# Advance Java Programming (Assignment)

## Q- A) Explain life cycle of JSP.

- JSP stands for Java Server Pages.
  - JSP is Server-side scripting language: **Server-side scripting** means that the JSP code is processed on the web server rather than the client machine.
  - JSP page is a file with a.JSP extension that contains could be the combination of HTML Tags and JSP codes.
  - JSP is used to build dynamic web applications.

- JSP Life Cycle is defined as translation of JSP Page into servlet as a JSP Page needs to be converted into servlet first in order to process the service requests. The Life Cycle starts with the creation of JSP and ends with the disintegration of that.

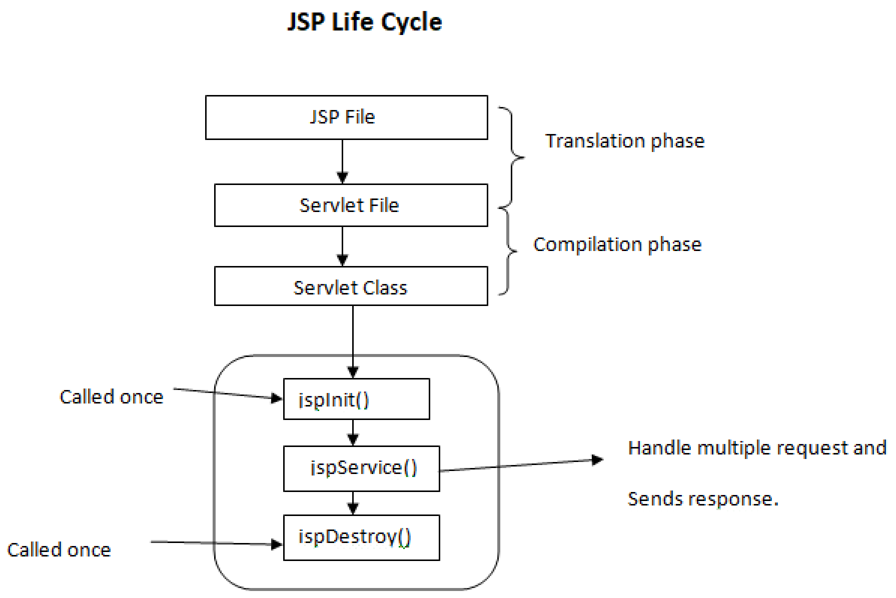


**Fig.01 -> JSP LIFE CYCLE**

- When the browser asks for a JSP, JSP engine first checks whether it needs to compile the page. If the JSP is last compiled or the recent modification is done in JSP, then the JSP engine compiles the page.

Compilation process of JSP page involves three steps:

- Parsing of JSP
- Turning JSP into servlet
- Compiling the servlet

JSP Lifecycle is depicted in the below diagram by example point of view:

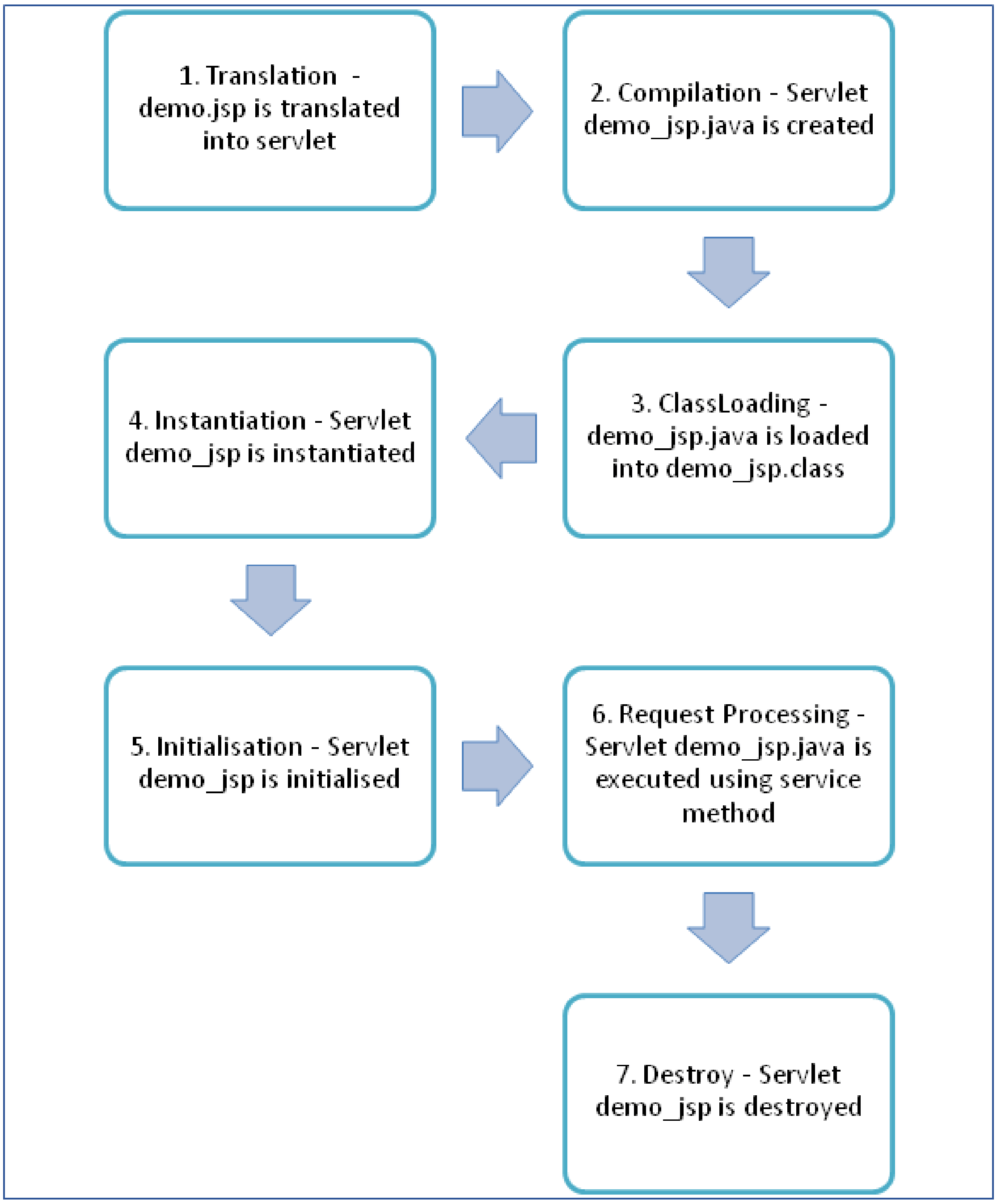

1. Translation - demo.jsp is translated into servlet
2. Compilation - Servlet demo_jsp.java is created
3. ClassLoading - demo_jsp.java is loaded into demo_jsp.class
4. Instantiation - Servlet demo_jsp is instantiated
5. Initialisation - Servlet demo_jsp is initialised
6. Request Processing - Servlet demo_jsp.java is executed using service method
7. Destroy - Servlet demo_jsp is destroyed

**Fig.02 -> Diagram for Steps in JSP cycles(As example view)**

Following steps explain the JSP life cycle:

1. **Translation of JSP page**
2. **Compilation of JSP page (Compilation of JSP page into _jsp.java)**
3. **Class loading (_jsp.java is converted to class file_jsp.class)**
4. **Instantiation (Object of generated servlet is created)**
5. **Initialisation (_jspinit() method is invoked by container)**
6. **Request Processing(_jspservice() method is invoked by the container)**
7. **Destroy (_jspDestroy() method invoked by the container)**

Now Let's began to understand step by step:

## 1 -> Translation of the JSP Page:

A Java servlet file is generated from a JSP source file. This is the first step of JSP life cycle. In translation phase, container validates the syntactic correctness of JSP page and tag files.

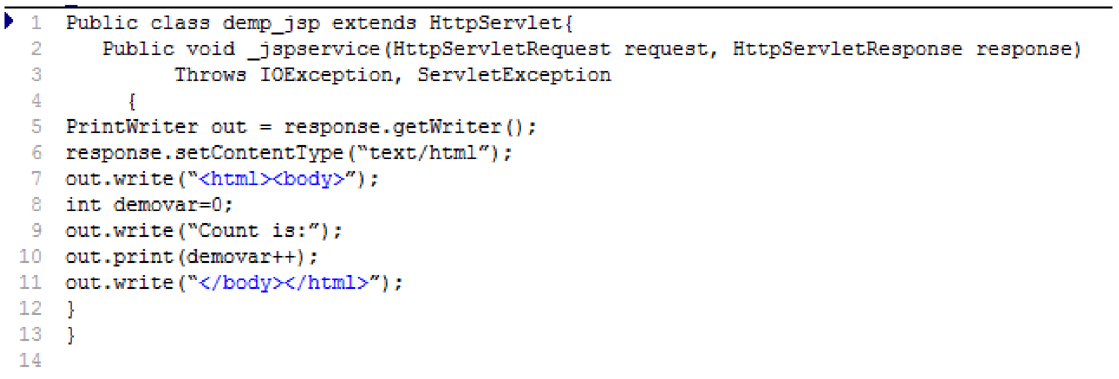
- The JSP container interprets the standard directives and actions, and the custom actions referencing tag libraries (they are all part of JSP page and will be discussed in the later section) used in this JSP page.
- In the above pictorial description, demo.jsp is translated to demo_jsp.java in the first step
- Let's take an example of "demo.jsp" as shown below:

**Demo.jsp**

```
<html>

<head>

<title> DEMO JSP </title>

</head>

<%

Int demvar=0;%>

<body>
```

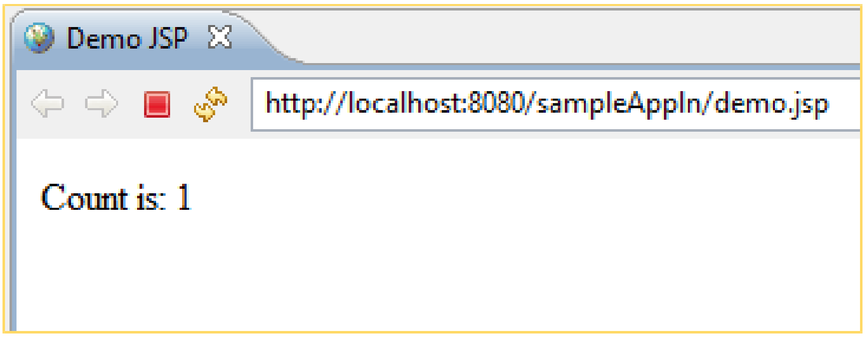
```
Count is:

<% Out.println(demovar++); %>

</body>

</html>
```

Demo JSP Page is converted into demo_jsp servlet in the below code.



```
Public class demp_jsp extends HttpServlet{
   Public void _jspservice(HttpServletRequest request, HttpServletResponse response)
         Throws IOException, ServletException
     {
PrintWriter out = response.getWriter();
response.setContentType("text/html");
out.write("<html><body>");
int demovar=0;
out.write("Count is:");
out.print(demovar++);
out.write("</body></html>");
}
}
```

Output:



Count is: 1

## 2 -> Compilation of the JSP Page

- The generated java servlet file is compiled into java servlet class
- The translation of java source page to its implementation class can happen at any time between the deployment of JSP page into the container and processing of the JSP page.
- In the above pictorial description demo_jsp.java is compiled to a class file demo_jsp.class

## 3 -> Class loading

- Servlet class that has been loaded from JSP source is now loaded into the container

## 4 -> Instantiation

- In this step the object i.e. the instance of the class is generated.
- The container manages one or more instances of this class in the response to requests and other events. Typically, a JSP container is built using a servlet container. A JSP container is an extension of servlet container as both the container support JSP and servlet.
- A JSPPage interface which is provided by container provides init() and destroy () methods.
- There is an interface HttpJSPPage which serves HTTP requests, and it also contains the service method.

## 5 -> Initialization

```
public void jspInit()
{
  //initializing the code
}
```

- _jspinit() method will initiate the servlet instance which was generated from JSP and will be invoked by the container in this phase.
- Once the instance gets created, init method will be invoked immediately after that
- It is only called once during a JSP life cycle, the method for initialization is declared as shown above

## 6 -> Request processing

```
void _jspservice(HttpServletRequest request HttpServletResponse res
onse){          //handling all request and responses
}
```

- _jspservice() method is invoked by the container for all the requests raised by the JSP page during its life cycle
- For this phase, it has to go through all the above phases and then only service method can be invoked.
- It passes request and response objects

- This method cannot be overridden
- The method is shown above: It is responsible for generating of all HTTP methods exm =>GET, POST, etc.

## 7 -> Destroy

```
public void _jspdestroy()
{
        //all clean up code
}
```

- _jspdestroy() method is also invoked by the container
- This method is called when container decides it no longer needs the servlet instance to service requests.
- When the call to destroy method is made then, the servlet is ready for a garbage collection
- This is the end of the life cycle.
- We can override jspdestroy() method when we perform any cleanup such as releasing database connections or closing open files.

## Q-2) Explain Cookie with example.

### ➢ Cookies in JSP:

- Cookies are defined as the text files it will be stored in the computer mainly in the client machine for tracking purposes like login details, file transfer, etc.
- We can also saw the cookies in web browsers in web applications using debug mode choose "Application ->Storage-. Cookies"
- We can see whatever we see the web applications or websites in the cookie's column.
- In JSP we handline the cookies concept like cookies handling JSP transparently supports for the cookies using HTTP cookies and JSP has some predefined methods for handling the cookies.
- The client-server interactions like request and response methods using like request. get cookies () used to return the array of the cookie instance.

### ➢ There are three steps involved in identifying and returning user's script:

- Server script sends a set of cookies to the browser.
  For example, name, age, or identification number, etc.
- Browser stores this information on the local machine for future use.
- When the next time the browser sends any request to the web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

### ➢ The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this −

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;
  path = /; domain = example.com
Connection: close
```

```
Content-Type: text/html
```

- As you can see, the **Set-Cookie header** contains **a name value pair, a GMT date, a path** and **a domain**. The name and value will be URL encoded. The **expires** field is an instruction to the browser to **"forget"** the cookie after the given time and date.
- If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this −

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126

Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name = xyz
```

- A JSP script will then have access to the cookies through the request method ***request.getCookies()*** which returns an array of *Cookie* objects.

## Setting Cookies with JSP

Setting cookies with JSP involves three steps −

### Step 1: Creating a Cookie object

You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters −

[ ] ( ) = , " / ? @ : ;

## Step 2: Setting the maximum age

You use **setMaxAge** to specify how long (in seconds) the cookie should be valid. The following code will set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

## Step 3: Sending the Cookie into the HTTP response headers

You use **response.addCookie** to add cookies in the HTTP response header as follows

```
response.addCookie(cookie);
```

**Example: Main.jsp**

```
<%
  // Create cookies for first and last names.
  Cookie firstName = new Cookie("first_name",
request.getParameter("first_name"));
  Cookie lastName = new Cookie("last_name",
request.getParameter("last_name"));

  // Set expiry date after 24 Hrs for both the cookies.
  firstName.setMaxAge(60*60*24);
  lastName.setMaxAge(60*60*24);

  // Add both the cookies in the response header.
  response.addCookie( firstName );
  response.addCookie( lastName );
%>

<html>
  <head>
    <title>Setting Cookies</title>
  </head>

  <body>
    <center>
      <h1>Setting Cookies</h1>
    </center>
    <ul>
      <li><p><b>First Name:</b>
```

```
        <%= request.getParameter("first_name")%>
      </p></li>
      <li><p><b>Last  Name:</b>
        <%= request.getParameter("last_name")%>
      </p></li>
    </ul>

  </body>
</html>
```

**Hello.jsp**

```
<html>
  <body>

    <form action = "main.jsp" method = "GET">
      First Name: <input type = "text" name = "first_name">
      <br />
      Last Name: <input type = "text" name = "last_name" />
      <input type = "submit" value = "Submit" />
    </form>

  </body>
</html>
```

**Output:**

```
First Name: 
Last Name: 
```

- To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies( )** method of *HttpServletRequest*. Then cycle through the array, and use **getName()** and **getValue()** methods to access each cookie and associated value.

**Example: Main.jsp**

```
<html>
  <head>
```

```
    <title>Reading Cookies</title>
  </head>

  <body>
    <center>
      <h1>Reading Cookies</h1>
    </center>
    <%
      Cookie cookie = null;
      Cookie[] cookies = null;

      // Get an array of Cookies associated with the this domain
      cookies = request.getCookies();

      if( cookies != null ) {
        out.println("<h2> Found Cookies Name and Value</h2>");

        for (int i = 0; i < cookies.length; i++) {
          cookie = cookies[i];
          out.print("Name : " + cookie.getName( ) + ",  ");
          out.print("Value: " + cookie.getValue( )+" <br/>");
        }
      } else {
        out.println("<h2>No cookies founds</h2>");
      }
    %>
  </body>

</html>
```

**Output:**

```
Name : first_name, Value: John

Name : last_name,  Value: Player
```

## Delete Cookies with JSP

To delete cookies is very simple. If you want to delete a cookie, then you simply need to follow these three steps −

- Read an already existing cookie and store it in Cookie object.

- Set cookie age as zero using the **setMaxAge()** method to delete an existing cookie.
- Add this cookie back into the response header.

Example: Main.jsp

```
<html>
  <head>
    <title>Reading Cookies</title>
  </head>

  <body>
    <center>
      <h1>Reading Cookies</h1>
    </center>
    <%
      Cookie cookie = null;
      Cookie[] cookies = null;

      // Get an array of Cookies associated with the this domain
      cookies = request.getCookies();

      if( cookies != null ) {
        out.println("<h2> Found Cookies Name and Value</h2>");

        for (int i = 0; i < cookies.length; i++) {
          cookie = cookies[i];

          if((cookie.getName( )).compareTo("first_name") == 0 ) {
            cookie.setMaxAge(0);
            response.addCookie(cookie);
            out.print("Deleted cookie: " +
            cookie.getName( ) + "<br/>");
          }
          out.print("Name : " + cookie.getName( ) + ",  ");
          out.print("Value: " + cookie.getValue( )+" <br/>");
        }
      } else {
        out.println(
        "<h2>No cookies founds</h2>");
      }
    %>
  </body>

</html>
```

**Output:**

```
Deleted cookie : first_name

Name : first_name, Value: John

Name : last_name,  Value: Player
```

- Now run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows −

```
Found Cookies Name and Value


Name : last_name,  Value: Player
```

- You can delete your cookies in the Internet Explorer manually. Start at the Tools menu and select the Internet Options. To delete all cookies, click the Delete Cookies button.

## Q-3) Explain JSP Standard Tag Libraries.

- JSP Standard Tag Library (JSTL) is a standard library of readymade tags. The JSTL contains several tags that can remove scriplet code from a JSP page by providing some ready to use, already implemented common functionalities.
- JSTL is divided into 5 groups:

1. **JSTL Core:** JSTL Core provides several core tags such as if, for Each, import, out etc. to support some basic scripting tasks. URL to include JSTL Core Tag inside JSP page is →

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

2. **JSTL Formatting:** JSTL Formatting library provides tags to format text, date, number for Internationalized web sites. URL to include JSTL Formatting Tags inside JSP page is →

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

3. **JSTL sql:** JSTL SQL library provides support for Relational Database Connection and tags to perform operations like insert, delete, update, select etc. on SQL databases. URL to include JSTL SQL Tag inside JSP page is →

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

4. **JSTL XML:** JSTL XML library provides support for XML processing. It provides flow control, transformation features etc. URL to include JSTL XML Tag inside JSP page is →

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

5. **JSTL functions:** JSTL functions library provides support for string manipulation. Url to include JSTL Function Tag inside JSP page is →

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"
%>
```

## JSP JSTL Core Library

- The JSTL core library contains several tags that can be used to eliminate the basic scripting overhead such as for loop, if...else conditions etc from a JSP Page. Let's study some important tags of JSTL Core library.

- **JSTL if tag:** The if tag is a conditional tag used to evaluate conditional expressions. When a body is supplied with if tag, the body is evaluated only when the expression is true.

For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:if test="${param.name == 'studytonight'}">

   <p>Welcome to ${param.name} </p>

  </c:if>

 </body>

</html>
```

- **JSTL out tag:** The out tag is used to evaluate an expression and write the result to JspWriter.

For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:out value="${param.name}" default="StudyTonight" />

 </body>

</html>
```

The value attribute specifies the expression to be written to the JspWriter. The default attribute specifies the value to be written if the expression evaluates null.

- **JSTL forEach tag:** This tag provides a mechanism for iteration within a JSP page. JSTL forEach tag works similarly to enhanced for loop of Java Technology. You can use this tag to iterate over an existing collection of items.

For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:forEach var="message" items="${errorMsgs}" >

   <li>${message}</li>

  </c:forEach>

 </body>

•   </html>
```

Here the attribute items has its value as an EL expression which is a collection of error messages. Each item in the iteration will be stored in a variable called message which will be available in the body of the forEach tag.

- **JSTL choose, when, otherwise tag:** These are conditional tags used to implement conditional operations. If the test condition of
  the when tag evaluates to true, then the content within when tag is evaluated, otherwise the content within the otherwise tag is evaluated.

We can also implement if-else-if construct by using multiple when tag. The when tags are mutually exclusive, that means the first when tag which evaluates to true is evaluated and then, the control exits the choose block. If none of the when condition evaluates to true, then otherwise condition is evaluated.

For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:forEach var="tutorial" items="${MyTutorialMap}" begin="0"
end="5" varStatus="status">

   <c:choose>

    <c:when test="${status.count %2 == 0 }">

     <p> Divisible by 2 : ${tutorial.key} </p>

     <br/>

    </c:when>

    <c:when test="${status.count %5 == 0 }">

     <p > Divisible by 5 : ${tutorial.key} </p>

      <br/>

    </c:when>

    <c:otherwise>
```

```
          <p> Neither divisible by 2 nor 5 : ${tutorial.key} </p><br/>

        </c:otherwise>

      </c:choose>

    </c:forEach>

  </body>

</html>
```

- **JSTL import tag:** < c: import> tag is used to dynamically add the contents from the provided URL to the current page, at request time. The URL resource used in the < c: import> URL attribute can be from outside the web Container.
- For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:import url="http://www.example.com/hello.html">>

  <c:param name="showproducts" value="true"/>

  </c:import>

 </body>

</html>
```

- **JSTL url tag:** The JSTL URL tag is used to store a URL in a variable and also perform url rewriting when necessary.

  For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <a href='<c:url value="/home.jsp"/>' > Go Home </a>

 </body>

</html>
```

- **JSTL set tag:** The JSTL set tag is used to store a variable in specified scope or update the property of JavaBean instance. Following is the example of setting the name property of a Student bean :

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

<c:set target="student" property="name" value="${param.name}" />

 </body></html>
```

- **JSTL catch tag:** The JSTL catch tag is used to handle exception and doesn't forward the page to the error page.

For Example:

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

<html>

 <head>

  <title>Tag Example</title>

 </head>

 <body>

  <c:catch>

   <% int a = 0;

     int b = 10;

     int c = b/a;

   %>

   </c:catch>

 </body>

</html>
```

## Q-4) Explain Hibernate Architecture.

- Hibernate is a framework which is used to develop persistence logic which is independent of Database software. In JDBC to develop persistence logic we deal with primitive types. Whereas Hibernate framework we use Objects to develop persistence logic which are independent of database software.

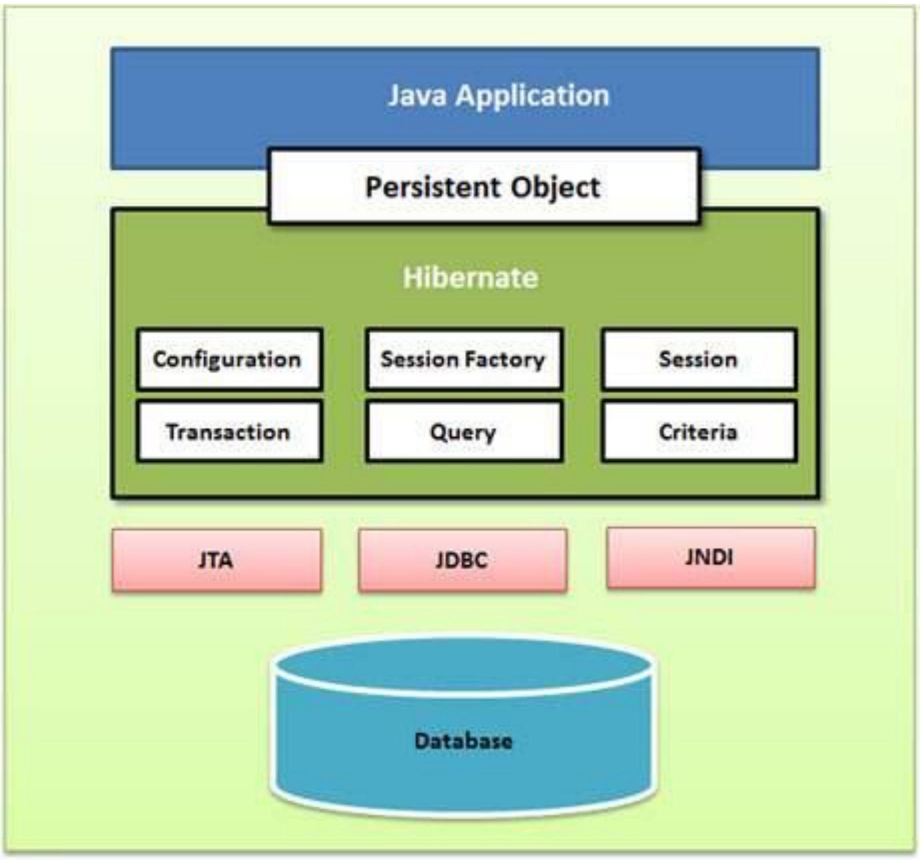


Fig.03 -> Hibernate layout

### Hibernate Architecture:

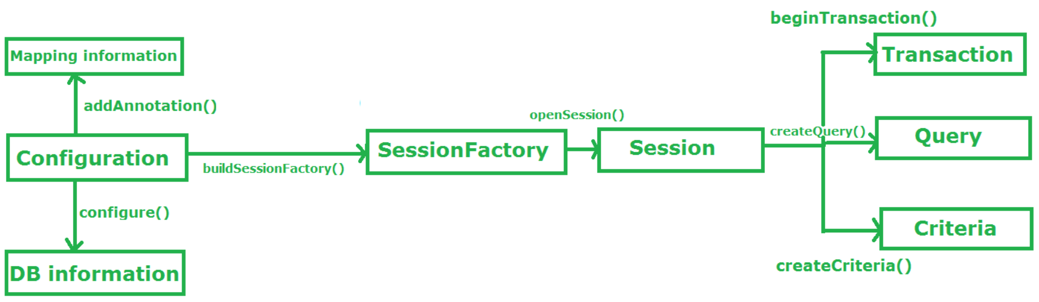


Fig: Hibernate Architecture

- **Configuration:**

  - Configuration is a class which is present in org.hibernate.cfg package. It activates Hibernate framework. It reads both configuration file and mapping files.
  - It activates Hibernate Framework

**Configuration cfg=new Configuration ();**

- It read both cfg file and mapping files

**cfg.configure();**

- It checks whether the config file is syntactically correct or not.
- If the config file is not valid then it will throw an exception. If it is valid then it creates a meta-data in memory and returns the meta-data to object to represent the config file.

- **Session Factory:**

  - SessionFactory is an Interface which is present in org.hibernate package and it is used to create Session Object.
  - It is immutable and thread-safe in nature.
  - buildSessionFactory() method gathers the meta-data which is in the cfg Object.
  - From cfg object it takes the JDBC information and create a JDBC Connection.

**SessionFactory factory=cfg.buildSessionFactory();**

- **Session:**
  - Session is an interface which is present in org.hibernate package. Session object is created based upon SessionFactory object i.e. factory.
  - It opens the Connection/Session with Database software through Hibernate Framework.
  - It is a light-weight object and it is not thread-safe.
  - Session object is used to perform CRUD operations.

**Session session=factory.buildSession();**

## ➢ Transaction:

- Transaction object is used whenever we perform any operation and based upon that operation there is some change in database.
- Transaction object is used to give the instruction to the database to make the changes that happen because of operation as a permanent by using commit() method.

```
Transaction tx=session.beginTransaction();
tx.commit();
```

## ➢ Query:

- Query is an interface that present inside org.hibernate package.
- A Query instance is obtained by calling Session.createQuery().
- This interface exposes some extra functionality beyond that provided by Session.iterate() and Session.find():
    1. A particular page of the result set may be selected by calling setMaxResults(), setFirstResult().
    2. Named query parameters may be used.

```
Query query=session.createQuery();
```

## ➢ Criteria:

- Criteria is a simplified API for retrieving entities by composing Criterion objects.
- The Session is a factory for Criteria. Criterion instances are usually obtained via the factory methods on Restrictions.

```
Criteria criteria=session.createCriteria();
```

## Flow of working during operation in Hibernate Framework:

- Suppose We want to insert an Object to the database. Here Object is nothing but persistence logic which we write on java program and create an object of that program. If we want to insert that object in the database or we want to retrieve the object from the database. Now the question is that how hibernate save the Object to the database or retrieve the object from the database. There are several layers through which Hibernate framework go to achieve the above task. Let us understand the layers/flow of Hibernate framework during performing operations:

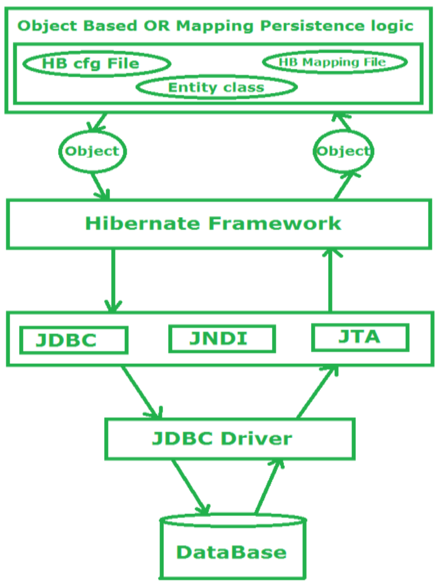


Fig: Working Flow of Hibernate framework to save/retrieve the data from the database in form of Object

**Stage I:** In first stage, we will write the persistence logic to perform some specific operations to the database with the help of Hibernate Configuration file and Hibernate mapping file. And after that we create an object of the particular class on which we wrote the persistence logic.

**Stage II:** In second stage, our class which contains the persistence logic will interact with the hibernate framework where hibernate framework gives some abstractions do perform some task. Now here the picture of java class is over. Now Hibernate is responsible to perform the persistence logic with the help of layers which is below of Hibernate framework or we can say that the layers which are the internal implementation of Hibernate.

**Stage III:** In third stage, our hibernate framework interact which JDBC, JNDI, JTA etc to go to the database to perform that persistence logic.
**Stage IV & V:** In fourth & fifth stage, hibernate is interact with Database with the help of JDBC driver. Now here hibernate perform that persistence logic which is nothing but **CRUD** operation. If our persistence logic is to retrieve a record then in the reverse order it will display on the console of our java program in terms of Object.

## Q-5) Explain Spring Architecture.

- Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.
- The Spring Framework provides about 20 modules which can be used based on an application requirement.
-

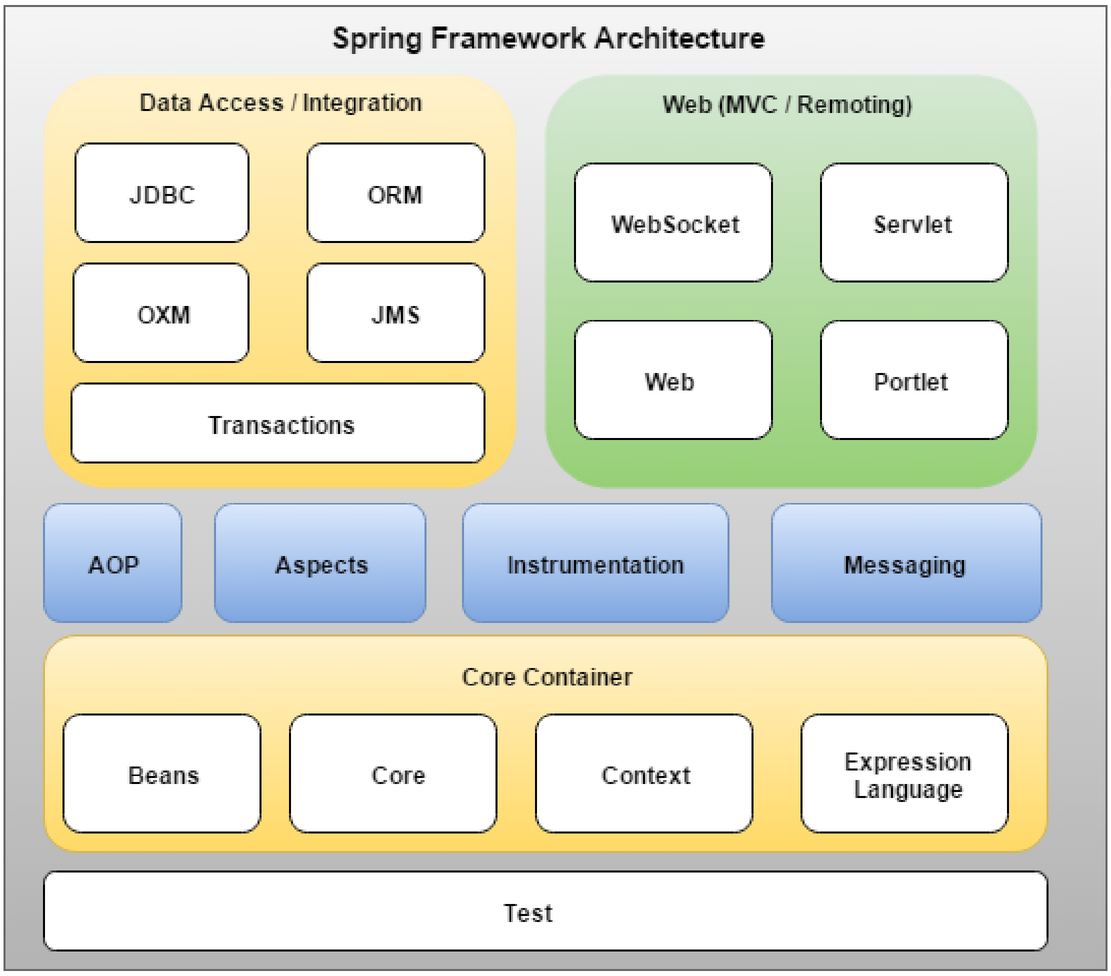


**Fig.04 -> Spring Architecture**

### 1) Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.

- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

## 2) Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

## 3) Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.

- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

## 4) Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.