Unit – 5 JSP

JSP: JSP architecture, Life Cycle, Creating Simple JSP Pages, JSP Basic tags, Implicit Objects.

.

Problems with Servlets

- Servlets need a special "servlet container" to run servlets.
- Servlets need a Java Runtime Environment on the server to run servlets.
- For developing Servlet based application, knowledge of java as well as HTML code is necessary.

- The servlet has to do various tasks such as acceptance of request, processing of request, handling of business logic and generation of response.

- In many Java servlet-based applications, processing the request and generating the response are both handled by a single servlet class.

  An example servlet looks like this:

```
public class OrderServlet extends HttpServlet
 {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
response.setContentType("text/html");
PrintWriter out = response.getWriter( );
if (isOrderInfoValid(request)) {
saveOrderInfo(request);
out.println("<html>");
```

```
out.println(" <head>");
out.println(" <title>Order Confirmation</title>");
out.println(" </head>");
out.println(" <body>");
out.println(" <h1>Order Confirmation</h1>");
renderOrderInfo(request);
out.println(" </body>");
out.println("</html>");
}
}
```

The pure servlet-based approach still has a few problems:

- Detailed Java programming knowledge is needed to develop and maintain all aspects of the application, since the processing code and the HTML elements are lumped together.

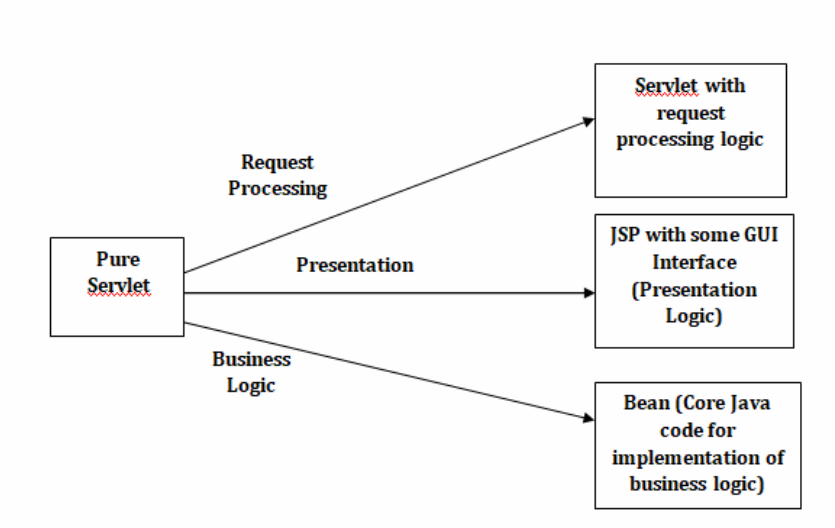
- Changing the look and feel of the application, or adding support for a new type of client (such as a WML client), requires the servlet code to be updated and recompiled.

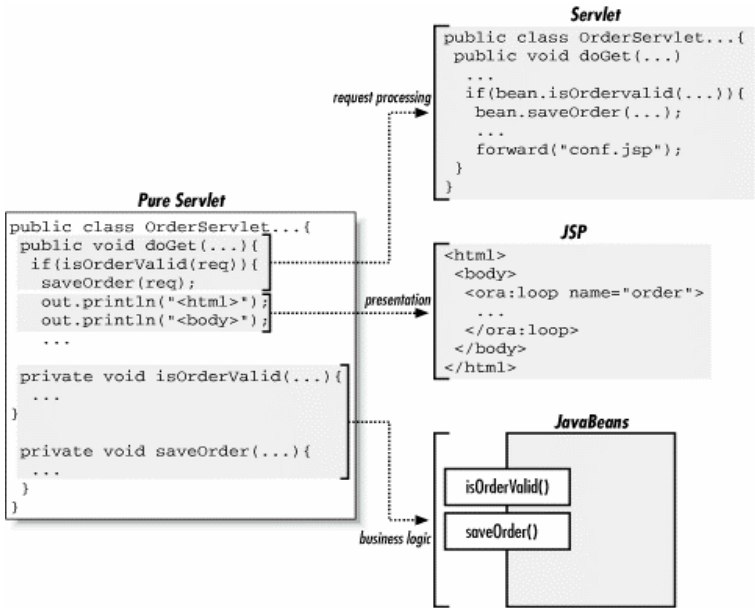
- It's hard to take advantage of web page development tools when designing the application interface. If such tools are used to develop the web page layout, the generated HTML must then be manually embedded into the servlet code, a process that is time-consuming and error-prone

JSP lets you solve these problems "*by separating the request processing and business logic code from the presentation",* as illustrated in Figure 1.1. Instead of embedding HTML in the code, you place all static HTML in JSP pages, just as in a regular web page, and

add a few JSP elements to generate the dynamic parts of the page. The request processing can remain the domain of servlet programmers, and the business logic can be handled by JavaBeans and Enterprise JavaBeans (EJB) components.

Figure 1.1. Separation of request processing, business logic, and presentation





Advantages of JSP:

- Separating the request processing and business logic from presentation makes it possible to divide the development tasks among people with different skills. Java programmers implement the request processing and business logic pieces, web page authors implement the user interface, and both groups can use best-of-breed development tools for the task at hand. The result is a *much more productive development process*.
- It also makes it *possible to change different aspects of the application independently*, such as changing the business rules without touching the user interface.
- This model has clear benefits even for a web page author *without programming skills* who is working alone. A page author can develop web applications with many dynamic features, using generic Java components provided by open source projects or commercial companies.

- It provides a very powerful and flexible mechanism to produce dynamic web pages.
- Dynamic contents can be handled using JSP because JSP allows scripting and element based programming.
- JSP allows creating and using our own custom tag libraries. Hence any application specific requirements can be satisfied using custom tag libraries. This helps the developer to develop any kind of application.
- JSP is an essential component of J2EE. Hence using JSP it is possible to develop simple as well as complex applications.
- In JSP we can directly embed java code into html code but in servlet is not possible.
- JSP page is automatically compiled but servlet will manually redeploy.
- In jsp implicit objects are presents which is we can implement directly into jsp pages but in servlet there are no implicit objects.

Introduction to JSP

JSP Overview

- JavaServer Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications.
- JavaServer Pages (JSP) is a technology for developing web pages that support dynamic content which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.
- JSP is a specification and not a product.Hence developers can develop variety of applications and add up to performance

and quality of software products.It is essential component of J2EE.

- A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.
- Using JSP, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.
- JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages etc.

Why Use JSP?

- Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself.
- JSP allows to separate the presentation logic and business logic
- JSP are always compiled before it's processed by the server
- JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, EJB, JAXP etc.
- JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

JSP ARCHITECTURE

Anatomy of a JSP page

JSP page is simply a regular web page with JSP elements for generating the parts of the page that differ for each request.

The JSP Page consists of 2 parts:

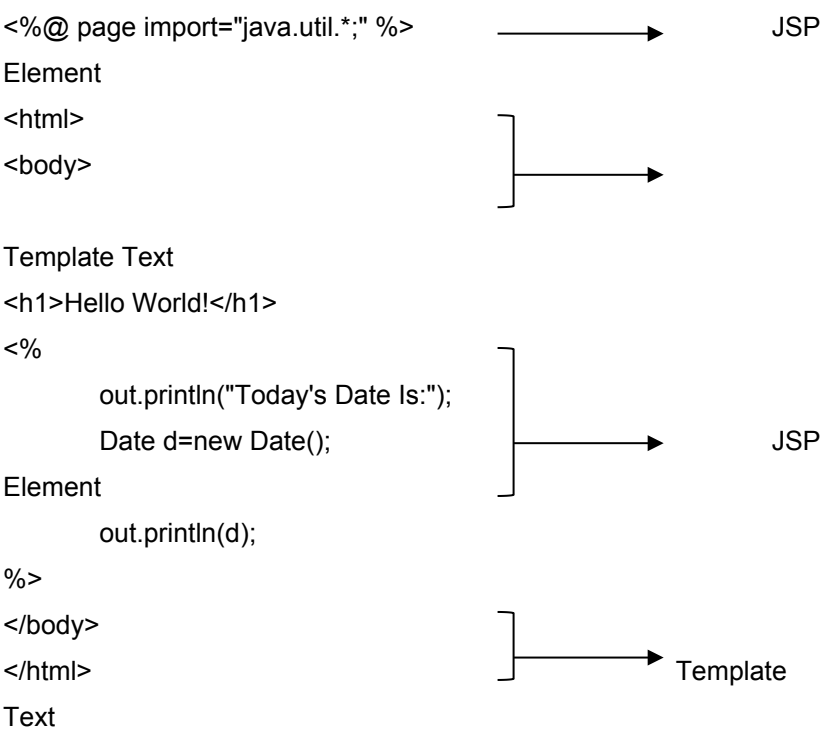
i) Template text

ii) JSP Elements

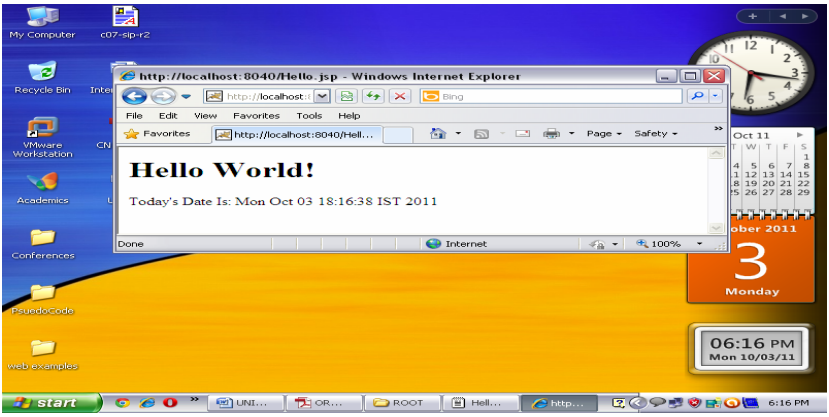
i) Template Text:

Everything in the page that is not a JSP element is called *template text*. Template text can really be any text: HTML, WML, XML, or even plain text. Template text is always passed straight through to the browser.

Figure 1.2. Template text and JSP elements



OUTPUT:



When a JSP page request is processed, the template text and the dynamic content generated by the JSP elements are merged, and the result is sent as the response to the browser.

ii) JSP Elements:

There are three types of elements with Java Server Pages:

***Directive*,**
***Aaction*, and**
***Scripting E*lements.**

(Additional Elements are **JSP Implicit Objects**)

The Directive elements, shown in Table 1.1, are used to specify information about the page itself that remains the same between page requests, for example, the scripting language used in the page, whether session tracking is required, and the name of a page that should be used to report errors, if any.

Table 1.1 Directive elements

Table 3.1, Directive Elements	
Element	**Description**
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements
<%@ include ... %>	Includes a file during the translation phase
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

Action elements typically perform some action based on information that is required at the exact time the JSP page is requested by a client. An action element can, for instance, access parameters sent with the request to do a database lookup. It can also dynamically generate HTML, such as a table filled with information retrieved from an external system. The JSP specification defines a few standard action elements, listed in Table 1.2, and includes a framework for developing custom action elements. A custom action element can be developed by a programmer to extend the JSP language.

Table 1.2 Action Elements

Table 3.2, Standard Action Elements	
Element	**Description**
<jsp:useBean>	Makes a JavaBeans component available in a page
<jsp:getProperty>	Gets a property value from a JavaBeans component and adds it to the response
<jsp:setProperty>	Sets a JavaBeans property value
<jsp:include>	Includes the response from a servlet or JSP page during the request processing phase
<jsp:forward>	Forwards the processing of a request to a servlet or JSP page
<jsp:param>	Adds a parameter value to a request handed off to another servlet or JSP page using <jsp:include> or <jsp:forward>
<jsp:plugin>	Generates HTML that contains the appropriate client browser-dependent elements (OBJECT or EMBED) needed to execute an Applet with the Java Plugin software

Scripting elements, shown in Table 1.3, allow you to add small pieces of code to a JSP page, such as an if statement to generate different HTML depending on a certain condition. Like actions, they are also executed when the page is requested. Scripting elements must be used with extreme care: if you embed too much code in your

JSP pages, you will end up with the same kind of maintenance problems as with servlets embedding HTML

1. Scriptlets
2. Expressions
3. Declarations

Table 1.3 Scripting elements

Table 3.3, Scripting Elements	
Element	**Description**
<% ... %>	Scriptlet, used to embed scripting code.
<%= ... %>	Expression, used to embed Java expressions when the result shall be added to the response. Also used as runtime action attribute values.
<%! ... %>	Declaration, used to declare instance variables and methods in the JSP page implementation class.

LIFE CYCLE

JSP Processing/ Life Cycle of JSP:

A JSP page cannot be sent as-is to the browser; all JSP elements must first be processed by the server. This is done by turning the JSP page into a servlet, and then executing the servlet.

JSP Container:

Just as a web server needs a servlet container to provide an interface to servlets, the server needs a JSP container to process JSP pages. The JSP container is often implemented as a servlet configured to handle all requests for JSP pages. In fact, these two containers - a servlet container and a JSP container - are often combined into one package under the name *web container.*

JSP Processing is done in 2 phases:

i) Translation Phase

ii) Request Processing Phase

i) Translation Phase:

A JSP container is responsible for *converting the JSP page into a servlet* (known as the *JSP page implementation class)* and *compiling the servlet.* These two steps form the *translation phase* . The JSP container automatically initiates the translation phase for a page when the first request for the page is received. The translation phase can also be initiated explicitly; this is referred to as *precompilation* of a JSP page.

When a JSP container receives a jsp request, it checks for the jsp's servlet instance. If no servlet instance is available, then, the container creates the servlet instance using following stages.

- Translation
- Compilation
- Loading
- Instantiation
- Initialization

Translation - In this step the JSP page is translated into the corresponding Servlet.

Compilation - Once the JSP page has been translated into the corresponding Servlet, the next obvious step is to compile that Servlet.

Loading & Instantiation - As is the case with any compiled class (.class file), this servlet class also needs to be loaded into the memory before being used. The default class loader of the Container will load this class. Once the class is loaded, an instance of this class gets created.

Initialization: JspPage interface contains the jspInit() method, which is used by the JSP container to initialize the newly created instance. This jspInit() method is just like the init()method of the Servlet and it's called only once during the entire life cycle of a JSP/Servlet.

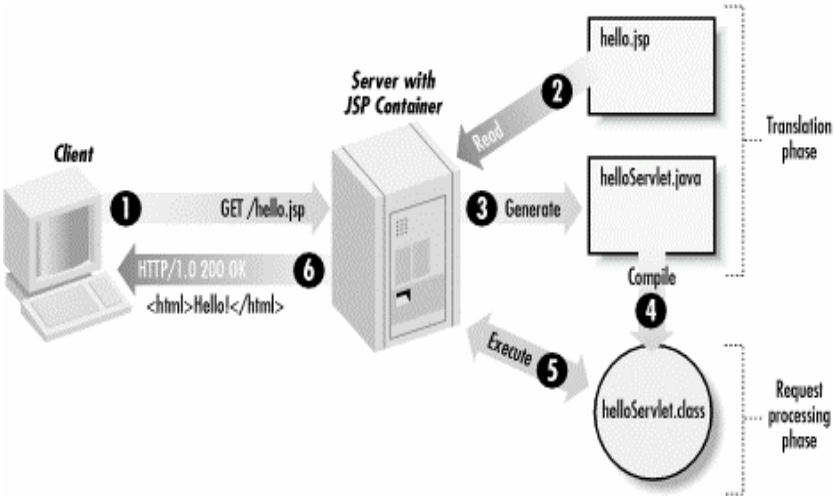
ii) Request Processing Phase:

The JSP container is also responsible for invoking the JSP page implementation class to process each request and generate the response. This is called the *request processing phase*.

_jspService() is the method which is called every time the JSP is requested to serve a request. This method normally executes in a separate thread of execution and the main JSP thread keeps waiting for other incoming requests. Every time a request arrives, the main JSP thread spawns a new thread and passes the request (incoming request) and response (new) objects to the_jspService() method which gets executed in the newly spawned thread.

The two phases are illustrated in Figure 1.3.

Figure 1.3. JSP page translation and request processing phases



As long as the JSP page remains unchanged, any subsequent processing goes straight to the request processing phase (i.e., it simply executes the class file).

When the JSP page is modified, it goes through the translation phase again before entering the request processing phase. So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming expert.

And, except for the translation phase, a JSP page is handled exactly like a regular servlet: it's loaded once and called repeatedly, until the server is shut down.

By virtue of being an automatically generated servlet, a JSP page inherits all of the advantages of servlets: platform and vendor independence, integration, efficiency, scalability, robustness, and security.

Life Cycle Methods:

i). The jspInit()- The container calls the jspInit() to initialize te servlet instance.It is called before any other method, and is called only once for a servlet instance.

```
public void jspInit(){
  // Initialization code...
}
```

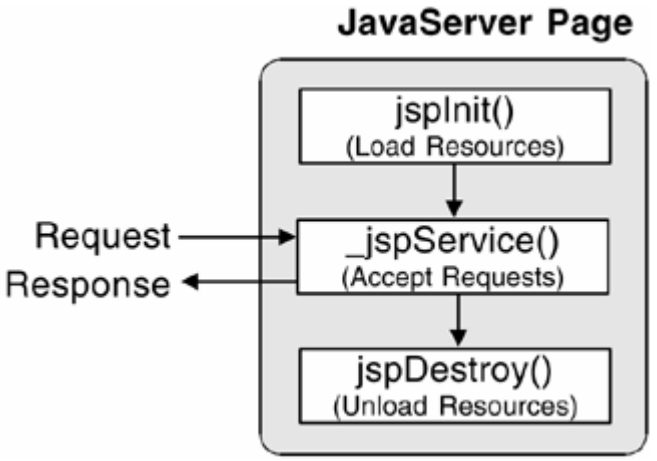
ii). The _jspservice()- The container calls the _jspservice() for each request, passing it the request and the response objects.

```
void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
{
  // Service handling code...
}
```

iii). The jspDestroy()- The container calls this when it decides to take the instance out of service. It is the last method called in the servlet instance.

```
public void jspDestroy()
{
  // cleanup code goes here.
}
```

Figure 1.4 Life Cycle Methods of JSP



Generating Dynamic Content:

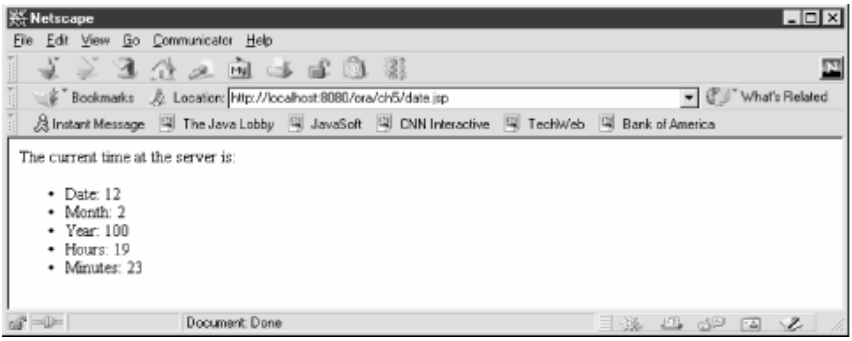
- JSP is all about generating dynamic content: content that differs based on user input, time of day, the state of an external system, or any other runtime conditions.
- JSP provides you with lots of tools for generating this content.
- Dynamic content can be generated using all JSP Elements
  standard actions,
  custom actions,
  JavaBeans, and
  scripting elements
  Directives

JSP Page Showing the Current Date and Time (date.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
 <body bgcolor="white">

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

  The current time at the server is:
  <ul>
   <li>Date: <jsp:getProperty name="clock" property="date" />
   <li>Month: <jsp:getProperty name="clock" property="month" />
   <li>Year: <jsp:getProperty name="clock" property="year" />
   <li>Hours: <jsp:getProperty name="clock" property="hours" />
   <li>Minutes: <jsp:getProperty name="clock" property="minutes" />
  </ul>

 </body>
</html>
```

The *date.jsp* page displays the current date and time.

Output of date.jsp example



Using Scripting Elements ,Directive, Action, Scriptlet:

Using JSP Directives

- Directives are used to specify attributes of the page itself, primarily those that affect how the page is converted into a Java servlet.
- There are three JSP directives:

  page, include, and taglib.

- JSP pages typically start with a page directive that specifies the scripting language and the content type for the page:

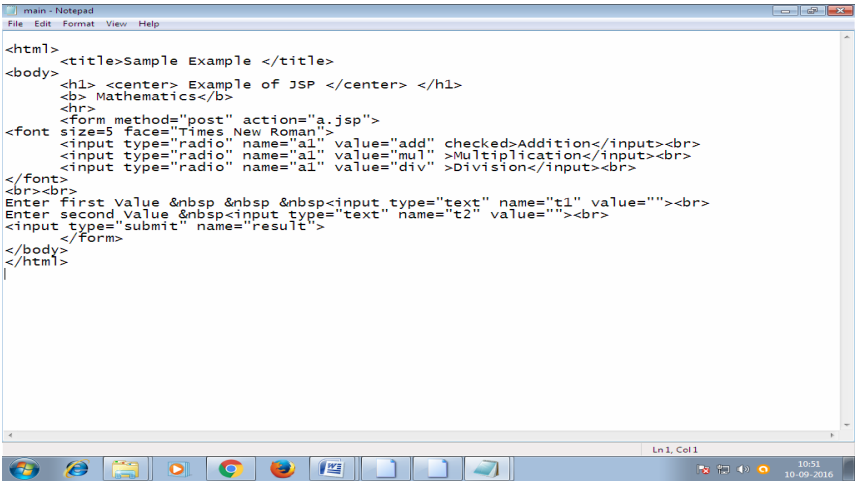
```
<%@ page language="java" contentType="text/html" %>
```

- A JSP directive element starts with a directive-start identifier (<%@) followed by the directive name (e.g., page) and directive attributes, and ends with %>.
- A directive contains one or more attribute name/value pairs (e.g.,language="java").
- Note that JSP element and attribute names are case-sensitive, and in most cases the same is true for attribute values.

- For instance, the language attribute value must be java, not Java.
- All attribute values must also be enclosed in single or double quotes.
- The page directive has many possible attributes

Example:

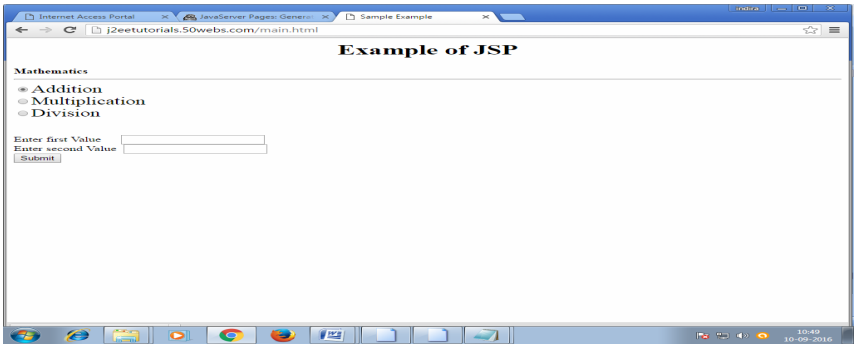
Main.html



a.jsp

```
<%@ page language="java"%>
<%@ page import="java.lang.*"%>
<html>
<body>
<H1><center>Result for <%=request.getParameter("a1")%></center></H1>
<%
int i=Integer.parseInt(request.getParameter("t1"));
int j=Integer.parseInt(request.getParameter("t2"));
int k=0;
String str=request.getParameter("a1");

if(str.equals("add"))
  k=i+j;
if(str.equals("mul"))
  k=i*j;
if(str.equals("div"))
  k=i/j;
%>
Result is <%=k%>
</body>
</html>
```

Output:



JSP Scripting Elements

- Expressions
  - Format: <%= expression %>
- Scriptlets
  - Format: <% code %>
- Declarations
  - Format: <%! code %>

JSP Expressions

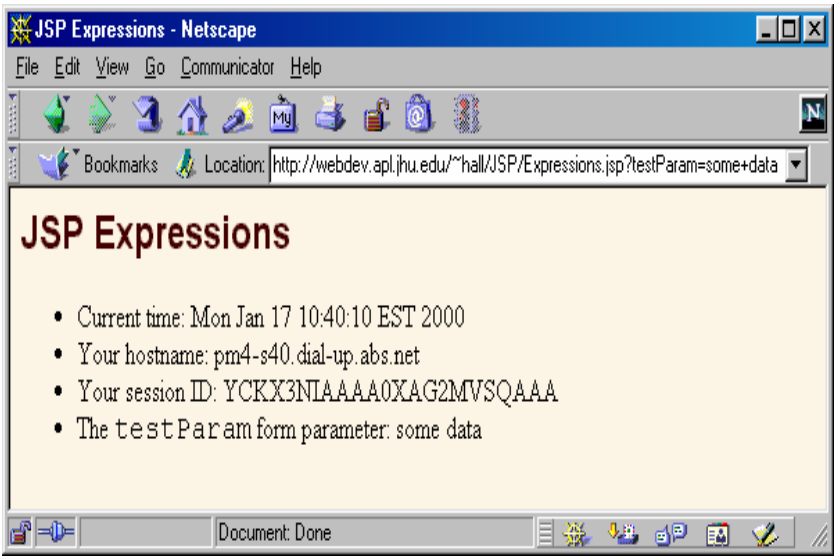
- Format

- <%= Java Expression %>

- Result
  - expression placed in _jspService inside out.print
- Examples
  - Current time: <%= new java.util.Date() %>
  - <%= request.getRemoteHost() %>
- XML-compatible syntax
  - <jsp:expression>Java Expression</jsp:expression>

Example:

Output:



Predefined Variables

- request
  - The HttpServletRequest (1st argument to service/doGet)
- response
  - The HttpServletResponse (2nd arg to service/doGet)
- out
  - The Writer (a buffered version of type JspWriter) used to send output to the client
- session
  - The HttpSession associated with the request (unless disabled with the session attribute of the page directive)
- application
  - The ServletContext (for sharing data) as obtained via getServletContext().

JSP Scriptlets

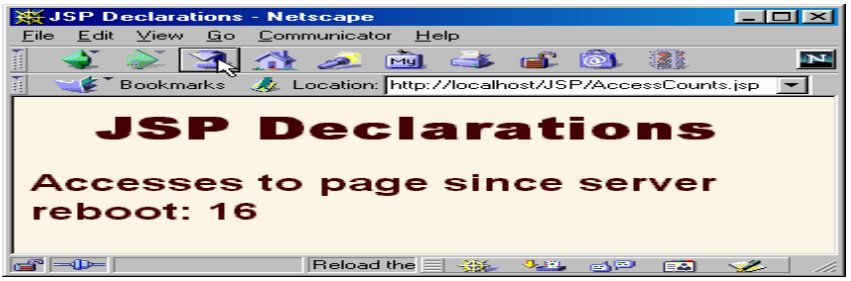
- Format
  - <% Java Code %>
- Result
  - Code is inserted verbatim into servlet's _jspService
- Example
  - <%
    String queryData = request.getQueryString();
    out.println("Attached GET data: " + queryData);
    %>
  - <% response.setContentType("text/plain"); %>
- XML-compatible syntax
  <jsp:scriptlet>Java Code</jsp:scriptlet>

JSP Declarations

- Format
  - <%! Java Code %>
- Result
  - Code is inserted verbatim into servlet's class definition, outside of any existing methods
- Examples
  - <%! private int someField = 5; %>
  - <%! private void someMethod(...) {...} %>
- XML-compatible syntax
  - <jsp:declaration>Java Code</jsp:declaration>

Example:

Output:



Declaring Variables and Methods

The **JSP declaration tag** is used *to declare fields and methods*.

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

1. <%! field or method declaration %>

Difference between JSP Scriptlet tag and Declaration tag

Jsp Scriptlet Tag	Jsp Declaration Tag
The jsp scriptlet tag can only declare variables not methods.	The jsp declaration tag can declare variables as well as methods.
The declaration of scriptlet tag is placed inside the _jspService() method.	The declaration of jsp declaration tag is placed outside the _jspService() method.

Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

index.jsp

1. <html>

```
2.      <body>
3.      <%! int data=50; %>
4.      <%= "Value of the variable is:"+data %>
5.      </body>
6.      </html>
```

Example of JSP declaration tag that declares method

In this example of JSP declaration tag, we are defining the method which returns the cube of given number and calling this method from the jsp expression tag. But we can also use jsp scriptlet tag to call the declared method.

index.jsp

```
1.      <html>
2.      <body>
3.      <%!
4.      int cube(int n){
5.      return n*n*n*;
6.      }
7.      %>
8.      <%= "Cube of 3 is:"+cube(3) %>
```

9. </body>

10. </html>

IMPLICIT JSP OBJECTS:

JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared.

JSP Implicit Objects are also called pre-defined variables.

JSP supports nine Implicit Objects which are listed below:

Object	Description
Request	This is the **HttpServletRequest** object associated with the request.
Response	This is the **HttpServletResponse** object associated with the response to the client.
Out	This is the **PrintWriter** object used to send output to the client.
Session	This is the **HttpSession** object associated with the request.
Application	This is the **ServletContext** object associated with application context.
Config	This is the **ServletConfig** object associated with the page.
pageContext	This encapsulates use of server-specific features like higher performance **JspWriters**.
Page	This is simply a synonym for **this**, and is used to call

	the methods defined by the translated servlet class.
Exception	The **Exception** object allows the exception data to be accessed by designated JSP.

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 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 are the important methods which we would use to write boolean char, int, double, object, String etc.

Method	Description
out.print(dataType dt)	Print a data type value
out.println(dataType dt)	Print a data type value then terminate the line with new line character.
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();
```

This returns the servlet name, which is the string contained in the <servlet-name> element defined in the WEB-INF\web.xml file

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**, which 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.

Database Connectivity

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

Steps to connect database in java using JDBC are given below:

1. Load the JDBC driver.
2. Connection.
3. Statement.
4. Execute statement.
5. Close database connection.

1. Load the JDBC driver:

First step is to load or register the JDBC driver for the database. Class class provides forName() method to dynamically load the driver class.

Syntax:

Class.forName("driverClassName");

To load or register OracleDriver class:

Syntax:

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

To load or register MySQL class:

Syntax:

Class.forName("com.mysql.jdbc.Driver");

2. Create connection:

Second step is to open a database connection. DriverManager class provides the facility to create a connection between a database and the appropriate driver.To open a database connection we can call getConnection() method of DriverManager class.

Syntax:

Connection connection = DriverManager.getConnection(url, user, password);

To create a connection with Oracle database:

Syntax:

Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","user","password");

To create a connection with **MySQL** database:

Syntax:

Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root"," ");

3. Create statement:

The statement object is used to execute the query against the database. Connection interface acts as a factory for statement object. A statement object can be any one of the *Statement, CallableStatement,* and *PreparedStatement types* .To create a statement object we have to call createStatement() method of Connection interface. Connection interface also provides the transaction management methods like commit() and rollback() etc.

Syntax:

Statement stmt=conn.createStatement();

4. Execute statement:

Statement interface provides the methods to execute a statement.

To execute a statement for select query use below:

Syntax:

ResultSet resultSet = stmt.executeQuery(selectQuery);

5. Close database connection:

After done with the database connection we have to close it. Use close() method of Connection interface to close database connection. The statement and ResultSet objects will be closed automatically when we close the connection object.

Syntax:

connection.close();

Example

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
}
catch(Exception e)
{
out.println(e.toString());

}
%>
```

Studying Javax.sql.* package

Provides the API for server side data source access and processing from the Java™ programming language. This package supplements the java.sql package.

The javax.sql package provides for the following:

1. The DataSource interface as an alternative to the DriverManager for establishing a connection with a data source
2. Connection pooling and Statement pooling
3. Distributed transactions
4. Rowsets

Applications use the DataSource and RowSet APIs directly, but the connection pooling and distributed transaction APIs are used internally by the middle-tier infrastructure.

CRUD OPERATIONS.

Basic database operations (CRUD - Create, Retrieve, Update and Delete) using JDBC (Java Database Connectivity) API. These CRUD

operations are equivalent to the INSERT, SELECT, UPDATE and DELETE statements in SQL language.

Create a table

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql="CREATE TABLE studentbalu " + "(sno INTEGER not
Null,"+
                        " sname VARCHAR2(20)," +
                                        " age VARCHAR2(20))";
stmt.executeUpdate(sql);
System.out.println("Create table in given database...");

}
catch(Exception e)
{
out.println(e.toString());
}
%>
```

Insert operation

```
<%@page import="java.sql.*" %>
```

```
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql= sql = "INSERT INTO studentbalu VALUES (101, 'balu',
25)";
stmt.executeUpdate(sql);
 System.out.println("Inserted records into the table...");



}
catch(Exception e)
{
out.println(e.toString());
}
%>
```

Update operation

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
```

```
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql=  "UPDATE studentbalu " +
          "SET age = 42 WHERE sno=101";
stmt.executeUpdate(sql);
 System.out.println("Updated records into the table...");



}
catch(Exception e)
{
out.println(e.toString());

}
%>
```

Delete operation

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
```

```
out.println("Connect set");
Statement stmt=c.createStatement();
String sql = "DELETE FROM studentbalu " +
          "WHERE sno = 101";
stmt.executeUpdate(sql);
 System.out.println("deleted records into the table...");



}
catch(Exception e)
{
out.println(e.toString());

}
%>
```

Accessing a database from a JSP Page

Sample programs to access database from JSP:

Write a JSP to display employee number and name from emp table

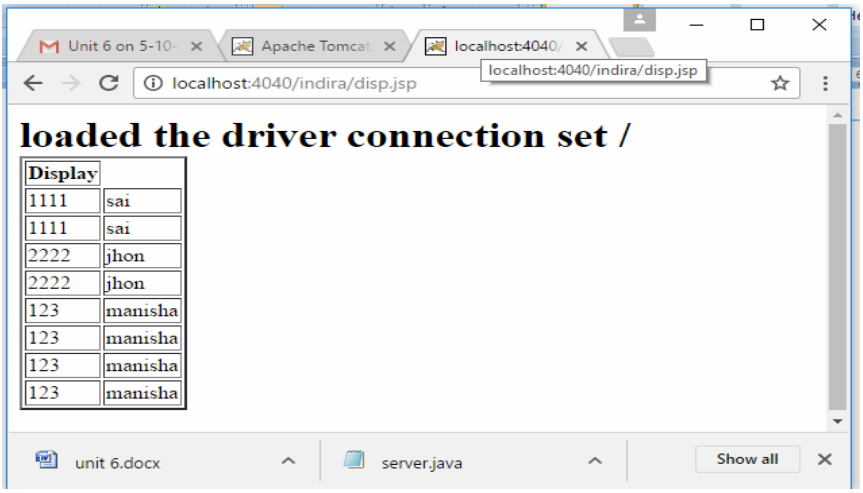
disp - Notepad
File Edit Format View Help

```
<%@ page import="java.sql.*"%>
<%
try
 {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            out.println("<h1>");

            out.println("loaded the driver");
            Connection c=DriverManager.getConnection
("jdbc:odbc:indira","system","oracle");
            out.println("connection set");
            Statement st=c.createStatement();
            ResultSet rs=st.executeQuery("select * from emp");
            out.println("<table border=2>");
           out.println("<th> Display </th>");
            while(rs.next())
            {
                        out.println("<tr>");
                        out.println("<td>"+rs.getString(1)+"</td>");
```

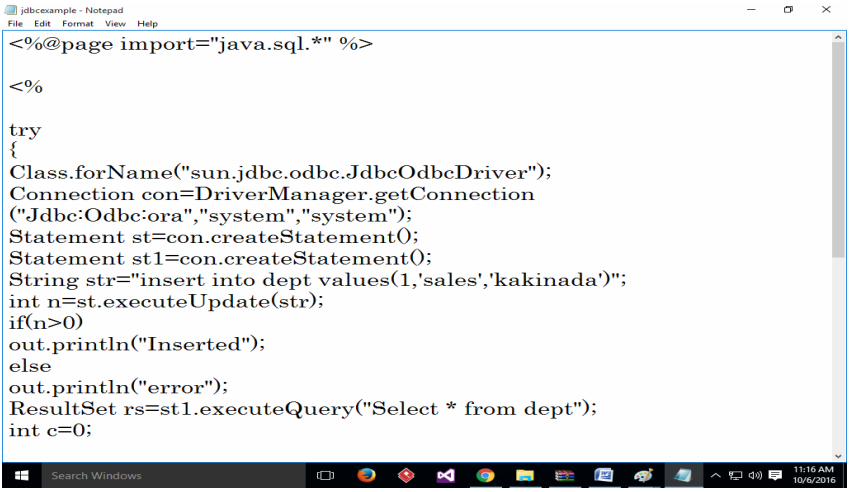
disp - Notepad
File Edit Format View Help

```
            Statement st=c.createStatement();
            ResultSet rs=st.executeQuery("select * from emp");
            out.println("<table border=2>");
           out.println("<th> Display </th>");
            while(rs.next())
            {
                        out.println("<tr>");
                        out.println("<td>"+rs.getString(1)+"</td>");
                        out.println("<td>"+rs.getString(2)+"</td>");
                        out.println("</tr>");
            }
            out.println("/<table>");
out.println("</h1>");
 }
 catch(Exception e)
 {
            out.println(e.toString());
 }
 %>
```

Output:



- **Write a JSP to Insert one record into dept table and Display them.**



jdbcexample - Notepad

File Edit Format View Help

```
ResultSet rs=st1.executeQuery("Select * from dept");
int c=0;
while(rs.next())
{
int t1=rs.getInt("deptid");
String t2=rs.getString("dname");
String t3=rs.getString("loc");
out.println(t1+t2+t3);
c++;
}
if(c==0)
out.println("table is empty");
else
out.println(c+"rows are selected");
con.close();
}
catch(Exception e){}

%>
```

Search Windows 11:16 AM 10/6/2016

WRITE A JSP TO STORE USER_ID AND PASSWORD and Display User.html

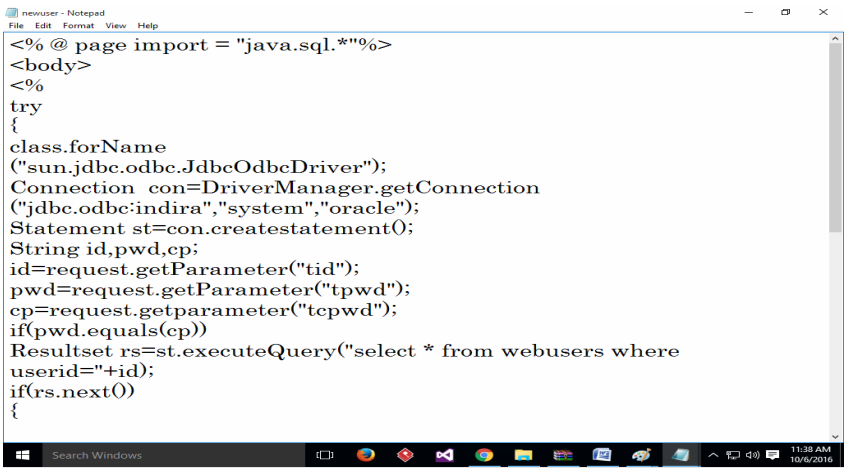
client - Notepad

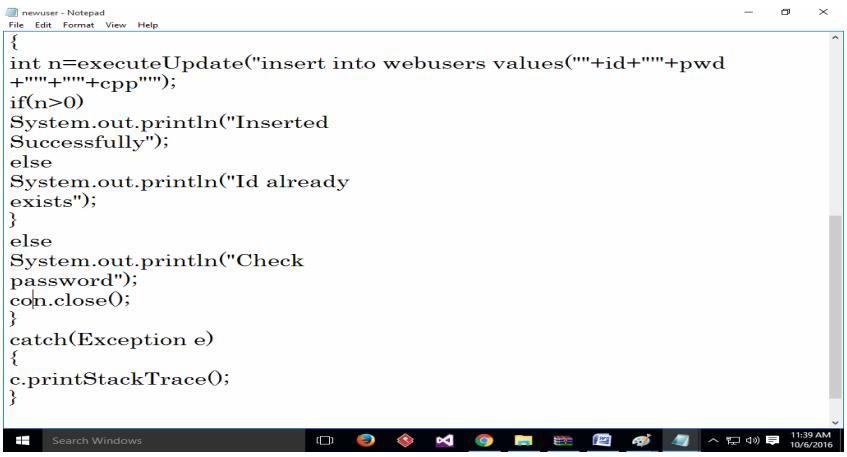
File Edit Format View Help

```
<center> WRITE A JSP TO STORE USER_ID AND
PASSWORD</center>

<body text = ref bgcolor=cyan>
<center>
<h1>
<h1>Registration</h1>
<form method = post action = "http://localhost:4040/newuser.jsp">
userid<input type = text name="tid">
pwd<input type = password name="tpwd">
confirm password<input type = password name="tcpwd">
<input type = submit value = "Register">
</form>
</h1>
</center>
</body>
```

Search Windows 11:21 AM 10/6/2016

Newuser.jsp


newuser - Notepad
File Edit Format View Help
<% @ page import = "java.sql.*"%>
<body>
<%
try
{
class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=DriverManager.getConnection
("jdbc.odbc:indira","system","oracle");
Statement st=con.createstatement();
String id,pwd,cp;
id=request.getParameter("tid");
pwd=request.getParameter("tpwd");
cp=request.getparameter("tcpwd");
if(pwd.equals(cp))
Resultset rs=st.executeQuery("select * from webusers where
userid="+id);
if(rs.next())
{
Search Windows
11:38 AM
10/6/2016


newuser - Notepad
File Edit Format View Help
{
int n=executeUpdate("insert into webusers values(""+id+"""+pwd
+"""+"""+cpp""");
if(n>0)
System.out.println("Inserted
Successfully");
else
System.out.println("Id already
exists");
}
else
System.out.println("Check
password");
con.close();
}
catch(Exception e)
{
c.printStackTrace();
}
Search Windows
11:39 AM
10/6/2016

```
+"""+"""+cpp""");
if(n>0)
System.out.println("Inserted
Successfully");
else
System.out.println("Id already
exists");
}
else
System.out.println("Check
password");
con.close();
}
catch(Exception e)
{
c.printStackTrace();
}
%>
</body>
```

Accessing Database using Type 4 Driver

```
<%@page import="java.sql.*" %>

<%
        String name=request.getParameter("t1");
        String pwd=request.getParameter("t2");

  try{
        Class.forName("oracle.jdbc.driver.OracleDriver");

        Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
                                                   "system","manager");

        Statement st=con.createStatement();

                int i=st.executeUpdate("insert into UserTab values("+name+","+pwd+")");

                if(i==1)
                {
                        out.println("<br> You have registerd Successfully");
                }
  }
  catch(Exception e)
  {
        out.println(e.toString());
  }

%>
```