its implementation class SimpleTagSupport.



SimpleTag Interface

→ This is child Interface of JSPTag and contains the following 5 methods.

1. Public void setJspContext(JspContext c)
2. public void setParent(JSPTag t)
3. public void setJspBody(JspFragment f)
4. public void doTag() throws JSPException, IOException.
5. Public JSPTag getParent();

Lifecycle of Simple Tag Handler

1. when JSP Engine Encounters a custom Tag in JSP, it will identify the corresponding Tag Handler class by using taglib directive and tld File. It creates a new Instance of Tag Handler class by executing public no Argument Constructor. Simple Tag objects are never reused by web Containers. So for each Tag Invocation a new Tag Handler Object will be created.

2. web Containers Executes setJspContext() to make JSPContext Object available to Tag Handler class. By using this JSP Context Object Tag Handler class can get all JSP Implicit Objects and Attributes.

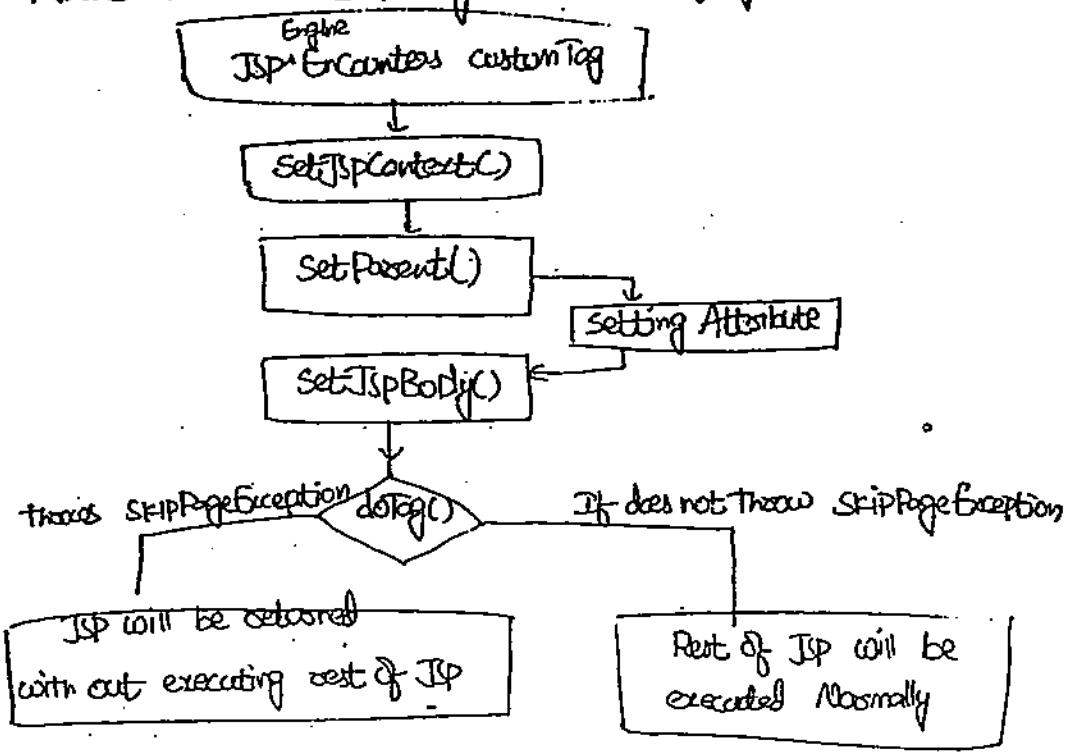
3. web Containers executes setParent() to make Parent Tag Object available to Tag Handler class. This method is usefull in the case of nested Tags.

4. If the Tag is invoked with a body then setJSPBody() will be executed by taking JSPFragment Object as argument. JSPFragment represents Tag Body in Simple Tag model TagBody should not contain any Scripting Elements.

5. finally doTag() will be executed which is responsible to provide required functionality. This method is equivalent to doStartTag(), doEndTag() & doAfterBody().

6. once doTag() completes, the Tag Handler Object will be destroyed by web Container.

Flowchart for SimpleTag Handler Lifecycle



SimpleTagSupport Class

→ This class provides default implementation for all methods of SimpleTag Interface.

→ This class also provides the following additional methods.

1. JSPContext getJspContext()
2. JSPFragment getJspBody()
3. JSPTag findAncestorWithClass(JSPTag t, class c).

first.jsp

```
<%@ taglib prefix="mine" uri="
<mine:myTag> This is tag Body </mine:myTag>
<h1> This is rest of the JSP </h1>
```

tld-file

```
<tag>
    <name> myTag </n>
    <tag-class> tags.myCustomTag </t-c>
    <body-content> tagdependent </b-c>
                   consider as Plain Text
```

```
Public class MyCustomTag extends SimpleTagSupport
{
    Public void doTag() throws JSPException, IoException
    {
        JSPWriter out = getJSPContext().getOut();
        o.pln ("<h1> Hello this is Simple Tag </h1>");
        throws new SkipPageException();
    }
}
```

If doTag() throws SKIP-Page Exception Hello this is Simple Tag

If doTag() doesn't throw SKIP-Page Exception then o/p is

Hello this is Simple Tag

This is rest of JSP.

What is the Diff btw classic & Simple Tag w.r.t Tag Body.

In classic Tag model we are allow to take Scripting Elements in Tag body hence the Possible values for Body Content Tag are

empty, tagdependent, scriptless, JSP.

→ But in simple Tag model scripting Elements are not allowed in in tag body. Hence the allowed values for <Body-Content> tag are empty, tagdependent, scriptless but not JSP.

Processing Body Content in SimpleTags

→ we can Access Tag Body inside a simple Tag Handler by using getJSPBody(). This method returns JspFragment Object.

```
public JspFragment getJspBody()
```

→ JSP Fragment object represents Tag Body on this object we can call the following 2 methods.

1. JSPContext getJSPContext();
2. void invoke(Writer w); — It causes evaluation of TagBody and return to supplied writer.

→ use "null" Argument to write directly to the JSP OutPutStream

test.JSP

```
<%@ taglib
<h1> This is Before Tag Invocation </h1>
<mime:MyTag> This is Tag Body </m-M>

<h1> This is after Tag Invocation </h1>
```

tld file

```
<tag>
    <name>
    <tag-class>
    <body-content> tagdependent </b-c>
```

```
public class MyCustomTag extends SimpleTagSupport
{ public void doTag() throws JspException, IoException
  {
    JspWriter out = getJspContext().getOut();
    o.pln("<h1> Hello This is from SimpleTag </h1>");
    getJspBody().invoke(null);
    throws new SkipPageException();
  }
}
```

This is Before Tag Invocation
Hello This is from simple tag
This is Tag Body

Key Diff btw Simple and classic Tags

Property	Simple Tags	Classic Tags.
Tag Interfaces	Simple Tag	Tag ↑ IterationTag ↑ BodyTag.
Supporting Implementation classes	SimpleTagSupport	TagSupport ↑ BodyTagSupport.
Key - life cycle methods that we have to implement	doTag()	doStartTag() doEndTag() doAfterBody()
How to write response to JspOutputStream	getJspContext().getOut. Println( ); ⇒ no need to use try, catch for IoException	PageContext.getOut(), Println(); ⇒ compulsory we should enclose by using try, catch for IOExcep

5) How to Access JSP Implicit Objects & Attributes.	By using JSPContext Object	By using PageContext Object.
6). How to cause the body to be Processed	getJspBody().invoke(null)	Return EVAL_BODY_INCLUDE from doStartTag() (or) EVAL_BODY_BUFFERED in doAfterBody() of body tag Interface from doStartTag().
7) How to cause the current Page Evaluation to stop.	throw SKIPPageException from doTag()	Return SKIP_PAGE from doEndTag().

Dynamic Attributes

In general, Tags can Contain Attributes for which we can have declarations in the tld file by <attribute> tag.

For these Attributes we have to maintain Instance's variables and corresponding setter methods in Tag Handler class. Such Type of Attributes are called Static Attributes

But we can use Attributes, eventhough Tld File doesn't contain any Attribute Tag Declarations. Such type of Attributes are called Dynamic Attributes.

Dynamic Attributes are applicable for both classic & Simple Tags and Introduced in JSP 2.0 Versions.

To support Dynamic Attributes we have to do following Arrangements.

In the Tld File we have to use Dynamic Attributes Tag

```
<taglib ----  >
    <tag>
        <dynamic-attribute>true</dynamic-attribute>
    </tag>
</taglib>
```

② The corresponding Tag Handler class should Implement Dynamic Attributes Interface.

⇒ This interface Introduced in Jsp 2.0 Version and Contains only one method setDynamicAttribute().

```
public void setDynamicAttribute (String namespace,
                                 String name, Object value)
```

⇒ web container calls this method for each Dynamic Attribute.

test.jsp

```
<%@ taglib prefix="mime" uri="------" %>
<mime:myTag num="2" min="4" max="5" pow="3" />
<mime:myTag num="4" min="15" />
```

tld File

```
<taglib version="2.1" >
    <tlib-version > 1.2.3 </t-v>
    <tag >
        <name > myTag </n >
        <tag-class > tags.MyCustomTag </t-c>
        <body-content> empty </b-c>
```

```
    <attribute>
        <name> num <\n>
        <required> true <\r>
    <\attribute>
    <dynamic-attribute> true <\dynamic-attribute>
  <\tag>
<\ taglib>
```

myCustomTag.java

```
Public class MyCustomTag extends SimpleTagSupport
                implements DynamicAttribute
{
    int num;
    String result=" ";
    Public void SetNum(int num)
    {
        this.num =num;
    }
  Public void SetDynamicAttribute (String ns, String name,
                                   Object Value)
  {
    int i= Integer.ParseInteger((String) Value);
```

```
if( name ==="min")
{
    result=result + "The minimum Value is :" +
                        math . min(num, i);
}
else if (name ==="max")
{
    result= result+"The maximum Value is ; " +
                        math . max( num, i);
}
else if( name. name ==="pow")
{
    result = result + "The Powers of numbers is : " +
                        math. max (num, d);
}
}
Public void doTag() throws JspException , IOException
{
    getJspContext(). getOut(). Println ("<h1>" + The result +
                        "</h1>");
}
}
```

o/p :- The minimum Value is : 2

The maximum Value is : 5

The Powers of numbers is : 8

Tag Files

Describe the symantics of Tag File model.

Describe Application Structure for Tag Files.

write a Tag File and Explain the constraints on JspContent in body of Tag.

Tag Files Concept has Introduced in JSP 2.0 Version.

Tag File is a simple JSP Page (or) JSP Document designed to be used as custum Action.

The main Advantage of Tag Files is, we can build very easily when composed with classic & simple Tags.

The main Limitation of Tag Files is, it won't suggestable for doing much processing.

Building and using a Simple Tag File

write a JSP Page or Document and save it with ".tag" extension.

place this .tag file in /WEB-INF/tags.

put a taglib Directive in the JSP with "tagdir" Attribute.

```
<%@ taglib prefix="mime"  tagdir="/WEB-INF/tags" %>
```

Demo Program

test.jsp

```
<%@ taglib prefix="mine" tagdir="/WEB-INF/tags" %>
<h1> This is Tag File Demo Example </h1>
<mine : myTag />
<mine : myTag />
```

This 2 should be same name

myTag.tag

```
<h1> Hello !! This is from TagFile </h1>
```

Tag File1
- test.jsp
- WEB-INF
  - tags
    - myTag.tag

Note

⇒ Tag Files internally converted in to Simple Tag Handlers (available in work folder)

⇒ Not required to write tld File.

Declaring a TagFile with Attribute

⇒ we can define Attribute for Tag Files by using attribute Directive.

```
<%@ attribute name="num" required="true"
              rtexprvalue="true" %>
```

test.jsp

```
<%@ taglib prefix="mine" tagdir="/WEB-INF/tags" %>
<h1> This is TagFile Demo Example </h1>
<mine:myTag title="Kick" />
```

myTag.tag

```
<%@ attribute name="title" required="true"
              rtexprvalue="true" %>
<h1> Hello --- ${title} is a big Hit movie </h1>
```

o/p:- This is TagFile DemoExample
Hello --- Kick is a big Hit movie.

Declaring Body Content for a Tag File

- Tag File Invocation can contain body but we are not allow to use scripting Elements.
- we can declare body-content in tag File by using tag Directive.

```
<%@ tag body-content="tagdependent" %>
```

allowed values are — empty, tagdependent, scriptless

Default value — scriptless.

→ In the Tag File we can Access Tag body by using

```
<jsp:doBody />
```

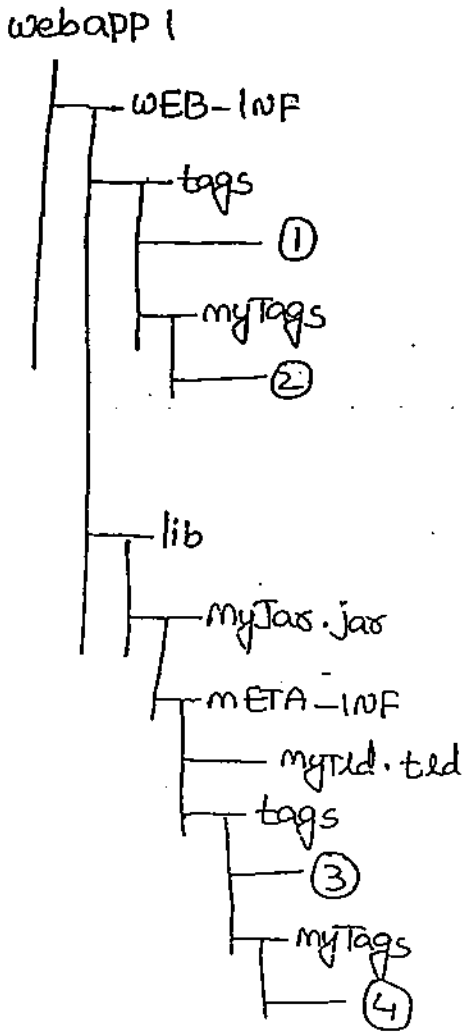
test.jsp

```
<%@ taglib prefix="mime" tagdir="/WEB-INF/tags" %>
<mime:header color="yellow" >
        This is body of Tag File
</mime:header>
```

header.tag

```
<%@ attribute name="colors" required="true"
              rtexprvalue="true" %>
<%@ tag body-content="tagdependent" %>
<h1> Hello --- This is from Tag File </h1>
<font color="${colors}" >
        <jsp:doBody />
</font>
```

where web Container looks for Tag Files.



1) Either Directly or Indirectly in WEB-INF / tags folder.

2) Either Directly or Indirectly in META-INF / tags folder present in jar file of WEB-INF / lib.

3) whenever we are Deploying tag file in Same jar then Compulsory we have to write tld File.

```
<taglib version="2.1">
    <tlib-version> 1.2.3 </t-v>
    <uri> www.durgasoft.com </uri>
    <tag-file>
        <name> myTag </name>
        <path> /META-INF/tags/myTag.tag
    </tag-file>                    </path>
</taglib>
```

test.jsp

```
<%@ taglib prefix="mime" uri="www.durgasoft.com" %>

<mime : myTagfile />

<mime : myTagfile />
```

tld File – myTld.tld

```
<taglib version="2.1">
    <tlib-version> 1.2.3 </t-v>
    <uri> www.durgasoft.com </uri>
    <tag-file>
        <name> myTagfile </name>
        <path> /META-INF/tags/myTag.tag </p
    <tag-file>
</taglib>
```

myTags.tag

<h1> This is from Tag File Deployed in Jar. </h1>

tagfile-Jar
- test.jsp
- WEB-INF
  - lib
    - durga.jar
      - META-INF
        - myTld.tld
        - tags
          - myTags.tag

→ first create durga
- META-INF
  - myTld.tld
  - tags
    - myTags.tag

→ create Jar file for durga folder

→ Place that Jar folder in lib.

⇒ jar -cvf durga.jar *.*

Disabling Scripting Language.

→ we can disable scripting language in JSP's by using <scripting-invalid> Tag.

```
<web-app>
    <jsp-config>
        <jsp-property-group>
            <url-pattern> *.jsp </u-p>
            <scripting-invalid> true </s-i>
        </jsp-property-group>
    </jsp-config>
</web-app>
```

→ once the scripting language is invalidate, we are not allow to use scripting elements in JSP. Violation leads to translation error. (Tomcat doesn't provide support).

Disabling Expression Language Globally

→ we can disable EL for a particular JSP by using "isELIgnored" attribute of page directive.

```
<%@ page isELIgnored="true" %>
```

→ we can disable EL at application level as follows.

```
<web-app>
    <jsp-config>
        <jsp-property-group>
            <url-pattern> *.jsp </u-p>
            <el-ignored> true </e-i>
        </jsp-property-group>
    </jsp-config>
</web-app>.
```