ADVANCED JAVA PROGRAMMING
NOTES BY: DR. PANKAJ MALIK
Unit-III: JDBC and JSP

What is JDBC?

JDBC is an acronym for Java Database Connectivity. It's an advancement for ODBC (Open Database Connectivity). JDBC is a standard API specification developed in order to move data from the front end to the back end. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer can send data from Java code and store it in the database for future use.

Why JDBC Come into Existence?

As previously told JDBC is an advancement for ODBC, ODBC being platform-dependent had a lot of drawbacks. ODBC API was written in C, C++, Python, and Core Java and as we know above languages (except Java and some part of Python)are platform-dependent. Therefore to remove dependence, JDBC was developed by a database vendor which consisted of classes and interfaces written in Java.

Steps to Connect Java Application with Database

Below are the steps that explains how to connect to Database in Java:

Step 1 – Import the Packages
Step 2 – Load the drivers using the *forName() method*
Step 3 – Register the drivers *using DriverManager*
Step 4 – Establish a connection *using the Connection class object*
Step 5 – Create a statement
Step 6 – Execute the query
Step 7 – Close the connections

Java Database Connectivity

Step 1: Import the Packages

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.ResultSet;

import java.sql.Statement;
```

Step 2: Loading the drivers

In order to begin with, you first need to load the driver or register it before using it in the program. Registration is to be done once in your program. You can register a driver in one of two ways mentioned below as follows:

2-A Class.forName()

Here we load the driver's class file into memory at the runtime. No need of using new or create objects. The following example uses Class.forName() to load the Oracle driver as shown below as follows:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2-B DriverManager.registerDriver()

DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time. The following example uses DriverManager.registerDriver()to register the Oracle driver as shown below:

```
 DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

Step 3: Establish a connection *using* the *Connection class object*

After loading the driver, establish connections as shown below as follows:

```
Connection con = DriverManager.getConnection(url,user,password)
```

- **user: U**sername from which your SQL command prompt can be accessed.
- **password:** password from which the SQL command prompt can be accessed.
- **con:** It is a reference to the Connection interface.
- **Url**: Uniform Resource Locator which is created as shown below:

```
String url = " jdbc:oracle:thin:@localhost:1521:xe"
```

Where oracle is the database used, thin is the driver used, @localhost is the IP Address where a database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by the programmer before calling the function. Use of this can be referred to form the final code.

Step 4: Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.
Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Step 5: Execute the query

Now comes the most important part i.e executing the query. The query here is an SQL Query. Now we know we can have multiple types of queries. Some of them are as follows:

- The query for updating/inserting a table in a database.
- The query for retrieving data.

The executeQuery() method of the **Statement interface** is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get

all the records of a table.
The executeUpdate(sql query) method of the Statement interface is used to execute queries of updating/inserting.
Pseudo Code:
```
int m = st.executeUpdate(sql);
if (m==1)
   System.out.println("inserted successfully : "+sql);
else
   System.out.println("insertion failed");
```

Step 6: Closing the connections
So finally we have sent the data to the specified location and now we are on the verge of completing our task. By closing the connection, objects of Statement and ResultSet will be closed automatically. The close() method of the Connection interface is used to close the connection. It is shown below as follows:
```
 con.close();
```

Configuring Data Source to obtain JDBC Connection
create a database table **Student** in our database **TEST**
```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```
Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code shown as follows
```
<bean id = "dataSource"
class =
"org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name = "driverClassName" value =
"com.mysql.cj.jdbc.Driver"/>
   <property name = "url" value =
"jdbc:mysql://localhost:3306/TEST"/>
   <property name = "username" value = "root"/>
   <property name = "password" value = "admin"/>
</bean>
```

Data Access operations with JDBC Template
The **org.springframework.jdbc.core.JdbcTemplate** class is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving the application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic.

Class Declaration

Following is the declaration for org.springframework.jdbc.core.JdbcTemplate class –

```
public class JdbcTemplate
   extends JdbcAccessor
      implements JdbcOperations
```

- **Step 1 –** Create a JdbcTemplate object using a configured datasource.
- **Step 2 –** Use JdbcTemplate object methods to make database operations.

Example

Following example will demonstrate how to read a query using JdbcTemplate class. We'll read the available records in Student Table.

```
String selectQuery = "select * from Student";
List <Student> students = jdbcTemplateObject.query(selectQuery,
new StudentMapper());
```

Following is the content of the Data Access Object interface file **StudentDAO.java**.

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
   /**
      * This is the method to be used to initialize
      * database resources ie. connection.
   */
   public void setDataSource(DataSource ds);

   /**
      * This is the method to be used to list down
      * all the records from the Student table.
   */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
   private Integer id;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   public Integer getId() {
      return id;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws
SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO.

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationConte
xt;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
      StudentJDBCTemplate studentJDBCTemplate =
(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      System.out.println("------Listing Multiple Records--------
" );
      List<Student> students =
studentJDBCTemplate.listStudents();
```

```
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
          System.out.println(", Age : " + record.getAge());
      }    }}
```

Following is the configuration file **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation =
"http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd ">

   <!-- Initialization for data source -->
   <bean id="dataSource"
      class =
"org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name = "driverClassName" value =
"com.mysql.cj.jdbc.Driver"/>
      <property name = "url" value =
"jdbc:mysql://localhost:3306/TEST"/>
      <property name = "username" value = "root"/>
      <property name = "password" value = "admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class = "com.tutorialspoint.StudentJDBCTemplate">
      <property name = "dataSource" ref = "dataSource" />
   </bean>
</beans>
```

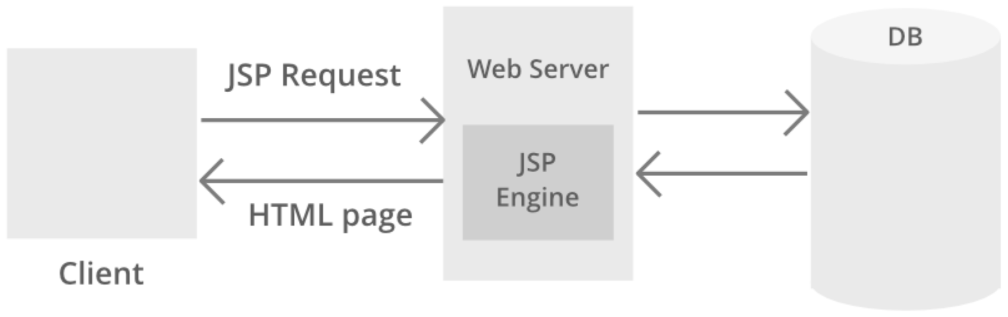
Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message.

```
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
```

JSP Architecture:

JSP architecture gives a high-level view of the working of JSP. JSP architecture is a 3 tier architecture. It has a Client, Web Server, and Database. The client is the web browser or application on the user side. Web Server uses a JSP Engine i.e; a container that processes JSP.

For example, Apache Tomcat has a built-in JSP Engine. JSP Engine intercepts the request for JSP and provides the runtime environment for the understanding and processing of JSP files. It reads, parses, build Java Servlet, Compiles and Executes Java code, and returns the HTML page to the client. The webserver has access to the Database.



JSP stands for Java Server Pages. It is a server-side technology. It is used for creating web applications. It is used to create dynamic web content. In this JSP tags are used to insert JAVA code into HTML pages. It is an advanced version of Servlet Technology. It is a Web-based technology that helps us to create dynamic and platform-independent web pages. In this, Java code can be inserted in HTML/ XML pages or both. JSP is first converted into a servlet by JSP container before processing the client's request. **JSP Processing** is illustrated and discussed in sequential steps prior to which a pictorial media is provided as a handful pick to understand the JSP processing better which is as follows:

Step 1: The client navigates to a file ending with the ***.jsp extension*** and the browser initiates an HTTP request to the webserver. For example, the user enters the login details and submits the button. The browser requests a status.jsp page from the webserver.

Step 2: If the compiled version of JSP exists in the web server, it returns the file. Otherwise, the request is forwarded to the JSP Engine. This is done by recognizing the URL ending with **.jsp** extension.

Step 3: The JSP Engine loads the JSP file and translates the JSP to Servlet(Java code). This is done by converting all the template text into println() statements and JSP elements to Java code. This process is called **translation.**

Step 4: The JSP engine compiles the Servlet to an executable **.class** file. It is forwarded to the Servlet engine. This process is called **compilation** or **request processing phase.**

Step 5: The **.class** file is executed by the Servlet engine which is a part of the Web Server. The output is an HTML file. The Servlet engine passes the output as an HTTP response to the webserver.

Step 6: The web server forwards the HTML file to the client's browser.

Life cycle of JSP

Following steps are involved in the JSP life cycle:

Translation of JSP page to Servlet:

This is the first step of the JSP life cycle. This translation phase deals with the Syntactic correctness of JSP. Here test.jsp file is translated to test.java.

1. **Compilation of JSP page:** Here the generated java servlet file (test.java) is compiled to a class file (test.class).
2. **Classloading:** The classloader loads the Java class file into the memory. The loaded Java class can then be used to serve incoming requests for the JSP page.
3. **Instantiation:** Here an instance of the class is generated. The container manages one or more instances by providing responses to requests.
4. **Initialization:** jspInit() method is called only once during the life cycle immediately after the generation of the Servlet instance from JSP.
5. **Request processing:** _jspService() method is used to serve the raised requests by JSP. It takes request and response objects as parameters. This method cannot be overridden.
6. **JSP Cleanup:** In order to remove the JSP from the use by the container or to destroy the method for servlets jspDestroy()method is used. This method is called once, if you need to perform any cleanup task like closing open files, or releasing database connections jspDestroy() can be overridden.

JSP building blocks Scripting Tags

The Scriptlet

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

Following is the syntax of Scriptlet −

```
<% code fragment %>
```

JSP Declarations

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

Following is the syntax for JSP Declarations −

```
<%! declaration; [ declaration; ]+ ... %>
```

JSP Expression

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression.

Following is the syntax of JSP Expression −

```
<%= expression %>
```

JSP Comments

JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out", a part of your JSP page.

Following is the syntax of the JSP comments −

```
<%-- This is JSP comment --%>
```

JSP Directives

A JSP directive affects the overall structure of the servlet class. It usually has the following form −

```
<%@ directive attribute="value" %>
```

There are three types of directive tag −

S.No.	Directive & Description
1	**<%@ page ... %>** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	**<%@ include ... %>** Includes a file during the translation phase.
3	**<%@ taglib ... %>** Declares a tag library, containing custom actions, used in the page

JSP Actions

JSP actions use **constructs** in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard −

```
<jsp:action_name attribute="value" />
```

implicit object

The request Object

The request object is an instance of a **javax.servlet.http.HttpServletRequest** object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc.

The response Object

The response object is an instance of a **javax.servlet.http.HttpServletResponse** object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

The out Object

The out implicit object is an instance of a **javax.servlet.jsp.JspWriter** object and is used to send content in a response.

The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the **buffered = 'false'** attribute of the page directive.

The JspWriter object contains most of the same methods as the **java.io.PrintWriter** class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws **IOExceptions**.

Following table lists out the important methods that we will use to write **boolean char, int, double, object, String**, etc.

S.No.	**Method & Description**
1	**out.print(dataType dt)** Print a data type value
2	**out.println(dataType dt)** Print a data type value then terminate the line with new line character.
3	**out.flush()** Flush the stream.

The session Object

The session object is an instance of **javax.servlet.http.HttpSession** and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests.

The application Object

The application object is direct wrapper around the **ServletContext** object for the generated Servlet and in reality an instance of a **javax.servlet.ServletContext** object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

The config Object

The config object is an instantiation of **javax.servlet.ServletConfig** and is a direct wrapper around the **ServletConfig** object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following **config** method is the only one you might ever use, and its usage is trivial −

```
config.getServletName();
```

The pageContext Object

The pageContext object is an instance of a **javax.servlet.jsp.PageContext** object. The pageContext object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The **application, config, session**, and out objects are derived by accessing attributes of this object.

The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including **PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE,** and **APPLICATION_SCOPE**, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the **javax.servlet.jsp.JspContext class**.

One of the important methods is **removeAttribute**. This method accepts either one or two arguments. For example, **pageContext.removeAttribute ("attrName")** removes the attribute from all scopes, while the following code only removes it from the page scope −

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the **this** object.

The exception Object

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

Introduction to Bean:

JavaBeans are classes that encapsulate many objects into a single object (the bean). It is a Java class that should follow the following conventions:

1. Must implement Serializable.
2. It should have a public no-arg constructor.
3. All properties in java bean must be private with public getters and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

avaBean Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define. A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyName ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

Advantages of JavaBean

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

standard actions

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard −

<jsp:action_name attribute = "value" />

Action elements are basically predefined functions.

1. The <jsp:include> Action

This action lets you insert files into the page being generated. The syntax looks like this −

<jsp:include page = "relative URL" flush = "true" />

Unlike the **include** directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

2. The <jsp:useBean> Action

The **useBean** action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows −

```
<jsp:useBean id = "name" class = "package.class" />
```

Once a bean class is loaded, you can use **jsp:setProperty** and **jsp:getProperty** actions to modify and retrieve the bean properties.

3. The <jsp:setProperty> Action

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action −

You can use **jsp:setProperty** after, but outside of a **jsp:useBean** element, as given below −

```
<jsp:useBean id = "myName" ... />
...
<jsp:setProperty name = "myName" property = "someProperty" .../>
```

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

4. The <jsp:getProperty> Action

The **getProperty** action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output.

The getProperty action has only two attributes, both of which are required. The syntax of the getProperty action is as follows −

```
<jsp:useBean id = "myName" ... />
...
<jsp:getProperty name = "myName" property = "someProperty" .../>
```

5. The <jsp:forward> Action

The **forward** action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.

Following is the syntax of the **forward** action −

```
<jsp:forward page = "Relative URL" />
```

6. The <jsp:plugin> Action

The **plugin** action is used to insert Java components into a JSP page. It determines the type of browser and inserts the **<object>** or **<embed>** tags as needed.

If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean.

The plugin action has several attributes that correspond to common HTML tags used to format Java components. The **<param>** element can also be used to send parameters to the Applet or Bean.

The <jsp:text> Action

The **<jsp:text>** action can be used to write the template text in JSP pages and documents. Following is the simple syntax for this action −

```
<jsp:text>Template data</jsp:text>
```

The body of the template cannot contain other elements; it can only contain text and EL expressions (Note − EL expressions are explained in a subsequent chapter). Note that in XML files, you cannot use expressions such as **${whatever > 0}**, because the greater than signs are illegal. Instead, use the **gt** form, such as **${whatever gt 0}** or an alternative is to embed the value in a **CDATA** section.

session tracking types and methods:

Servlets are the Java programs that run on the Java-enabled web server or application server. They are used to handle the request obtained from the webserver, process the request, produce the response, then send a response back to the webserver
HTTP is a “stateless” protocol, which means that each time a client requests a Web page, the client establishes a new connection with the Web server, and the server does not retain track of prior requests.

- The conversation of a user over a period of time is referred to as a **session**. In general, it refers to a certain period of time.
- The recording of the object in session is known as **tracking**.
- **Session tracking** is the process of remembering and documenting customer conversations over time. Session management is another name for it.
- The term “**stateful web application**” refers to a web application that is capable of remembering and recording client conversations over time.

Why is Session Tracking Required?

- Because the HTTP protocol is stateless, we require Session Tracking to make the client-server relationship stateful.
- Session tracking is important for tracking conversions in online shopping, mailing applications, and E-Commerce applications.
- The HTTP protocol is stateless, which implies that each request is treated as a new one.

Session Tracking employs Four Different techniques

A. Cookies

Cookies are little pieces of data delivered by the web server in the response header and kept by the browser. Each web client can be assigned a unique session ID by a web server. Cookies are used to keep the session going. Cookies can be turned off by the client.

B. Hidden Form Field

The information is inserted into the web pages via the hidden form field, which is then transferred to the server. These fields are hidden from the user's view.

Illustration**:**

```
<input type = hidden'  name = 'session' value = '12345' >
```

C. URL Rewriting

With each request and return, append some more data via URL as request parameters. URL rewriting is a better technique to keep session management and browser operations in sync.

D. HttpSession

A user session is represented by the HttpSession object. A session is established between an HTTP client and an HTTP server using the HttpSession interface. A user session is a collection of data about a user that spans many HTTP requests.

Custom Tags:

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

To write a custom tag, you can simply extend **SimpleTagSupport** class and override the **doTag()** method, where you can place your code to generate content for the tag.

Advantages of Custom Tags

The key advantages of Custom tags are as follows:

1. **Eliminates the need of scriptlet tag** The custom tags eliminates the need of scriptlet tag which is considered bad programming approach in JSP.
2. **Separation of business logic from JSP** The custom tags separate the the business logic from the JSP page so that it may be easy to maintain.
3. **Re-usability** The custom tags makes the possibility to reuse the same business logic again and again.

Example to create the **HelloTag** class:

```
package com.tutorialspoint;

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

public class HelloTag extends SimpleTagSupport {
   public void doTag() throws JspException, IOException {
      JspWriter out = getJspContext().getOut();
      out.println("Hello Custom Tag!");
   }
}
```

Introduction to JSP Standard Tag Library (JSTL)

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates the core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating the existing custom tags with the JSTL tags.

The JSTL tags can be classified, according to their functions, into the following JSTL tag library groups that can be used when creating a JSP page −

- Core Tags
- Formatting tags
- SQL tags
- XML tags
- JSTL Functions

Core Tags

The core group of tags are the most commonly used JSTL tags. Following is the syntax to include the JSTL Core library in your JSP −

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

S.No.	Tag & Description
1	**<c:out>** Like <%= ... >, but for expressions.
2	**<c:set >** Sets the result of an expression evaluation in a **'scope'**
3	**<c:remove >** Removes a **scoped variable** (from a particular scope, if specified).
4	**<c:catch>** Catches any **Throwable** that occurs in its body and optionally exposes it.
5	**<c:if>** Simple conditional tag which evalutes its body if the supplied condition is true.
6	**<c:choose>** Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by **<when>** and **<otherwise>**.
7	**<c:when>** Subtag of **<choose>** that includes its body if its condition evalutes to **'true'**.
8	**<c:otherwise >** Subtag of **<choose>** that follows the **<when>** tags and runs only if all of the prior conditions evaluated to **'false'**.
9	**<c:import>** Retrieves an absolute or relative URL and exposes its contents to either the page, a String in **'var'**, or a Reader in **'varReader'**.
10	**<c:forEach >** The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
11	**<c:forTokens>** Iterates over tokens, separated by the supplied delimeters.
12	**<c:param>** Adds a parameter to a containing **'import'** tag's URL.
13	**<c:redirect >** Redirects to a new URL.
14	**<c:url>** Creates a URL with optional query parameters

Formatting Tags

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Websites. Following is the syntax to include Formatting library in your JSP −

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

S.No.	Tag & Description
1	**<fmt:formatNumber>** To render numerical value with specific precision or format.
2	**<fmt:parseNumber>** Parses the string representation of a number, currency, or percentage.
3	**<fmt:formatDate>** Formats a date and/or time using the supplied styles and pattern.
4	**<fmt:parseDate>** Parses the string representation of a date and/or time
5	**<fmt:bundle>** Loads a resource bundle to be used by its tag body.
6	**<fmt:setLocale>** Stores the given locale in the locale configuration variable.
7	**<fmt:setBundle>** Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
8	**<fmt:timeZone>** Specifies the time zone for any time formatting or parsing actions nested in its body.
9	**<fmt:setTimeZone>** Stores the given time zone in the time zone configuration variable
10	**<fmt:message>** Displays an internationalized message.
11	**<fmt:requestEncoding>** Sets the request character encoding

SQL Tags

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as **Oracle, mySQL**, or **Microsoft SQL Server**.

Following is the syntax to include JSTL SQL library in your JSP −

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

S.No.	Tag & Description
1	**<sql:setDataSource>** Creates a simple DataSource suitable only for prototyping
2	**<sql:query>** Executes the SQL query defined in its body or through the sql attribute.
3	**<sql:update>** Executes the SQL update defined in its body or through the sql attribute.
4	**<sql:param>** Sets a parameter in an SQL statement to the specified value.
5	**<sql:dateParam>** Sets a parameter in an SQL statement to the specified java.util.Date value.
6	**<sql:transaction >** Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

XML tags

The JSTL XML tags provide a JSP-centric way of creating and manipulating the XML documents. Following is the syntax to include the JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with the XML data. This includes parsing the XML, transforming the XML data, and the flow control based on the XPath expressions.

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

S.No.	Tag & Description
1	**<x:out>** Like <%= ... >, but for XPath expressions.
2	**<x:parse>** Used to parse the XML data specified either via an attribute or in the tag body.

3	**<x:set >** Sets a variable to the value of an XPath expression.
4	**<x:if >** Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
5	**<x:forEach>** To loop over nodes in an XML document.
6	**<x:choose>** Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by **<when>** and **<otherwise>** tags.
7	**<x:when >** Subtag of **<choose>** that includes its body if its expression evalutes to 'true'.
8	**<x:otherwise >** Subtag of **<choose>** that follows the **<when>** tags and runs only if all of the prior conditions evaluates to 'false'.
9	**<x:transform >** Applies an XSL transformation on a XML document
10	**<x:param >** Used along with the **transform** tag to set a parameter in the XSLT stylesheet

JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP −

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

S.No.	Function & Description
1	**fn:contains()** Tests if an input string contains the specified substring.
2	**fn:containsIgnoreCase()** Tests if an input string contains the specified substring in a case insensitive way.
3	**fn:endsWith()** Tests if an input string ends with the specified suffix.
4	**fn:escapeXml()** Escapes characters that can be interpreted as XML markup.
5	**fn:indexOf()** Returns the index withing a string of the first occurrence of a specified substring.
6	**fn:join()** Joins all elements of an array into a string.

7	**fn:length()** Returns the number of items in a collection, or the number of characters in a string.
8	**fn:replace()** Returns a string resulting from replacing in an input string all occurrences with a given string.
9	**fn:split()** Splits a string into an array of substrings.
10	**fn:startsWith()** Tests if an input string starts with the specified prefix.
11	**fn:substring()** Returns a subset of a string.
12	**fn:substringAfter()** Returns a subset of a string following a specific substring.
13	**fn:substringBefore()** Returns a subset of a string before a specific substring.
14	**fn:toLowerCase()** Converts all of the characters of a string to lower case.
15	**fn:toUpperCase()** Converts all of the characters of a string to upper case.
16	**fn:trim()** Removes white spaces from both ends of a string.