**:: JavaServer Page Fundamentals Q&A ::**

**:: Chapter One ::**

**Q. What is JSP technology?**

**Ans:** Java Server Page is a standard Java extension that is defined on top of the servlet Extensions. The goal of JSP is the simplified creation and management of dynamic Web pages. JSPs are secure, platform-independent, and best of all, make use of Java as a server-side scripting language.

**Q. What is JSP page?**

**Ans:** Java Server Pages (JSP) is a platform independent presentation layer technology that comes with SUN s J2EE platform. JSPs are normal HTML pages with Java code pieces embedded in them. JSP pages are saved to \*.jsp files. A JSP compiler is used in the background to generate a Servlet from the JSP page.

A JSP page is a text-based document that contains two types of text: static template data, which can be expressed in any text-based format such as HTML, SVG, WML, and XML, and JSP elements, which construct dynamic content.

**Q. What is Servlet? Why it is used?**

**Ans:** A Servlet is a server-side component that is capable of dynamically processing requests and constructing response in a protocol independent manner.

Servlet are best used in situations where a great deal of programmatic control is required, Such as decision making, database querying or accessing other enterprise resource. Servlet technology has been an extremely popular choice for building dynamic web applications such as e-commerce sites, online banking, and news portals.

**Q. Identify the advantages of JSP over Servlet.**Ans : a) Embedding of Java code in HTML pages b) Platform independence  
c) Creation of database-driven Web applications d) Server-side programming capabilities

**Q. What are the differences between JSP over Servlet?**

**Ans:** The differences between JSP over Servlet are**-**

1. Servlets embed content into logic, whereas JSP pages embed logic into content.
2. JSP pages contain markup interlaced with special JSP elements that provide logic for controlling the dynamic content. Servlets are built using Java classes that contain statements to output markup code.
3. JSP pages are preferred for presenting dynamic content in the presentation tier due to their greater readability, maintainability, and simplicity.

**Q. What are the basic advantages of Java Servlet?**

**Ans**: Some of the most basic advantages of servlet technology are as follows:

* Simplicity: Servlets are easy to write, and all the complicated threading and request delegating is managed by the servlet container.
* Extensibility: The Servlet API is completely protocol independent.
* Efficiency: the execution of a servlet doesn’t require a separate process to be spawned by the web server each time.
* Performance: Servlets are persistent, and their life cycle extends beyond that of each HTTP request.

**Q. What is JSP implementation Servlet?**

**Ans**: JSP technology is an extension of servlet technology, and it’s not surprising that each JSP page is converted into a servlet in order to provide the processing logic. JSP page being translated and compiled into a servlet in response to a request. This servlet is known as the JSP implementation Servlet.

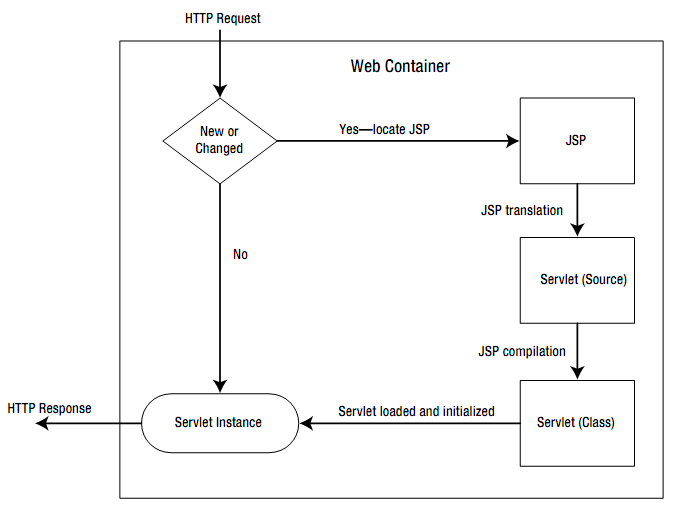
**Q What are the life-cycle methods of JSP?  
Ans**: Life-cycle methods of the JSP are:  
a) **jspInit():**The jspInit() method of the javax.servlet.jsp.JspPage interface is similar to the init() method of servlets. The container calls the jspInit() to initialize the servlet instance. It is called before any other method, and is called only once for a servlet instance. It can be overridden by a page author to initialize resources such as database and network connections, and to allow a JSP page to read persistent configuration data.

b)**\_jspService():**The \_jspService() method of the javax.servlet.jsp.HttpJspPage interface is invoked every time a new request comes to a JSP page. This method takes the HttpServletRequest and HttpServletResponse objects as its arguments. A page author cannot override this method, as its implementation is provided by the container.The container calls the \_jspservice() for each request and it passes the request and the response objects. \_jspService() method cann't be overridden.   
c) **jspDestroy():**The **jspDestroy()** method of the *javax.servlet.jsp.JspPage interface* **is invoked** by the container when a JSP page is about to be destroyed. This method is similar to the destroy() method of servlets. It can be overridden by a page author to perform any cleanup operation such as closing a database connection.

**The jspInit() and jspDestroy() methods can be overridden within a JSP page.**

**Q What is the life-cycle of JSP?**

**Ans:** JSP pages don’t directly return content to the client browser themselves. Instead, they rely on some initial server-side processing that converts the JSP page into the JSP page implementation class, which handles all requests made of the JSP.



According to the figure the JSP servlet container decides whether the JSP page has been translated before. If not, the JSP container starts the translation phase to generate the JSP page implementation servlet, which is then compiled, loaded and initialized, and used to service the request. If the JSP container detects that a JSP page has already been translated and hasn’t subsequently changed, the request is simply serviced by the implementation servlet that already exists inside the container.

**Q. Explain the phases of the life cycle of a JSP page.**

**Ans:** The life cycle of a JSP page can be split into four phases: translation, initialization, execution, and finalization.

1. **Translation:**

The first stage in the life cycle of a JSP page is known as the translation phase. When a request is first made for a JSP page, the JSP engine will examine the JSP file to check that it’s correctly formed and that the JSP syntax is correct to translate the JSP page into its page implementation class, which takes the form of a standard Java servlet. After the page’s implementation servlet has been created, it will be compiled into a class file by the JSP engine and will be ready for use. Each time a container receives a request, it first checks whether the JSP file has changed since it was last tran-slated. If it has, it’s retranslated so that the response is always generated by the most up-to-date implementation of the JSP file.

1. **Initialization:**

After the translation phase has been completed, the JSP engine load the generated class file and create an instance of the servlet in order to continue processing the initial request. After the servlet class is loaded, the container initializes the servlet instance with the jspInit() method.

1. **Execution**

After the web container has loaded and initialized the implementation servlet, the initial request is serviced by calling the \_jspService() method of the implementation servlet. The jspService()method provides all the functionality for handling a request and returning a response to the client. All the scriptlets and expressions end up inside this method, in the order in which they were declared inside the JSP page.

1. **Finalization**

The last phase in the life cycle is the finalization phase. jspDestroy() method is called by the servlet container when the page implementation servlet is about to be destroyed. After this method has been called, the servlet can no longer serve any requests.

**Q. What are the advantages of JSPs?**

**Ans:** JSPs has the following three key factors benefit:

**Reusability:**

Reusing code leads to increased maintainability and productivity, and higher quality because changes to such common functionality need to be made in only a single place. There are many techniques for aiding reusability provided by the JSP specification—such as JavaBeans components, custom tags, and tag libraries.

**Readability**

Another important best practice is that of readability. Wrapping up reusable functionality as custom tags or JavaBeans components removes complexity of core programming code from the page, there-fore making it cleaner, shorter, and more readable. Also, choosing appropriate names for custom tags can make a page more readable by page designers those who are responsible for the look and feel of a page rather than the mechanics of how it works.

**Maintainability**

The maintainability of an application is how well the system can be modified and fixed during its lifetime, which for a given application is typically hard to measure.

**Q. What is tag library? How it help reusability in JSP?**

**Ans**: A tag library (commonly known as a taglib) is simply a collection of one or more custom tags that are generally related in some way. The JSP 2.0 specification introduced a standard tag library known as the JSTL. The JSP 2.1 specification refines the JSTL. The JSTL’s core library contains tags that solve many of the common and recurring problems encountered when building JSP-based web applications. After the tags are bundled into a tag library, that tag library can be reused across the following:

• A single page

• The pages of a web application

• Different web applications

The ability to easily reuse custom tags across more than a single page illustrates the true potential of tag libraries when building web applications.

**Q. Explain JavaServer Pages Application Architecture.**

**Ans**: In recent times, two popular web-application architectures have been repeatedly used for -application design, and there are strengths and weaknesses to consider with both.

**Model 1 Architecture**

The simplicity of the Model 1 architecture is that each JSP page is entrusted to deal with its request entirely by itself, thereby generating a response and sending it back to the client, and for this reason it’s often known as page-centric. As long as the application is relatively small with few pages, the Model 1 architecture is a good choice because it’s quick and simple to put together. However, such a page-centric architecture can begin to introduce problems when used with larger, more complex app-lications. Some of the more common problems are-

* Maintainability Problems
* Reusability Problems
* Security Problems

**Model 2 Architechture(Model-View-Controller)**

The Model 2 architecture is a server-side implementation of the popular Model-View-Controller (MVC) design pattern. It utilizes a controller servlet to handle all the request processing and delegate requests to separate JSP components to provide the presentation, thereby making the best use of both technologies.

**Q. What are the advantages of Model-2 architecture over Model-1 architecture?**

**Ans: Maintainability**

Many of the maintainability problems associated with the Model 1 architecture were a direct result of implementing a controller and view as part of the same component: the JSP page. Because all the business processing and content generation were forced together, the result was messy pages that could be hard to maintain. By separating your application’s logic from its presentation by using MVC components, it’s far easier to develop cleaner code that focuses specifically on the job at hand, resulting ultimately in a more flexible and maintainable application.

**Security**

By providing an initial single point of access for potential requests, a servlet-based controller com-ponent is an excellent place to provide some form of authentication. The controller component is responsible for handling every request, security checks have to exist in only a single place, and of course any changes to the security mechanism have to be made only once. However, the Model-1 architecture required each page to provide similar security checks by itself, which provides a signi-ficant security hole if the developer forgets to provide it!

**Extensibility**

One of the best points about the Model 2 architecture is that all the processing logic is centralized. No longer does such code have to be placed in a scriptlet located deep within a JSP page, where it’s so much more difficult to access.

**Q. How does container allow web applications or JSP pages to be deployed or set up?**

**Ans:** Most containers allow web applications to be deployed in one of the following two forms:

**• Expanded directory format:** The web application in its predefined structure is simply

copied into the container’s deployment directory.

**• Web ARchive file (WAR):** The web application in its predefined structure is archived into a compressed WAR before being copied to the container’s deployment directory.

**Q. What is Context?**

**Ans**: The test directory that is used to house the web application is known as the context of the web application.

**Q. What are the JSP elements?**

**Ans:** This programmatic logic may be classified into the following JSP elements:

• Scripting elements

• Directives

• Action elements

**Q. What is Template Text?**

**Ans:** Any non-JSP code located inside a JSP page is known as template text. Template text can take any form as long as it’s text based. The most common form of template text is markup such as HTML or XML

**Q what are the JSP scripting elements?**

**Ans**: Scripting elements are used within a JSP page to manipulate objects and perform computa-

tions that enable the generation of dynamic content. Scripting elements can be classified into the following categories:

• Comments

• Declarations

• Scriptlets

• Expressions

• Expression language expressions

**Q What is JSP declaration?  
Ans**: JSP Decleratives are the JSP tag used to declare variables and methods. Declaratives are en-closed in the <%! %> tag and ends in semi-colon. It is possible to declare variables and functions in the declaration tag and can be use anywhere in the JSP. Here is the example of declaratives:

<%@page contentType="text/html" %>

<html>

<body>

<%!  
int cnt=0;  
private int getCount(){  
//increment cnt and return the value  
cnt++;  
return cnt;  
}  
%>

<p>Values of Cnt are:</p>

<p><%=getCount()%></p>

</body>

</html>

**Q What is JSP Scriptlet?  
Ans**: Scriptlets are small blocks of source code contained within the <% and %> delimiters that can be used to provide programming-style language functionality around a page’s content, thus making their output dynamic. Java code written inside scriptlet executes every time the JSP is invoked.   
Example:  
  **<%  
  //java codes  
   String userName=null;  
   userName=request.getParameter("userName");  
   %>**

**Q What is JSP Custom tags?  
Ans:** JSP Custom tags **are** user defined JSP language element. JSP custom tags are user defined tags that can encapsulate common functionality. For example you can write your own tag to access the database and performing database operations. You can also write custom tag for encapsulate both simple and complex behaviors in an easy to use syntax and greatly simplify the readability of JSP pages.

**Q What is expression in JSP?  
Ans:** Expression tag is used to insert Java values directly into the output. Syntax for the Expression tag is:   
**<%= expression %>**An expression tag contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file. The following expression tag displays time on the output:  
<%=new java.util.Date()%>

**Q What are scripting elements?**

**Ans:** JSP scripting elements **let** you insert Java code into the servlet that will be generated from the current JSP page. There are three forms:

**Expressions of the form <%= expression %>** that are evaluated and inserted into the output,

**Scriptlets of the form <% code %>** that are inserted into the servlet's service method,

**Declarations of the form <%! code %>** that are inserted into the body of the servlet class, outside of any existing methods.

**Q What are implicit objects in JSP?**

**Ans:** Implicit objects in JSP **are** the Java objects that the JSP Container **makes** available to developers in each page. These objects need not be declared or instantiated by the JSP author. They are automatically instantiated by the container and are accessed using standard variables; hence, they are called implicit objects. The implicit objects available in JSP are as follows:

|  |  |
| --- | --- |
| **Objects** | **Description** |
| request | Request Refers to the current request from the client. This is the **HttpServletRequest** object associated with the request. |
| response | Page Refers to the current response to the client. This is the **HttpServletResponse** object associated with the response to the client. |
| out | Page Refers to the outputstream. This is the **PrintWriter** object used to send output to the client. |
| session | Session refers to the user’s session. This is the **HttpSession** object associated with the request. |
| application | Application Same as ServletContext. Refers to the web application’s environment.This is the **ServletContext** object associated with application context. |
| config | Page same as ServletConfig. Refers to the servlet’s configuration. This is the **ServletConfig** object associated with the page. |
| pageContext | Page Refers to the page’s environment.This encapsulates use of server-specific features like higher performance **JspWriters**. |
| page | Page Refers to the page’s Servlet instance. This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. |
| Exception | Page exception created on this page. Used for error handling. Only available if it is an errorPage with the following directive: <%@ page isErrorPage="true" %>. The **Exception** object allows the exception data to be accessed by designated JSP. |

**Q What are JSP directives?**

**Ans**: JSP directives are used for passing important information to the JSP engine. JSP directives

generate no output, they provide a powerful mechanism for providing page-level information that is typically used during the compilation and translation phases. They are used to set global values such as a class declaration, method implementation, output content type, etc. JSP page authors have the following three types of directives at their disposal:

• page directives

• include directives

• taglib directives

Each type of directive provides different information to the JSP engine or signifies some required behavior of the generated servlet. All three directive types must be declared between <%@ and %> directive delimiters and take the following form:

<%@ directive {attribute="value"}\* %>

**Q What is page directive?**

**Ans:** The first directive type is the page directive, which is used to define any page-dependent pro-perties that a JSP page may have, such as library dependencies, buffering, or error-handling require-ments. Typically, the page directive is found at the top of almost all of JSP pages. There can be any number of page directives within a JSP page (although the attribute – value pair must be unique).The syntax of the include directive is: **<%@ page attribute="value">**

Example:<%@ include file="header.jsp" %>

**Q What are the attributes of page directive?**

**Ans**: There are fifteen attributes defined for a page directive of which the important attributes are as follows:

**import:** It specifies the packages that are to be imported.

**session:** It specifies whether a session data is available to the JSP page.

**contentType:** It allows a user to set the content-type for a page.

**isELIgnored:** It specifies whether the EL expressions are ignored when a JSP is translated to a servlet.

**Q What is the include directive?**

**Ans:** The include directive is used to pass information to the JSP engine during the translation phase, to control how the page implementation class is generated. This enables a user to reuse the code without duplicating it, and includes the contents of the specified file at the translation time.

The syntax of the include directive is as follows:

**<%@ include file = "FileName" %>**

This directive has only one attribute called file that specifies the name of the file to be included. A JSP page may contain any number of include directives at any position in the page to indicate the exact positions where the content from the included resource should be inserted.

**Q What is the difference between include directive and include action?**

|  |  |
| --- | --- |
| **Include directive** | **Include action** |
| The include directive, includes the content of the specified file during the translation phase–when the page is converted to a servlet. | The include action, includes the response generated by executing the specified page during the request processing phase–when the page is requested by a user. |
| The include directive is used to statically insert the contents of a resource into the current JSP. | The include standard action enables the current JSP page to include a static or a dynamic resource at runtime. |
| Use the include directive if the file changes rarely. It’s the fastest mechanism. | Use the include action only for content that changes often, and if which page to include cannot be decided until the main page is requested. |

**Q. Explain taglib directives.**

**Ans:** A tag library contains a collection of actions (also known as tags) that can be grouped together to perform some form of logic. These actions are XML based, so their use is considerably easier for a non-Java-speaking UI designer. Tag libraries come in two flavors these days. The first is custom tag libraries and second come from the JSTL, which contains a set of useful actions

that are applicable in some form to almost every web application in use today. The syntax for the taglib directive is as follows:

<%@ taglib {uri="/tagLibraryURI" | tagdir="/WEB-INF/tags/dirname"

prefix="tagPrefix" %>

**Q. What is Tag Library Descriptor (TLD)?**

**Ans:** To use a custom tag library, the web container needs to be made aware of specific information about the library itself. A special file called a tag library descriptor (TLD) is used for this purpose.The TLD file contains essential information about each of the custom actions or tags that are include-ed inside the tag library, such as which attributes are permitted by which tags, whether the tags accept body content, and so on.

**Q. What are taglib attributes?**

**Ans**: . Attributes for the taglib Directive

**uri :**

It can be either an absolute or a relative Uniform Resource Identifier (URI) identifies the TLD, and therefore, the tag library that is associated with the prefix.

**tagdir :**

Indicates this prefix is to be used to identify tag extensions installed in the WEB-INF\tags directory or a subdirectory. An implicit tag library descriptor is used. A translation error must occur if the value does not start with WEB-INF\tags. A translation error must occur if the value does not point to a directory that exists. A translation error must occur if used in conjunction with the uri attribute.

**prefix:**

Indicates a uniquely identifiable string, which is used in the <prefix:tagname> declaration to identify the particular tag in use. Note that prefixes that start with any of the following aren’t allowed because

they’re reserved by Sun: jsp, jspx, java, javax, servlet, sun, and sunw. All prefixes must follow the naming convention as specified in the XML namespaces specification. The current version of the JSP specification doesn’t support empty prefixes.

**Q. How many ways the taglib directive can be used to make a tag library available to a**

**Jsp page authors?**

**Ans:** There are four ways that the taglib directive can be used to make a tag library available to JSP page authors.

**Option 1—Absolute URI**

The first option for using the taglib directive passes an absolute value in the uri attribute that

repressents the location of the TLD file:

<%@ taglib uri="/WEB-INF/tlds/myTaglib.tld" prefix="myPrefix" %>

The location of the TLD file is explicitly given by the uri attribute.

**Option 2—Relative URI**

The second form of the taglib directive uses a relative URI to indicate the location of the TLD file. The taglib directive no longer explicitly declares the location of the TLD file, but instead relies upon the existence of a relative URI mapping in the application’s deployment descriptor.

**Option 3—Packaged JAR**

The third use of the taglib directive provides an absolute path to an external JAR file. The JAR file contain all the tag handler classes as well as the TLD file, which must be located inside the META-INF directory of the JAR file.

**Option 4—Tag Files**

The final use of the taglib directive provides a path to a directory that contains tag files. Tag files are special JSP files that end with the suffix .tag or .tagx. They can contain the same kinds of JSP elements as in a JSP page, with some exceptions.

**Q. What do you understand by Action Elements?**

**Ans:** It is difficult to read and maintain JSP pages that are full of scriptlet code. A better alternative is to use existing actions (tags) provided by a tag library that encapsulate pieces of functional logic. These actions make JSP pages much cleaner and more readable. Also, because they’re XML tag-based, they are usable by a non-Java UI developer. In JSP 2.1 there are three types of action elements:

• Standard actions

• Custom actions

• JSTL actions

**Q What are the JSP standard actions?**

**Ans**: The JSP standard actions affect the overall runtime behavior of a JSP page and also the response sent back to the client. They can be used to include a file at the request time, to find or instantiate a JavaBean, to forward a request to a new page, to generate a browser-specific code, etc. Ex: **include, forward, useBean,etc. object**

**Q What are the standard actions available in JSP?**

**Ans**: The standard actions available in JSP are as follows:

**<jsp:include>:** It includes a response from a servlet or a JSP page into the current page. It differs from an include directive in that it includes a resource at request processing time, whereas the include directive includes a resource at translation time.

**<jsp:forward>:** It forwards a response from a servlet or a JSP page to another page.

**<jsp:useBean>:** It makes a JavaBean available to a page and instantiates the bean.

**<jsp:setProperty>:** It sets the properties for a JavaBean.

**<jsp:getProperty>:** It gets the value of a property from a JavaBean component and adds it to the response.

**<jsp:param>:** It is used in conjunction with <jsp:forward>;, <jsp:, or plugin>; to add a parameter to a request. These parameters are provided using the name-value pairs.

**<jsp:plugin>:** It is used to include a Java applet or a JavaBean in the current JSP page.

**Q What is the <jsp:useBean> standard action?**

**Ans**: The <jsp:useBean> standard action is used to locate an existing JavaBean or to create a Java Bean if it does not exist. It has attributes to identify the object instance, to specify the lifetime of the bean, and to specify the fully qualified classpath and type.

**Q What are the scopes available in <jsp:useBean>?**

**Ans**: The scopes available in <jsp:useBean> are as follows:

**page scope::** It specifies that the object will be available for the entire JSP page but not outside the page. request scope: It specifies that the object will be associated with a particular request and exist as long as the request exists.

**application scope:** It specifies that the object will be available throughout the entire Web application but not outside the application.

**session scope:** It specifies that the object will be available throughout the session with a particular client.

**Q What is the <jsp:forward> standard action?**

**Ans**: The <jsp:forward> standard action forwards a response from a servlet or a JSP page to another page. The execution of the current page is stopped and control is transferred to the forwarded page. The syntax of the <jsp:forward> standard action is :    
<jsp:forward page="/targetPage" />  
Here, targetPage can be a JSP page, an HTML page, or a servlet within the same context. If anything is written to the output stream that is not buffered before <jsp:forward>, an IllegalStateException will be thrown.

**Q What is the <jsp:include> standard action?**

**Ans**: The <jsp:include> standard action enables the current JSP page to include a static or a dynamic resource at runtime. In contrast to the include directive, the include action is used for resources that change frequently. The resource to be included must be in the same context.The syntax of the <jsp:include> standard action is as follows:  
<jsp:include page="targetPage" flush="true"/>   
Here, targetPage is the page to be included in the current JSP.

**Q Differentiate between pageContext.include and jsp:include?**

**Ans**: The <jsp:include> standard action and the pageContext.include() method are both used to include resources at runtime. However, the pageContext.include() method always flushes the output of the current page before including the other components, whereas <jsp:include> flushes the output of the current page only if the value of flush is explicitly set to true as follows:

<jsp:include page="/index.jsp" flush="true"/>

**Q What is the jsp:setProperty action?**

**Ans**: jsp:setProperty is used to give values to properties of beans that have been referenced earlier. This can be done in two contexts. First, using jsp:setProperty after, but outside of, a jsp:useBean element, as below:

<jsp:useBean id="myName" ... />

<jsp:setProperty name="myName" property="myProperty" ... />

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.   
A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

<jsp:useBean id="myName" ... >

...

<jsp:setProperty name="myName"

property="someProperty" ... />

</jsp:useBean>

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

**Q What is the jsp:getProperty action?**

**Ans**: The **<jsp:getProperty>** action is used to access the properties of a bean that was set using the <jsp:getProperty> action. The container converts the property to a String as follows:

If it is an object, it uses the toString() method to convert it to a String. If it is a primitive, it converts it directly to a String using the valueOf() method of the corresponding Wrapper class. The syntax of the <jsp:getProperty> method is: <jsp:getProperty name="Name" property="Property" /> Here, name is the id of the bean from which the property was set. The property attribute is the property to get. A user must create or locate a bean using the **<jsp:useBean>** action before using the <jsp:getProperty> action.

**Q What is the <jsp:param> standard action?**

**Ans**: The <jsp:param> standard action is used with <jsp:include> or <jsp:forward> to pass parameter names and values to the target resource. The syntax of the <jsp:param> standard action is as follows:   
<jsp:param name="paramName" value="paramValue"/>

**Q.What is the jsp:plugin action ?**

**Ans**: This action lets you insert the browser-specific OBJECT or EMBED element needed to specify that the browser run an applet using the Java plugin.

**Q. Explain Custom Actions.**

**Ans**: Custom actions are another mechanism for encapsulating functionality into reusable

components for use inside JSP pages. Unlike JavaBeans components, custom actions have full

access to their environment (such as the request and session objects), which makes it far easier

to provide functionality suitable for a website.

**Q. What are the tag libraries provided by the JSTL?**

**Ans:** The JSTL provides four new tag libraries that may be used in a similar manner to the standard tags you saw earlier:

•Core

• Internationalization (I18n) and Formatting

• XML

• SQL

**:: Chapter Two ::**

**Q. Where servlets are appropriate and inappropriate**

Ans: Servlets are best used in situations where a great deal of programmatic control is required, such as decision making, database querying, or accessing other enterprise resources. However, there are also many times when using a servlet isn’t appropriate. These situations occur primarily when a lot of display formatting is required.

**Q. What interfaces are used for servlet?**

**Ans:** The javax.servlet package is composed of fourteen interfaces. The web container implements

these seven interfaces:

• ServletContext

• ServletConfig

• ServletResponse

• ServletRequest

• RequestDispatcher

• FilterChain

• FilterConfig

The remaining seven interfaces are implemented by the web-application developer to provide the application’s functionality:

• Servlet

• ServletContextListener

• ServletContextAttributeListener

• ServletRequestAttributeListener

• ServletRequestListener

• SingleThreadModel

• Filter

**Q. Explain shortly Servlet, ServletContext, ServletConfig, ServletContextListener, ServletContextAttributeListener, RequestDispatcher, ServletRequest?**

**Ans**: **Servlet:** The Servlet interface is key in developing servlets. This interface defines the life-cycle methods of a basic servlet: initialization, service, and destruction. This interface also provides a method to obtain an instance of the ServletConfig interface.

**ServletConfig**: The container uses the ServletConfig interface to pass initialization information to a servlet. The ServletConfig interface also has a way to get hold of an instance of ServletContext for the current web application (via the getServletContext() method).

**ServletContext:** The ServletContext interface is the web application’s view on the web container. This allows a web application to use the services of the container, such as logging and request dispatching.

**ServletContextListener**: The ServletContextListener interface is a life-cycle interface that programmers can implement to listen for changes to the state of the ServletContext object. This means that programmers can choose to be informed of events such as the destruction or creation of a

ServletContext object. This allows the developer to perform application startup- and shutdown type functionality (for example, logging creation or destruction of the resource, or initializing application-level constants) within a web applications.

**ServletContextAttributeListener**: Implementations of the ServletContextAttributeListener interface can perform similar functionality, but the events that they are notified about relate to the modification (adding, changing, deleting) of attributes on the servlet context.

**RequestDispatcher**: The RequestDispatcher interface manages client requests by directing them to the appropriate resources on the server. The developer can use this interface to redirect the application to different pages and servlets.

**ServletRequest**: The ServletRequest interface encapsulates all the information that is transmitted to a servlet through its service() method during a single client request. A ServletRequest object created by the container provides methods to access parameter names, parameter values, and attributes, as well as an input stream.

**Q.What are the abstract classes of javax.servlet package?**

**Ans**: there are nine classes contained within the javax.servlet package. They are as follows:

• GenericServlet

• ServletContextEvent

• ServletContextAttributeEvent

• ServletInputStream

• ServletOutputStream

• ServletRequestEvent

• ServletRequestAttributeEvent

• ServletRequestWrapper

• ServletResponseWrapper

The GenericServlet abstract class can be used to develop protocol-independent servlets and requires only that subclasses implement its service() method. For servlets intended to function in a web context, it’s more common to extend the HttpServlet abstract class.

The two event classes, ServletContextEvent and ServletContextAttributeEvent, are used for notification about changes to the ServletContext object and its attributes.

The two event classes, ServletRequestEvent and ServletRequestAttributeEvent, are used for notification about changes to the ServletRequest object and its attributes.

The ServletInputStream and ServletOutputStream abstract classes provide the ability to read and write binary data from and to the client. Classes that extend either of these abstract classes must provide an implementation of the java.io.InputStream.read() and java.io.OutputStream.write() methods, respectively.

Last, the wrapper classes ServletRequestWrapper and ServletResponseWrapper provide useful implementation of the ServletRequest and ServletResponse interfaces. These can be used or subclassed to give developers the ability to adapt the standard behavior of these objects for their own applications’ needs.

**Q. Explain the life cycle of a servlet.**

**Ans:** The javax.servlet.Servlet interface defines the methods that all servlets must implement

and, among others, three methods that are known as life-cycle methods:

* public void init(ServletConfig config) throws ServletException
* public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException
* public void destroy()

These life-cycle methods are each called at separate times during the life span of a servlet, from the initial creation to the moment it’s removed from service and destroyed. These methods are called in the following order:

1. When the servlet is constructed, it is initialized with the init() method.

2. Any requests from clients are handled initially by the service() method before delegating to the doXxx() methods in the case of an HttpServlet. The service() method is responsible for processing the request and returning the response.

3. When the servlet needs to be removed from service, it’s destroyed with the destroy() method, then garbage collected and finalized. When the container decides to take a servlet out of service, it first ensures that any service() method calls have been completed.

**Q. Explain HttpServlet abstract class.**

**Ans:** HttpServlet abstract class is the main class in the javax.servlet.http package. This class extends from the javax.servlet.GenericServlet class. This means that all functionality provided by this class is available to HTTP servlets.It has several new methods that provide protocol-specific functionality to perform a different task depending upon the manner in which it’s being called.The request-handling methods that are provided are as follows:

protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException

The doGet() method is intended to retrieve an entity from the server as referenced by a request URL.

protected void doHead(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException

The doHead() method is simply a GET request that is intended to return only the HTTP header information.

protected void doPost(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException

The doPost() method is intended to allow posting of information (forms and so on) to the server.

protected void doPut(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException

The doPut() method is used to upload a file to a server in a manner similar to the FTP.

protected void doOptions(HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException

protected void doTrace(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException

The doOptions() and doTrace() methods allow you to override the behavior of HTTP.

There is almost no reason to override either of these methods unless the servlet implements

functionality beyond the HTTP 1.1 specification.

**Q. Explain HttpServletRequestabstract class.**

Ans: the HttpServletRequest interface used to access the HTTP-specific information of the request of HttpServlet. This interface extends the ServletRequest interface. Some of the useful operations that is possible to perform through requests are as follows

* Retrieving HTTP Request Header Information
* Retrieving Path Information

**Q. Explain HTTP header information.**

**Ans**: HTTP headers store a wide range of information about the user and the request, and they are

transmitted between a user (usually a browser) and a web server during each request. HTTP header information is separate from the body of a request and provides some useful information that can be used by a web component (servlet or JSP page) when constructing a response.

HTTP Header Names and Their Usage

Header Usage

Date The date and time the request was served

Accept The media types accepted by the client

Accept-Encoding The types of data encoding that the browser knows how to decode

Connection Whether the client can handle persistent HTTP connections

Content-Length The length of the body in bytes, or –1 if the length is unknown

Cookie The cookies returned by the client browser to the server

Host The host and port of the original URL

Referrer The URL of any referring web page

User-Agent The client or browser making the request

**Q. Explain HttpServletResponse abstract class.**

Ans; the HttpServletResponse interface used to access the HTTP-specific information of the request of HttpServlet. This interface extends the ServletResponse interface. Some of the useful operations that is possible to perform through requests are as follows

* Setting an HTTP Response Header and Setting the Content Type of the Response
* Acquiring a Text Stream for the Response
* Acquiring a Binary Stream for the Response
* Redirecting an HTTP Request to Another URL

**Q. What is deployment descriptor? Which things make deployment descriptor valid?**

**Ans:** The deployment descriptor describes the web application to the container. The deployment descriptor file is perhaps the single most important item of your web application. For the deployment descriptor to be valid for web applications using the Servlet 2.5 specification, several things must be true:

• The file must conform to the web application XML Schema, shown here:

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/➥

web-app\_2\_5.xsd"

version="2.5">

• The deployment descriptor must be a well-formed XML file.

• The deployment descriptor must be named web.xml.

• The deployment descriptor must reside at the top level of the WEB-INF directory of your

web application.

In a nutshell, the deployment descriptor conveys the elements and configuration information of a web application to developers, assemblers, and deployers. All manner of information is defined in the deployment descriptor: from information about the web application itself to information about its constituent parts, and most important, how those parts are assembled into a complete web application.

**Q. What is Servlet Definitions?**

**Ans:** The <servlet> tag contains several child tags that give information about the declaration

of the servlet. This information includes the unique name that the servlet is registered with in

this web application, and the full name of the class that implements the servlet’s functionality.

The <servlet-name> tag gives the servlet’s unique name within the web application.

The <servlet-class> tag gives the fully qualified class name of the class that implements

the functionality of this servlet.

It also contains an <init-param> tag. This tag allows to specify initialization parameters for the servlet.

Example of a servlet is defined here:

**<servlet>**

**<servlet-name>Servlet1</servlet-name>**

**<servlet-class>com.apress.projsp.Servlet1</servlet-class>**

**<init-param>**

**<param-name>version</param-name>**

**<param-value>0.1b</param-value>**

**</init-param>**

**</servlet>**

**Q.Explain Servlet Mappings.**

**Ans**: After defining the servlet through the <servlet> tag, it’s necessary to map it to a particular

URL pattern. This is necessary so that the web container knows which requests to send to a

particular servlet. The unique servlet name that is defined in the <servlet> tag (referenced here as

<servlet-name>) is mapped to a URL pattern referenced here in a <url-pattern> element:

<servlet-mapping>

<servlet-name>Servlet1</servlet-name>

<url-pattern>/home.html</url-pattern>

</servlet-mapping>

servlets can be mapped to more than one URL through the use of wildcards in the <url-pattern> child tag of the <servlet-mapping> tag. For example, the following maps every URL encountered to the same servlet, which allows you to have a central servlet that handles all requests:

<servlet-mapping>

<servlet-name>ValidatorServlet</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

**Q.What is Error page?**

**Ans**: Error pages allow responding to problems with custom pages that offer specific information about the trouble at hand. These errors can include Java exceptions as well as HTTP errors. For example, the result of a page not being found. The <error-page> tag has two children: <exception-type> and <location>. The <exception-type> child tag defines the exception to catch, and <location> defines the page or resource to display on encountering the error defined.

**Q. What can be done by the JavaServer Pages Configuration Elements?**

**Ans**: JavaServer Pages Configuration Elements allow doing the following:

• **Enable or disable EL evaluation:** the <el-ignored> element used. By default, the EL evaluation is enabled for web applications using a Servlet 2.4 or Servlet 2.5 web.xml.

**• Enable or disable scripting elements:** <scripting-invalid> element is used. By default, scripting is enabled

**• Indicate page-encoding information:** Using the <page-encoding> element it’s possible defining the encoding for a group of JSP pages.

**• Automatically include preludes and codas:** Through the use of the <include-prelude> and <include-coda> elements, it’s possible to automatically include a page before and after the evaluation of a group of pages.

The <include-prelude> element is a context-relative path that must correspond to an element in the web application. When the element is present, the given path will be automatically included (as in an include directive) at the beginning of each JSP page in this <jsp-property-group>.

The <include-coda> element is a context-relative path that must correspond to an element in the web application. When the element is present, the given path will be automatically included (as in an include directive) at the end of each JSP page in this <jsp-property-group>.

**collected by qm.qutub mia –ESAD -J2EE-01713955862**

Further added, Recollect and Edited by:

[Yasin\_javed@yahoo.com](mailto:Yasin_javed@yahoo.com)

# JSP Operators:

JSP supports all the logical and arithmatic operators supported by Java. Following table give a list of all the operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.Within an expression, higher precedenace operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# JSP Literals:

The JSP expression language defines the following literals:

* **Boolean:** true and false
* **Integer:** as in Java
* **Floating point:** as in Java
* **String:** with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
* **Null:** null

**Chapter-3**

**Q. What is the basic syntax of Expression language? What is the difference between them?**

**Ans**: No matter where the EL is used, it’s always invoked in a consistent manner, via the construct ${expr} or #{expr}, where expr is the EL expression that is wished to have evaluated.

The syntax of ${expr} and #{expr} are equivalent and can be used interchangeably. However, when used with some other Java Platform, Enterprise EditionAPI, the other API may enforce restrictions on the use of ${expr} and #{expr}. Specifically, when used with JSP pages, the two forms cannot be used interchangeably. Within a JSP page, ${expr} is used for expressions that are evaluated immediately, whereas #{expr} is used for expressions for which evaluation is deferred.

**Q. How to use Expression language in a JSP file? Explain.**

**Ans**: It is possible to use the EL in the same places as scriptlet has been used, for example:

**• Within attribute values for JSP standard and custom tags:** Using the EL within the attributes of a custom tag in a JSP page allows you to dynamically specify the attribute values for a custom tag. This is an extremely powerful mechanism. For example-

<myTagLibrary:myTag counter="<%= 1+1 %>" />

**• Within template text (that is, in the body of the page:** Example-<td colspan="2">Hello ${param['name']}</td>

**Q. What are the reserved words?**

**Ans**: A reserved word (also known as a keyword) is one that has a special meaning within the language. This means that reserved word cannot be used to represent anything else, such as a variable identifier. The following are reserved words in the JSP EL: and

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| eq | and | gt | true | instanceof | or | ne | lt | false | empty |
| not | if | null | ge | le | div | mod |  |  |  |

**Q. How to disable scriplets and Expression language in jsp page?**

**Ans**: **Disabling scriplets:** EL is intended to replace the use of Java scriptlets in developing JSP-based web applications.It’s possible to disable the evaluation of scriptlets through configuration parameters. To disable scriptlets within a page the web.xml deployment descriptor is used by choosing to disable evaluation for a single page, a set of pages, or for the entire application. The tags that is required to add to the deployment descriptor are within the <jsp-config> element. The following example disables scriptlets for all JSP pages within an application:

<jsp-config>

<jsp-property-group>

<url-pattern>\*.jsp</url-pattern>

<scripting-invalid>true</scripting-invalid>

</jsp-property-group>

</jsp-config>

The <url-pattern> element can represent a single page, for example:

<url-pattern>/test.jsp</url-pattern>

It can also represent a set of pages, for example:

<url-pattern>/noscriptlets/</url-pattern>

**Disabling Expression language:** it is possible to disable EL evaluation in two ways:

**• Individually on each page by using the page directive:** To disable the EL for a single page, it’s simplest to use the isELIgnored attribute of the page directive in the header of the page:

<%@ page isELIgnored="true" %>

**• Within the web.xml file by using a JSP configuration element:** Within the web.xml file, it is possible to disable for a single page, a set of pages, or the entire application. The following XML example to disable the EL for an entire application:

<jsp-property-group>

<url-pattern>\*.jsp</url-pattern>

<el-ignored>true</el-ignored>

</jsp-property-group>

The <url-pattern> element can represent a single page, for example:

<url-pattern>/test.jsp</url-pattern>

It can also represent a set of pages, for example:

<url-pattern>/noel/</url-pattern>

**Q. What is the operator precedence in expression language?**

**Ans**: Operators in expression language follow the following precedence list:

• []

• ()

• - (unary) not ! empty

• \* / div % mod

• +- (binary)

• <><=>=ltgtlege

• == != eq ne

• && and

• || or

**Q. What is JavaBean? How to use javaBean in JSP page? What are the scopes of JavaBean?**

**Ans**: A Java Bean is a reusable software component that can be visually manipulated in builder tools. JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions can be a JavaBeans component.

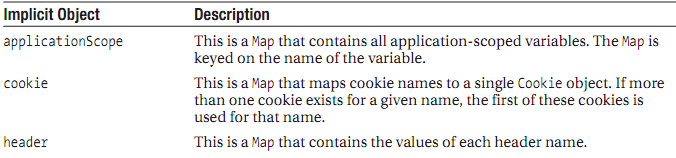
There are several ways to retrieve JavaBeans component properties. Two of the methods (the jsp:getProperty element and an expression) convert the value of the property into a String and insert the value into the current implicit out object:

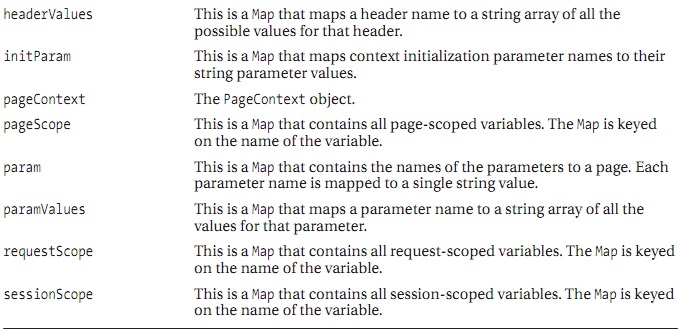
* <jsp:getProperty name="*beanName*" property="*propName*"/>
* <%= *beanName*.*getPropName*() %>

For both methods, *beanName* must be the same as that specified for the id attribute in a useBean element, and there must be a get*PropName* method in the JavaBeans component.

**Q. What are the Expression Language implicit Objects? Explain shortly?**

**Ans**:





**Q. What is TLD?**

**Ans**: A TLD is an XML file that declares a tag library. The TLD contains information relating to

the tags in the library and the classes that implement them. The TLD also contains the decla-

rations and mappings of EL functions. Each TLD can describe zero or more **static functions**.

**Q. Show with an example how to declare EL functions in a TLD file?**

**Ans**: TLD entries used to declare a function:

<taglib>

...

<function>

<name>nickname</name>

<function-class>com.apress.projsp.Functions</function-class>

<function-signature>

java.lang.String sayHello(java.lang.String, javax.servlet.jsp.PageContext)

</function-signature>

</function>

</taglib>

**Q. Show with example how to declare EL functions in a JSP page?**

**Ans**: <%@ taglib prefix="Source" uri="/WEB-INF/tags/taglib.tld"%>

<html>

<head>

<title>Source</title>

</head>

<body>

<pre>${Source:source(param.name, pageContext)}</pre>

</body>

</html>

**Q. How to add taglib in the deployment descriptor?**

**Ans**: Following XML code is written to to add a taglib in the deployment descriptor.

<taglib>

<taglib-uri>http://java.sun.com/jstl/core</taglib-uri>

<taglib-location>/WEB-INF/tlds/c.tld</taglib-location>

</taglib>

Where <taglib-uri>

<taglib-location>

**Q. What is the difference between expression language functions and Custom tags?**

**Ans**: The differences between expression language functions and Custom tags are as follows

• A tag provides easy access to pageContext and other variables; functions do not. To access these implicit objects within a function, it is necessary to pass them in as a parameter.

• Functions do not provide functionality to process a body (they don’t have one), whereas

tags do.

•Functions are much simpler to write than tags; therefore, they provide a great opportunity to

write small, self-contained pieces of functionality.

• Functions are ideal for reuse existing Java code in a web context than custom tags.

**Chapter-4**

**Q. What are the requirements to use JSTL?**

**Ans**: To be able to use JSTL following things are required

• At least a Servlet 2.3– and JSP 1.2–compliant container

• An implementation of the JSTL specification

**Q. Which jar files are required?**

**Ans**: There are two JAR files that are required

• jstl.jar

• standard.jar

**Q. Write the categories of tags contained by the JSTL.**

**Ans**: The JSTL contains four broad categories of tags identified by the TLDs:

1. c.tld,
2. fmt.tld,
3. x.tld and
4. sql.tld

**Q. What is JSTL?**

**Ans**: JSP Standard tag library or JSTL provides JSP authors a better way to make the JSP page cleaner and more friendly. JSTL likes HTML tags is XML based tags therefore it is very familiar to the JSP authors who are working with (X)HTML intensively.

**Q. How many tag libraries consist of JSTL?**

**Ans**: JSP standard tags library can be divided into four tag libraries which are:

* Core tags
* Internationalization (i18n) and formatting tags
* Relational database access tags
* XML processing tags

**Q. What is the goal of JSTL?**

**Ans**: The goals of those tag libraries above are:

* Simplify the task of writing JSP page by providing friendly XML base tags
* Provide reusable logic from page's presentation
* Make the JSP page easier to read and maintain

**Q. Explain the tag libraries?**

**Ans: The Core tag library**: Core tags provide the core functionality actions to JSP. These tags make the most common actions easier to achieve in a more effective way to everyday problems experienced in almost every JSP-based web application. Core tags specify several actions such as displaying content based on runtime conditions or iterating over a collection of item and URL managing.

**The Internationalization and Formatting Tag Library:** The Internationalization and Formatting tag library provides a series of actions to aid in the use of the three key components associated with internationalization: locales, resource bundles, and base names.

**The SQL Tag Library:** Accessing database is a most major task of web applications. JSTL provides a list of standard tags to manipulate data such as select, insert, update and delete from the relational databases.

**The XML Processing Tag Library:** XML becomes a standard of enterprise web application for exchanging data. Manipulate XML effectively therefore is very important for most web applications and of course JSTL also provides a list of tags for processing from XML parsing to XML transformation.

**Q. What are the main functional areas of the JSTL core library?**

**Ans**: The JSTL core library can be split further to expose its main functional areas:

• Scoped variable manipulation

• Conditionals

• Looping and iteration

• URL manipulation

**Q. Explain <c:out> action.**

**Ans**: <c:out> action is used to evaluate a variable or expression and output it. <c:out> is similar to the expression <%=  expression%>. The usage of <c:out> action is as follows:

<c:out value="variable" default="default value"/>  
<c:out value="expression" default="default value" />

There are two attributes:

1. Value:  You can put a variable or expression here
2. default value: if the variable or expression is evaluated as null, the default value will be used for output.

The other form of the <c:out> action is as follows:

<c:out value="expression">  
default value  
</c:out>

If the expression is evaluated as null, the body between opening and closing tag will be used for output.

**Q. Explain <c:set> action.**

**Ans**: JSTL provides <c:set >action to init or set a variable of the web application in a specific scope. The usage of the <c:set> is as follows:

<c:set var = "variable"  
       value = "value"  
       scope = "scope" />

In the ***var*** attribute used to declare or refer to a variable. The ***value*** attribute specify the value of the variable. To set the scope of the variable, ***scope*** attribute is used. The ***scope*** attribute accepts any valid JSP variable scopes such as ***page***, ***request***, ***session*** and ***application***.

The other usage of c:set action is used to set the value of a property of an object:

<c:set target= "object"  
       property = "property name"  
       value = "value of property" />

**Q. Explain <c:remove> action, <c:catch> Action**

**Ans**: <c:remove> action removes a variable from a specific application scope The <c:catch> action provides a simple mechanism for catching any java.lang.Throwable exceptions that are thrown by any nested actions.

**Q. Explain some JSTL functional areas of conditions.**

**Ans**: JSTL provides all basic conditionals to make the logic flow of JSP page easier to read and maintain. The conditionals includes: if condition and looping condition.

## <c:if> action:The most basic and simpliest condition is <c:if> action. It is used to output its body content based on a boolean expression. If the result of the expression is true, the body content will be processed by JSP container and output will be returned to the current *****JspWriter*****. The syntax of <c:if> action is as follows:

 <c:if test="expression">  
<%-- body content listed here --%>  
</c:if>

The expression is evaluated in the attribute ***test.***

## Multiple choices with <c:choose><c:when> and <c:otherwise> actions: If there is a set of mutually conditions then <c:choose><c:when> and <c:otherwise> are used instead of using multiple <c:if>. Here is the usage of those actions:

<c:choose>  
<c:whentest="expression1">  
<%-- body content for expression 1  -->  
</c:when>  
<c:when test="expression2">  
<%-- body content for expression 2  -->  
</c:when>  
...  
<c:otherwise>  
<%-- body content for otherwise  -->  
</c:otherwise>  
</c:choose>

In a range of conditions, if one of them is evaluated as true, the body content of that <c:when>branch will process and output to the current JspWriter and then no processing is performed. If none of conditions in <c:when> branch is true, the body content of <c:otherwise> branch will process and output to the current JspWriter. The combination of <c:choose> <c:when> and <c:otherwise> actions works like if elseif and else condition.

# Q. Explain JSTL Looping and Iteration Actions.

**Ans**: , JSTL provides you two useful actions for looping and iteration: for general data and for string of tokens.

## The <c:forEach> action: The <c:forEach> action is very useful. You can loop over a collection or you can iterate number of times. There are two usages of <c:forEach> action. Let's take a look at the first one which you can use <c:forEach>for loop over a collection.

<c:forEach(var="var"  
        items="collection"  
        varStatus="varStatusName">  
<%-- processing of each item here --%>  
</c:forEach>

The first two attributes are mandatory. a collection is specified in items attribute and each of item in collection in the var attribute. The varStatus attribute is optional. the varStatus attribute is an instance of class which implements interface [LoopTagStatus](http://java.sun.com/products/jsp/jstl/1.1/docs/api/javax/servlet/jsp/jstl/core/LoopTagStatus.html). varStatus provides a set of useful properties to work with such as begin, end, current, index, count.

**Q. What are the possible object reference of items attribute?**

**Ans**: The object referenced by the items variable can be any one of the following data types:

• An array

• An implementation of java.util.Collection

• An implementation of java.util.Iterator

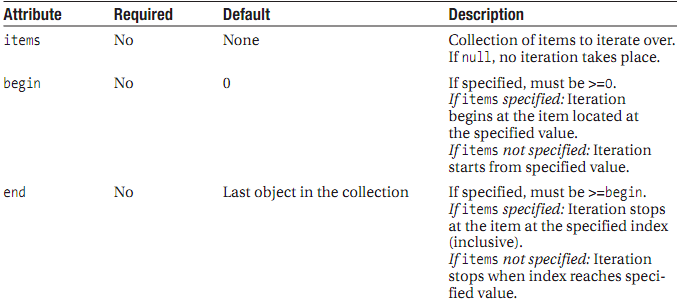
• An implementation of java.util.Enumeration

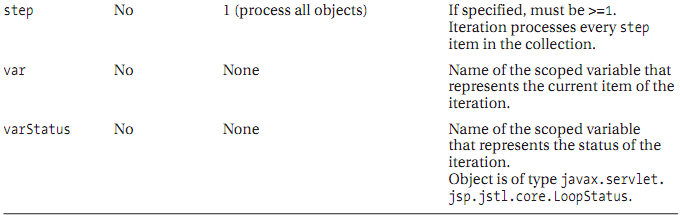
• An implementation of java.util.Map

• A string of comma-separated values

**Q.Explain the Attributes for the <c:forEach> Tag.**

**Ans**:





**Q. Explain the <c:forTokens> Action.**

**Ans**: The second iterating action provided by the JSTL is <c:forTokens>, which iterates over a string of tokens separated by a set of delimiters, much in the same way as the functionality provided by the java.util.StringTokenizer class that you may be familiar with. The syntax for the action is as follows:

<c:forTokens items="stringOfTokens" delims="delimiters"

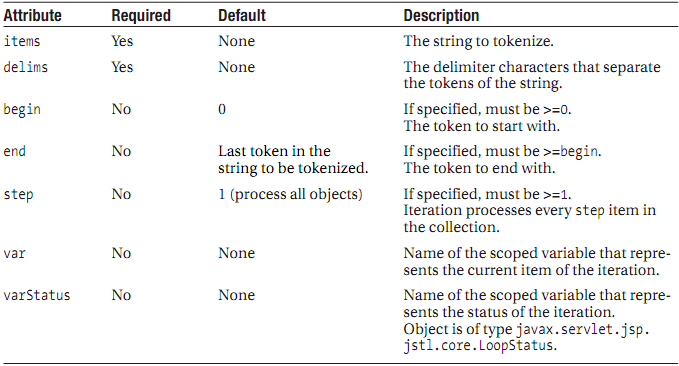
[var="varName"] [varStatus="varStatusName"]

[begin="begin"] [end="end"] [step="step"]>

body content

</c:forTokens>

Attributes for the <c:forTokens> Tag are as follows:



**Q. Explain URL-Related Actions**

**Ans**: In web application development, web developers usually have to deal with URL manipulation such as importing URL, redirect to another URL. JSTL provides several useful URL-related actions to simplify URL manipulation.

**The <c:import> Action:** This action imports the content of a URL-based resource and provides a simple, generic way to access URL-based resources that can be either included or processed within the JSP page. The syntax of the <c:import> action is as follows:

<c:importurl="url"  
         [context="context "]   
         [var="varName"  
         [scope="{page|request|session|application}"]   
         [charEncoding="charEncoding"]>  
<%-- optional body content for<c:param> subtags --%>  
</c:import>

The only attribute url is mandatory. The URL could be in absolute or relative form. If it is in relative form, the resource must be refered inside the web application.

**The <c:url> Action:** The <c:url> action provides a handy way of constructing correctly formatted URLs that have the correct URL-rewriting rules applied. i.e., The <c:url> action enables to format an URL correctly.

**The <c:redirect> Action:**  The <c:redirect> simply to redirect the current page to another page or URL. The syntax of the <c:redirect> is as follows:

<c:redirecturl="newurlhere"/>

## The <c:param> Action: The <c:param> action is used to define a parameter. It is used inside the body of other actions such as <c:import>. <c:url> and <c:redirect> actions. the syntax of the <c:param> action is simple as follows:

<c:param name="paramName"value="value"/>

## The parameter name is specified in the name attribute and its value in the value attribute.

**Q. How to control the locale setting of JSP pages?**

**Ans:** The Internationalization and Formatting tag library provides actions that allow to control the locale settings for JSP pages.

**The <fmt:setLocale> Action :** This action can be used to override the client-specified locale for the processing of a JSP page. Any I18n formatting actions such as <fmt:message> that are found on the page will use this specified locale instead of the one sent by the client browser. The chosen locale is stored in a variable called javax.servlet.jsp.jstl.fmt.locale and can be stored in any chosen scope.

This JSP code first sets the default locale for the page followed by the session:

<fmt:setLocale value ="en\_US" />

<fmt:setLocale value ="fr\_FR" scope="session" />

The value attribute accepts either a string representing the locale (a two-letter, lowercase language code followed a two-letter, optional uppercase country code), or a reference to a java.util.Locale object.

it’s also possible to set a default locale for use via the JSTL by using the following configuration setting in the web application’s deployment descriptor (web.xml):

<context-param>

<param-name>javax.servlet.jsp.jstl.fmt.locale</param-name>

<param-value>en</param-value>

</context-param>

This configuration uses the locale code for English to set the default locale for the application.

**Q. Explain <fmt:bundle> , <fmt:setBundle> and <fmt:message> Actions.**

**Ans**: To enable the use of localized messages, it is necessary to specify the required resource bundle that

provides the localized messages.

**<fmt:bundle> :**The <fmt:bundle> action is used to declare an I18n localization context for use by I18n-aware tags within its body content:

<fmt:bundle basename="Labels">

<fmt:message key="labels.firstName"/>

<fmt:message key="labels.lastName"/>

</fmt:bundle>

Here, a resource bundle with the name Labels is declared to provide the localized resources for any nested <fmt:message> actions. The resource bundle contains at least two name-value pairs given by the keys labels.firstName and labels.lastName. Because the <fmt:bundle> action is designed to work so closely with nested <fmt:message> actions, a handy optional attribute can also be used as follows:

<fmt:bundle basename="Labels" prefix="labels">

<fmt:message key="firstName"/>

<fmt:message key="lastName"/>

</fmt:bundle>

The optional prefix attribute enables the setting of a predefined prefix that is pretended to the key attribute of any nested <fmt:message> actions, which makes their use so much simpler.

**<fmt:setBundle>:**the <fmt:setBundle> action enables a resource bundle to be stored in the configuration variable javax.servlet.jsp.jstl.fmt.localizationContext, so any <fmt:message> actions that appear elsewhere in the JSP page can access the bundle without having to continually declare it as follows:

<fmt:setBundle basename="Labels" />

<fmt:message prefix="labels.firstName" />

**<fmt:message>** : Localized messages are retrieved from a resource bundle by using the <fmt:message> action, which uses a key parameter to extract the message from the resource bundle and print it to the current JspWriter.

**Q. What are the formatting actions?**

**Ans**: In addition to language, users from different locales have different standards regarding the following:

• Date and time formats:

• Number formats:

• Currency formats

• Colors

• Page layouts

• Address standards (zip codes)

**Q. Explain <fmt:timeZone>,<fmt:setTimeZone>,<fmt:formatDate> and <fmt:parseDate>.**

**Ans**: The <fmt:timeZone> action is used to specify a time zone for any nested I18n-aware actions that appear inside its body content, whereas <fmt:setTimeZone> can be used to store a reference to a time zone in an exportable variable for use anywhere on a JSP page. The <fmt:timeZone> action is used as follows:

<fmt:timeZone value="GMT">

//...date/time actions go here

</fmt:timeZone>

A single attribute called value is used to specify the time zone, which can either be a java.util.TimeZone object or a string that represents one of the time zone IDs supported by the Java platform (such as “America/Los Angeles” or a custom time zone such as “GMT-8”). The <fmt:setTimeZone> action is used as follows:

<fmt:setTimeZone value="GMT" var="myTimeZone" scope="request" />

This action enables a java.util.TimeZone object to be stored in a scoped variable that can be utilized by any I18n-aware actions such as the <fmt:formatDate> and <fmt:parseDate> actions

The two I18n-aware date actions provided by the JSTL are <fmt:formatDate> and <fmt:parseDate>. The <fmt:formatDate> action provides flexible, time zone–aware formatting of java.util.Date objects so that the date and time may be displayed correctly depending on the client’s time zone. In its simplest form, the <fmt:formatDate> action applies the default formats of the current time zone and outputs them to the current JspWriter as follows:

<jsp:useBean id="now" class="java.util.Date /">

<fmt:formatDate value="${now}"/>

As mentioned, the <fmt:formatDate> action is highly flexible and provides the ability to display dates and times in predefined or custom formats by using the conventions as set out by the java.text.DateFormat class. The ability to store the formatted date in a scoped string variable is also provided.

The <fmt:parseDate> action provides complementary functionality to the <fmt:formatDate> action by parsing and converting the string representation of dates and times that were formatted in a locale-sensitive or customized manner into java.util.Date objects. This action is particularly useful if you need to enable clients from around the world to enter date-and-time information in their own local format and have it correctly parsed into the correct object at the server.

<fmt:parseDate type="date" pattern="dd/MM/yyyy" var="parsedDate">

22/12/2005

</fmt:parseDate>

**Q. Explain <fmt:formatNumber> and <fmt:parseNumber>.**

**Ans**: The <fmt:formatNumber> action is also flexible and capable of formatting a numeric value in a locale-sensitive or custom format as a number, currency, or percentage. For example, the following action ensures that the given number has at least three decimal places:

<fmt:formatNumber value="123.4" type="NUMBER" minFractionDigits="3" />

The <fmt:parseNumber> action is the reverse formatting tag, used to convert a formatted string representing either a number, currency, or percentage into an appropriate java.lang.Number.

**Q. Explain actions of SQL Tag Library.**

**Ans: <sql:setDataSource>** : The primary job of the <sql:setDataSource> action is to configure a data source that represents an physical data store and expose it as either a scoped variable or the data source configuration object javax.servlet.jsp.jstl.sql.DataSource. The configured data source is used by the remaining actions in the SQL library to source database connections so they may perform queries and updates, and so on. A data source can be configured as follows:

<sql:setDataSource var="dataSource" driver="org.acme.sql.driver"

url="jdbc:msql://localhost/tempDB" user="Dan" password="pwd"/>

The var attribute sets a label by which the data source can be accessed. The driver attribute is the fully qualified class name of the JDBC driver class that is used to communicate with the data source. The url attribute is the JDBC URL used to connect to the data source. The user and password attributes are the username and password used to log in to the data source.

**<sql:query>** : Simple database query functionality is provided by the <sql:query> action:

<sql:query var="users" dataSource="${dataSource}" >

SELECT \* FROM User WHERE UserName='Dan'

</sql:query>

The dataSource attribute is used to reference a DataSource that was configured by using the sql:setDataSource> action. The mandatory var parameter is used to store the result of the query and is of type javax.servlet.jsp.

jstl.sql.Result. It’s possible to pass the string of SQL as either body content or by using the sql attribute.

**<sql:update>** : This action enables SQL Data Manipulation Language INSERT, UPDATE, and DELETE statements to be executed. It is also used to execute SQL Data Definition Language statements, such as a table creation or alteration statements. The syntax of this action is similar to that of the <sql:query> action. Again, a var attribute is available to store the result of the <sql:update> action except this time it’s not mandatory. The type of the var parameter is java.lang.Integer.

<sql:update var="count" dataSource="myDataSource">

DELETE FROM Users WHERE UserName <> 'Dan'

</sql:update>

**<sql:param> and <sql:dateParam>** : These actions are used to pass parameters into a parameterized string of SQL. Both actions are incredibly simple and exist only to hold a simple object via its value attribute:

<sql:param value="${userName}"/>

To see how these tags could be used as part of a parameterized SQL statement, lets consider the example:

<sql:update var="count" dataSource="myDataSource">

DELETE FROM Users WHERE UserName <> ? AND Status = ?

<sql:param value="${userName}"/>

<sql:param value="${status}"/>

</sql:update>

**<sql:transaction>:** This action enables a series of SQL actions to be grouped together to provide transactional behavior. Transactions enable a series of database actions (such as queries, insertions, deletions, and so on) to be treated as a single atomic action. The transaction is committed permanently to the database only when all the database actions within it complete successfully; otherwise, the transaction is rolled back and any actions are reversed. Any <sql:query> or <sql:update> actions that wish to be included as part of the transaction are nested inside the <sql:transaction> action itself. For example,

<sql:transaction dataSource="myDataSource" isolation="read\_committed">

<sql:query var="users">

SELECT \* FROM User WHERE UserName='Dan'

</sql:query>

<sql:update var="count">

DELETE FROM Users WHERE UserName <> ? AND Status = ?

<sql:param value="${userName}"/>

<sql:param value="${status}"/>

</sql:update>

</sql:transaction>

**Q. What are the functional areas concerned in the XML processing tag library?**

**Ans**: The XML processing tag library can be split into the following functional areas concerned

with XML data:

• XML core actions

• XML flow control actions

• XML transformation actions

**Q.Explain XML Core Actions.**

**Ans**: XML Core Actions are as follows:

* <x:parse>: The <x:parse> action simply parses a named XML document and saves it inside a scoped variable for use by other tags from the XML tag library.
* <x:out> : The <x:out> action evaluates an XPath expression (reference to somewhere in the XML
* document) and outputs the result of the evaluation to the current JspWriter.
* <x:set>: The <x:set> actions simply store the values inside scoped variables.

For example:

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>

<c:import url="book.xml" var="url" />

<x:parse xml="${url}" var="book" scope="session" />

<x:set select="$book/book/title" var="title" scope="session"/>

<x:set select="$book/book/author" var="author" scope="session"/>

<x:set select="$book/book/url" var="bookUrl" scope="session"/>

<b><x:out select="$title" /></b><br>

<x:out select="$author" /><br>

<x:out select="$bookUrl" /><br>

**Q. Explain XML Flow Control Actions.**

**Ans**: XML Flow Control Actions are as follows:

* <x:if> :The <x:if> action processes its body content only if the Boolean condition evaluates to true.
* <x:choose>, <x:when>, and <x:otherwise>
* <x:forEach>

**Q. Explain XML Transformation Actions.**

**Ans**: The XML standard tag library also includes tags that allow you to transform XML data into other formats such as HTML or WML.

* <x:transform>
* <x:param>

**Chapter-5**

**Q. How many ways JSF helps web application developers?**

**Ans**: The JSF specification lists the following ways that JSF helps web-application developers to create user interfaces (UIs):

* Makes it easy to construct a UI from a set of reusable UI components
* Simplifies migration of application data to and from the UI
* Helps manage UI state across server requests
* Provides a simple model for wiring client-generated events to server-side application code
* Allows custom UI components to be easily built and reused

**Q. What is JSF?**

**Ans**: JSF is a supporting technology and it can be used in conjunction with JSP pages, servlets, or other presentation technologies. The primary design pattern of JSF is the Model-View-Controller (MVC) pattern. JSF brings a component-based model to web- application development that is similar to the model used in stand-alone GUI applications. The JSF implementation includes a tag library of custom tags to easily create JSF-enabled applications.

**Q. Explain MVC in terms of Web Application.**

**Ans**: MVC separates a web application architecture into three categories of components: model, view, and controller. The model is the abstraction of all the domain data in the system. The view is the visualization of the model. In a web application, the view consists of the HTML pages and the components that create the HTML pages sent to web browsers, the Wireless Application Protocol (WAP) pages sent to mobile devices, or the UI components sent to a dedicated client. The controller is the set of components that manage the communications between model and view.

**Q. What are the main requests or responses we most concerned about in a JSF-enabled application?**

**Ans**: Several kinds of request/response cycles can occur in a JSF-enabled application. We are concerned with these three request/response pairs:

* Non-JSF request generates JSF response
* JSF request generates JSF response
* JSF request generates non-JSF response

**Q. Explain JSF life cycle.**

**Ans**: In JSF, the model is composed of business objects that are usually implemented as JavaBeans, the controller is the JSF implementation, and the UI components are the view. The JSF life cycle has six phases as defined by the JSF specification:

• **Restore View:** In this phase, the JSF implementation restores the objects and data structures that represent the view of the request.

• **Apply Request Values:** Any data that was sent as part of the request is passed to the appropriate UI objects that compose the view. Those objects update their state with the data values. It updates only the UI components with the new data.

**• Process Validations:** The data that was submitted with the form is validated (if it was not validated in the previous phase). This is because if the JSF implementation began to update the business objects as data was validated, and a piece of data failed validation, the model would be partially updated and in an invalid state.

**• Update Model Values:** After all validations are complete, the business objects that make up the application are updated with the validated data from the request. In addition, if any of the data needs to be converted to a different format to update the model the conversion occurs in this phase. Conversion is needed when the data type of a property is not a String or a Java primitive.

**• Invoke Application:** During this phase, the action method of any command button or link that was activated is called. In addition, any events that were generated during previous phases and that have not yet been handled are passed to the web application so that it can complete any other processing of the request that is required.

**• Render Response:** The response UI components are rendered, and the response is sent to the client. The state of the UI components is saved so that the component tree can be restored when the client sends another request.

**Q. Write down the name of the jar files needed for installation of the JSF.**

**Ans**: Following eight JAR files are located in the lib directory of each distribution, into a location that can be accessed by the server or the web application:

• Six JSF JARs:

1. commons-beanutils.jar,
2. commons-collections.jar,
3. commons-digester.jar,
4. commons-logging.jar,
5. jsf-api.jar and
6. jsf-impl.jar

• Two JSTL JARs:

1. jstl.jar and
2. standard.jar

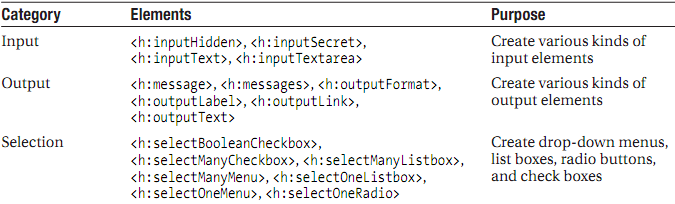
**Q. Which are the two libraries of custom actions that can be used with JSP pages?**

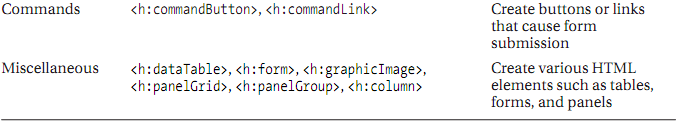
**Ans**: The JSF implementation from Sun comes with two libraries of custom actions that can be used with JSP pages:

* **HTML custom actions:** These are for components that vary based on the render kit used. These custom actions are used to create HTML elements. The HTML custom actions fall into five categories: input, output, selection, commands, and miscellaneous.
* **Core custom actions**: These create UI elements that are independent of the render kit. These actions are usually used with the HTML actions to modify the behavior of those actions.

**Q. Explain the HTML custom actions.**

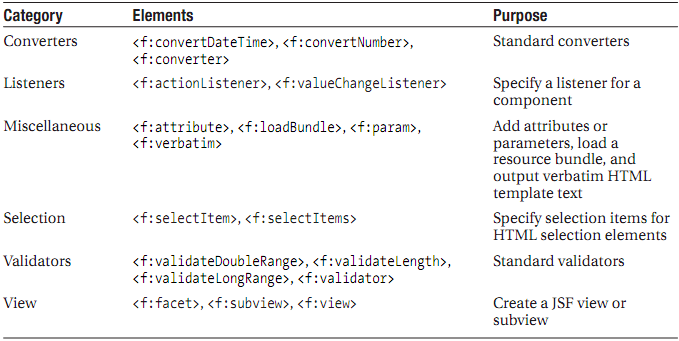
**Ans:**





**Q. Explain the Core custom actions.**

**Ans**:



**Q. What is faces-config.xml?**

**Ans**: Information about the view components in the web application and information about how control flows through the application is contained in a special configuration file named faces-config.xml.

The faces-config.xml file identifies the JavaBeans used by the web application in the <managed-bean> element. It is necessary to have a <managed-bean> element for every JavaBean used by the web application. The faces-config.xml file is also used to tell the controller how to navigate through the application.

**Q. What are the sub elements of <managed-bean> element?**

**Ans**: The <managed-bean> element contains three subelements:

* **<managed-bean-name>:** The first subelement is the name used to identify the JavaBean in a JSP page.
* **<managed-bean-class>:** The second element is the fully qualified class name of the JavaBean class. This name tells the JSP container which class to load and instantiate to create an instance of the JavaBean.
* **<managed-bean-scope>:** The third element identifies the scope of the object.

The <managed-bean> element has a number of optional elements, including <description>,<display-name>, <icon>, and <managed-property>.

**Q. Explain <managed-property>.**

**Ans**: <managed-property> can have an optional <description>, <display-name>, and <icon>. It must have a nested <property-name> element that identifies the name of an instance variable (property) of the class with a set and getmethod. It can have an optional <property-class> element that provides the fully qualified class name of the data type of the property. If the data type is not provided, the JSF implementation will attempt to infer the type from the bean class. Finally, it must have one of several elements that initialize the value of the property: **<value>:** I f the property of the bean is a Java primitive or a String

**<null-value>:** If the type of the property is a Java object and not a primitive

**<list-entries>** or **<map-entries>:** If the type of the property is List or a subtype of List, or Map or a subtype of Map.

**Q. Explain <navigation-rule> elements.**

**Ans:** Navigation flow is specified in <navigation-rule> elements. In general, a <navigation-rule> element identifies the start page, a condition, and the page to navigate to when the condition occurs. The general syntax of navigation rules is as follows:

**<navigation-rule>**

**<from-view-id>/abc.jsp</from-view-id>**

**<navigation-case>**

**<from-outcome>word</from-outcome>**

**<to-view-id>/xyz.jsp</to-view-id>**

**</navigation-case>**

**</navigation-rule>**

The <from-view-id> element is optional; it contains the path to the page from which navigation starts. The <from-outcome> element is the string value that is compared to the value of the action attribute.It is necessary to specify the <from-view-id> only one time. If there is an action that applies to every page in the application, then <navigation-rule> element can be used without a <from-view-id>. There are a number of optional elements used with <navigation-case>. Some of them are <description>, <display-name>, <icon>, <from-outcome>, <from-action>, and <redirect>.

**Q. How does a JSF application is deployed?**

**Ans**: The deployment descriptor identifies the controller servlet (Faces Servlet) for the application, specifies a servlet mapping indicating which requests should be sent to the controller servlet, and designates the welcome file for the application. It is necessary to copy the JSF and JSTL JAR files into the application’s WEB-INF\lib directory or into the Tomcat common\lib directory. After then copying the entire application directory into the Tomcat webapps directory, and deployment is complete.

Alternatively, after creating the directory structure, all the application files are packaged into a WAR file by using the Java jar command. After the WAR file has been created, copying it into the Tomcat webapps directory. Tomcat will automatically unpack the WAR file and start the application.

**Q. Write two aspects of JavaBeans.**

**Ans**: There are two aspects of the JavaBeans specification that are important:

• The JavaBean used in the web application must have a no-argument constructor. This allows the container to construct an instance of the JavaBean.

• Any property to be exposed must have a get or set method. If only a get method is present, the property is read-only. If only a set method is used, the property is write only. If both are present, the property is read-write. The format of the set method name is the word set followed by the name of the property, with the first letter of the property name capitalized. The get method format is the word get followed by the name of the property, again with the first letter of the property name capitalized. For Boolean properties, the method is the word is followed by the name of the property.

**Q. What do you understand by managed beans?**

**OR, How to initialize JavaBean properties?**

**Ans**: Within the JSF implementation, JavaBean instances that are used by a JSF-enabled application are referred to as managed beans, because the JSF implementation manages the creation and use of JavaBean objects. Within a JSF-enabled application, managed beans appear in two contexts:

• The information needed to create and initialize the managed bean is identified within the configuration files of the application.

• The properties and methods of managed beans are referenced in JSP pages by using value-binding expressions or method-binding expressions.

**Q. What are the scopes of JavaBean?**

**Ans**: JavaBean used in a JSF page can be configured with one of four scopes:

* **None:** Objects with this scope are not visible in any JSF page. When used in the configuration file, they indicate managed beans that are used by other managed beans in the application. Objects with none scope can use other objects with none scope.
* **Request**: Objects with this scope are visible from the start of the request until the end of the request. Request scope starts at the beginning of a request and ends when the response has been sent to the client. If the request is forwarded, the objects are visible in the forwarded page, because that page is still part of the same request/response cycle. Objects with request scope can use other objects with none, request, session, or application scope.
* **Session**: An object with session scope is visible for any request/response cycle that belongs to a session. Objects with this scope have their state persisted between requests and last until the object or the session is invalidated. Objects with session scope can use other objects with none, session, or application scope.
* **Application**: An object with application scope is visible in all request/response cycles for all clients using the application, for as long as the application is active. Objects with application scope can use other objects with none or application scope.

**Q. What is static navigation and dynamic navigation?**

**Ans**: **Static navigation:** To control the navigation through string values of the action attribute, the path of navigation is known when the application is deployed. This is known as static navigation, because the flow is statically determined and does not change. It is the same for every request. When using static navigation, it is required to explicitly code a value into the action attribute of a JSF custom tag and then define navigation rules in a configuration file.

**Dynamic navigation:** To control the navigation through value-binding expressions or method-binding expressions, the path of navigation is not known when the application is deployed. In fact, navigation flow can vary from request to request depending on the value of the expression. This is known as dynamic navigation.

**Q. Explain the standard Converters.**

**Ans**: The JSF implementation comes with two standard converters, one for numbers and one for dates and times:

* **<convertNumber>:** Converts strings to numbers, and vice versa. It is possible to include optional attributes to format the numbers in various ways including as currency, as integers, and as floating-point numbers.
* **<convertDateTime>:** Converts strings to dates or times, and vice versa. It is possible to include optional attributes to format by using various styles and time zones.

**Q. How to create custom converter?**

**Ans**: To create a custom converter, it is necessary to write a class that implements the javax.faces.convert. Converter interface. This interface has two methods:

* Object getAsObject(javax.faces.context.FacesContext context, javax.faces.component.UIComponent component, java.lang.String value)
* String getAsString(javax.faces.context.FacesContext context, javax.faces.component.UIComponent component, java.lang.Object value)

The **getAsObject()**method converts the String value (which can be null) to an instance of the supported type and returns the new instance. This method throws a ConverterException if the conversion fails.

The **getAsString()**method converts the provided value of the supported type (which can again be null) to a String instance and returns the new instance. This method also throws a ConverterException if the conversion fails.

**Q. Explain the Standard Validators provided in the JSF.**

**Ans**: JSF provides three standard validators as part of the JSF implementation through the following custom tags:

**• <validateDoubleRange>:** Validates that a value is a double. You can include the optional attributes minimum and maximum to set minimum and maximum values.

**• <validateLongRange>:** Validates that a value is a long. You can include the optional attributes minimum and maximum to set minimum and maximum values.

**• <validateLength>:** Validates a string value for length. You can include the optional attributes minimum and maximum to set minimum and maximum values.

With all three validators, both of the attributes are optional. This means it is possible to add a validator tag without either a minimum or maximum attribute. In that case, no validation will be performed.

**Q. Which are the required subelements of <validator> element?**

**Ans**: The validator is registered with the JSF implementation with the <validator> element in a configuration file. The <validator> element has two required subelements:

**• <validator-id>:** Used to create an ID string that can be used in the <validator> tag to specify a validator instance. This ID must be unique within the application.

**• <validator-class>:** The fully qualified class name of the validator class.

**Q. Explain ActionListener in terms of JSF.**

**Ans**: Action listeners are provided by JSF to make it easier to handle action events. Action listeners are attached to the two JSF command elements: command buttons and command links. Action events are handled in a manner very similar to value change events. A listener can be attached to a command element with the actionListener attribute. Value change listeners are attached to input elements such as text fields, radio buttons and check boxes, and menus.