# UNIT IV

**Introduction to JSP:** The Problem with Servlet. The Anatomy of a JSP Page, JSP Processing. JSP Application Design with MVC Setting Up and JSP Environment, JSP Declarations, Directives,

## JAVA SERVER PAGES

The Servlet technology and JavaServer Pages (JSP) are the two main technologies for developing java Web applications. When first introduced by Sun Microsystems in 1996, the Servlet technology was considered superior to the reigning Common Gateway Interface (CGI) because servlets stay in memory after they service the first requests. Subsequent requests for the same servlet do not require instantiation of the servlet‘s class therefore enabling better response time.

Servlets are Java classes that implement the javax.servlet.Servlet interface. They are compiled and deployed in the web server. The problem with servlets is that you embed HTML in Java code. If you want to modify the cosmetic look of the page or you want to modify the structure of the page, you have to change code. Generally speaking, this is left to the better hands (and brains) of a web page designer and not to a Java developer.

```
PrintWriter pw = response.getWriter(); pw.println("<html><head><title>Testing</title></head>");
pw.println("<body bgcolor=\"# ffdddd\">");
```

As seen from the example above this method presents several difficulties to the web developer:

1. The code for a servlet becomes difficult to understand for the programmer.
2. The HTML content of such a page is difficult if not impossible for a web designer to understand ordesign.
3. This is hard to program and even small changes in the presentation, such as the page‘s background color, will require the servlet to be recompiled. Any changes in the HTML content require the rebuilding of the whole servlet.
4. 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 which is time consuming, error prone, and extremely boring.
5. In many Java servlet-based applications, processing the request and generating the response are both handled by a single servletclass.
6. The servlet contains request processing and business logic (implemented by methods ), and also generates the response HTML code, are embedded directly in the servlet code.

JSP solves these problems by giving a way to include java code into an HTML page using scriptlets. This way the HTML code remains intact and easily accessible to web designers, but the page can sill perform its task.

In late 1999, Sun Microsystems added a new element to the collection of Enterprise Java tools: JavaServer Pages (JSP). JavaServer Pages are built on top of Java servlets and designed to increase the efficiency in which programmers, and even nonprogrammers, can create web content.

Instead of embedding HTML in the code, you place all static HTML in a JSP page, 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 the servlet, and the business logic can be handled by JavaBeans and EJBcomponents.

A JSP page is handled differently compared to a servlet by the web server. When a servlet is deployed into a web server in compiled (bytecode) form, then a JSP page is deployed in its original, human-readable form.
When a user requests the specific page, the web server compiles the page into a servlet and from there on handles it as a standard servlet.
This accounts for a small delay, when a JSP page is first requested, but any subsequent requests benefit from the same speed effects that are associated with servlets.

**The Problem with Servlet**

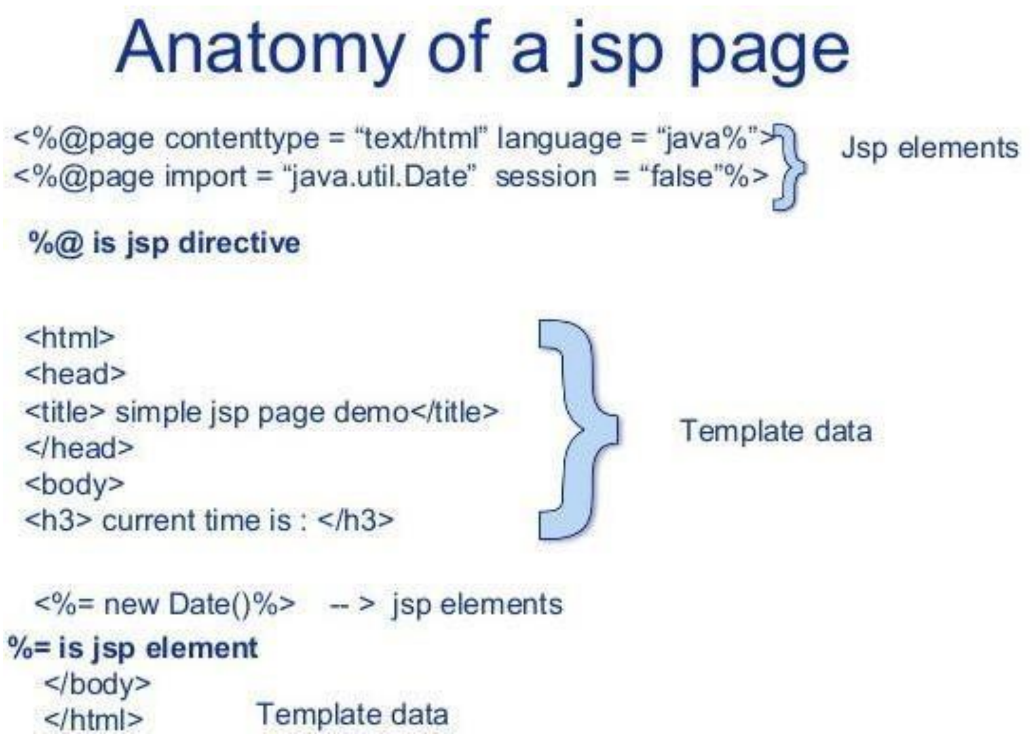
- Servlets are difficult to code which are overcome in JSP. Other way, we can say, JSP is almost a replacement of Servlets, (by large, the better word is extension of Servlets), where coding decreases more than half.
- In Servlets, both static code and dynamic code are put together. In JSP, they are separated. For example,In Servlets:

out.println(―Hello Mr.‖ + str + ‖you are great man‖);
where str is the name of the client which changes for each client and is known as dynamic content. The strings, ―Hello Mr.‖ and ―you are great man‖ are static content which is the same irrespective of client. In Servlets, in println(), both are puttogether.

- In JSP, the static content and dynamic content is separated. Static content is written in HTML and dynamic content in JSP. As much of the response comprises of static content (nearly 70%) only, the JSP file more looks as a HTML file.
- Programmer inserts, here and there, chunks of JSP code in a running HTML developed by Designer. As much of the response delivered to cleint by server comprises of static content (nearly 70%), the JSP file more looks like a HTML file. Other way we can say, JSP is nothting but Java in HTML (servlets are HTML
- in Java); java code embedded in HTML.
- When the roles of Designer and Programmer are nicely separated, the product development becomes cleaner and fast. Cost of developing Web site becomes cheaper as Designers are much paid less than Programmers, especially should be thought in the present competitive world.
- Both presentation layer and business logic layer put together in Servlets. In JSP, they can be separated with the usage of JavaBeans.
- The objects of PrintWriter, ServletConfig, ServletContext, HttpSession and RequestDispatcher etc. are created by the Programmer in Servlets and used. But in JSP, they are builtin and are known as "implicit objects". That is, in JSP, Programmer never creates these objects and straightaway use them as they are implicitly created and given by JSP container. This decreases lot of coding.
- JSP can easily be integrated with JavaBeans.
- JSP is much used in frameworks like Sturts etc.
- With JSP, Programmer can build custom tags that can be called in JavaBeans directly. Servlets do not have this advantage. Reusability increases with tag libraries and JavaBean etc.
- Writing alias name in <url-pattern> tag of web.xml is optional in JSP but mandatory in Servlets.
- A Servlet is simply a Java class with extension .java written in normal Javacode.
- A Servlet is a Java class. It is written like a normal Java. JSP is comes with some elements that are easy to write.

- JSP needs no compilation by the Programmer. Programmer deploys directly a JSP source code file in server where as incase of Servlets, the Programmer compiles manually a Servlet file and deploys a .class file in server.
- JSP is so easy even a Web Designer can put small interactive code (not knowing much of Java) in static Web pages.
- First time when JSP is called it is compiled to a Servlet. Subsequent calls to the same JSP will call the same compiled servlet (instead of converting the JSP to servlet), Ofcourse, the JSP code would have not modified. This increases performance.

## Anatomy of JSP



Anatomy of a jsp page

```
<%@page contenttype = "text/html" language = "java%">
<%@page import = "java.util.Date"  session  = "false"%>
```

%@ is jsp directive

```
<html>
<head>
<title> simple jsp page demo</title>
</head>
<body>
<h3> current time is : </h3>

 <%= new Date()%>    -- >  jsp elements
```

%= is jsp element

```
  </body>
  </html>
```

## JSP Processing

Once you have a JSP capable web-server or application server, you need to know the following information about it:

- Where to place the files
- How to access the files from your browser (with an http: prefix, not as file:)

You should be able to create a simple file, such as

```
<HTML>
<BODY>
Hello, world
</BODY> </HTML>
```

Know where to place this file and how to see it in your browser with an http:// prefix.

Since this step is different for each web-server, you would need to see the web-server documentation to find out how this is done. Once you have completed this step, proceed to the next.

**Your first JSP**

JSP simply puts Java inside HTML pages. You can take any existing HTML page and change its extension to ".jsp" instead of ".html". In fact, this is the perfect exercise for your first JSP. Take the HTML file you used in the previous exercise. Change its extension from ".html" to ".jsp". Now load the new file, with the ".jsp" extension, in your browser.

**You will see the same output, but it will take longer! But only the first time. If you reload it again, it will load normally.**

What is happening behind the scenes is that your JSP is being turned into a Java file, compiled and loaded. This compilation only happens once, so after the first load, the file doesn't take long to load anymore. (But everytime you change the JSP file, it will be re- compiled again.)

Of course, it is not very useful to just write HTML pages with a .jsp extension! We now proceed to see what makes JSP so useful

Adding dynamic content via expressions

As we saw in the previous section, any HTML file can be turned into a JSP file by changing its extension to .jsp. Of course, what makes JSP useful is the ability to embed Java. Put the following text in a file with .jsp extension (let us call it hello.jsp), place it in your JSP directory, and view it in a browser.

```
<HTML>
<BODY>
Hello! The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

Notice that each time you reload the page in the browser, it comes up with the current time. The character sequences

<%= and %> enclose Java expressions, which are evaluated at run time.

This is what makes it possible to use JSP to generate dyamic HTML pages that change in response to user actions or vary from user to user.

## Explain about JSP Elements

In this lesson we will learn about the various elements available in JSP with suitable examples. In JSP elements can be dividedinto 4 different types.

**These are:**

**1. Expressions**

We can use this tag to output any data on the generated page. These data are automatically converted to string and printed on the output stream.

Syntax of JSP Expressions are: <%="Any thing" %>

JSP Expressions start with Syntax of JSP Scriptles are with <%= and ends with %>. Between these this you can put anything and that will convert to the String and that will be displayed.

**Example:** <%="Hello World!" %> Above code will display 'Hello World!'

## 2. Scriplets

In this tag we can insert any amount of valid java code and these codes are placed in _jspService method by the JSP engine.

**Syntax of JSP Scriptles are:**

```
<% //java codes
%>
```

JSP Scriptlets begins with <% and ends %> .We can embed any amount of java code in the JSP Scriptlets. JSP Engine places these code in the _jspService() method. Variables available to the JSP Scriptlets are:

a. **Request:** Request represents the clients request and is a subclass of HttpServletRequest. Use this variable to retrieve the data submitted along the request.

Example:

```
<% //java codes
String userName=null; userName=request.getParameter("userName");
%>
```

b. **Response:** Response represents the server response and is a subclass of HttpServletResponse.

```
<% response.setContentType("text/html"); %>
```

c. **Session:** represents the HTTP session object associated with the request. Your Session ID: <%= session.getId() %>

d. **Out:** out is an object of output stream and is used to send any output to the client.

## 3. Directives

A JSP "directive" starts with <%@ characters. In the directives we can import packages, define error handling pages or the session information of the JSP page.

**Syntax of JSP directives is:**

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

a. **page:** page is used to provide the information about it. Example: <%@page language="java" %>

b. **include:** include is used to include a file in the JSP page. Example: <%@ include file="/header.jsp" %>

c. **taglib:** taglib is used to use the custom tags in the JSP pages (custom tags allows us to defined our own tags). Example: <%@ taglib uri="tlds/taglib.tld" prefix="mytag" %>

Page tag attributes are:

a. **language="java"**

This tells the server that the page is using the java language. Current JSP specification supports only java language. Example: <%@page language="java" %>

b. **extends="mypackage.myclass"**

This attribute is used when we want to extend any class. We can use comma(,) to import more than one packages. Example: %@page language="java" import="java.sql.*" %

c. **session="true"**

When this value is true session data is available to the JSP page otherwise not. By default this value is true.

Example: <%@page language="java" session="true" %>

**d. errorPage="error.jsp"**

errorPage is used to handle the un-handled exceptions in the page.
Example: <%@page session="true" errorPage="error.jsp" %>

**e. contentType="text/html;charset=ISO-8859-1"**

Use this attribute to set the mime type and character set of the
JSP. Example: <%@page contentType="text/html;charset=ISO-8859-1"%>

**4. Declarations**

T h i s   tag   is   u s e d   f o r   defini ng   t h e   functio ns   a n d   variab les
to be used in theJSP. Syntax of JSP Declaratives are:

```
<%!
//java codes
%>
```

JSP Declaratives begins with <%! and ends %> with .We can embed any amount of java code in the JSP Declaratives. Variables and functions defined in the declaratives are class level and can be used anywhere in the JSP page.

**Example**

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%!
Date theDate = new Date(); Date getDate()
{
System.out.println( "In getDate() method" ); return theDate;
}
%>
Hello! The time is now <%= getDate() %>
</BODY>
</HTML>
```

**Expalin about Jsp programs?**

A Web Page with JSP code

```
<HTML>
<HEAD>
<TITLE>A Web Page</TITLE>
</HEAD>
<BODY>
<% out.println("Hello there!"); %>
</BODY>
</HTML>
```

**Using a Literal**

```
<HTML>
<HEAD>
<TITLE>Using a Literal</TITLE>
</HEAD>
<BODY>
<H1>Using a Literal</H1>
<%
out.println("Number of days = "); out.println(365);
%>
</BODY>
</html>
```

**Declaration Tag Example**

```
<%!
String name = "Joe";
String date = "8th April, 2002";
%>
<HTML>
<TITLE>Declaration Tag Example</TITLE>
<BODY>
This page was last modified on <%= date %> by <%= name %>.
</BODY>
</HTML>
```

**Embedding Code**

```
<%!
String[] names = {"A", "B", "C", "D"};
%>
<HTML>
<HEAD><TITLE>Embedding Code</TITLE></HEAD>
<BODY>
<H1>List of people</H1>
<TABLE BORDER="1">
<TH>Name</TH>
<% for (int i=0; i<names.length; i++) { %>
<TR><TD><%=names[i]%></TD></TR>
<% } %>
</TABLE>
</BODY>
</HTML>
```

**Use out**

```
<%@ page language="java" %>
<HTML>
<HEAD><TITLE>JSP Example</TITLE></HEAD>
<BODY>
<H1>Quadratic Equation: y = x^2</H1>
<TABLE BORDER="1">
<TH>x</TH><TH>y</TH>
<%
for (int i=0; i<10; i++)
out.print("<TR><TD WIDTH='100'>" + i + "</TD><TD WIDTH='100'>" + (i*i) +
"</TD></TR>");
%>
</TABLE>
</BODY>
</HTML>
```

Casting to a New Type

```
<HTML>
<HEAD>
<TITLE>Casting to a New Type</TITLE>
</HEAD>
<BODY>
<H1>Casting to a New Type</H1>
<%
float float1;
double double1 = 1; float1 = (float) double1;

out.println("float1 = " + float1);
%>
</BODY>
</HTML>
```

**Creating a String**

```
<HTML>
<HEAD>
<TITLE>Creating a String</TITLE>
</HEAD>

<BODY>
<H1>Creating a String</H1>
```

```
<%
String greeting = "Hello from JSP!"; out.println(greeting);
%>
</BODY>
</HTML>
```

**Use for loop to display string array**

```
<%@ page session="false" %>
<%
String[] colors = {"red", "green", "blue"};
for (int i = 0; i < colors.length; i++) { out.print("<P>" + colors[i] + "</p>");
}
%>
```

**Creating an Array**

```
<HTML>
<HEAD>
<TITLE>Creating an Array</TITLE>
</HEAD>
<BODY>
<H1>Creating an Array</H1>
<%
double accounts[];
accounts = new double[100]; accounts[3] = 119.63; out.println("Account 3 holds $" + accounts[3]);
%>
</BODY>
</HTML>
```

**Using Multidimensional Arrays**

```
<HTML>
<HEAD>
<TITLE>Using Multidimensional Arrays</TITLE>
</HEAD>
<BODY>
<H1>Using Multidimensional Arrays</H1>
<%
double accounts[][] = new double[2][100]; accounts[0][3] = 119.63;
accounts[1][3] = 194.07;
```

```
out.println("Savings Account 3 holds $" + accounts[0][3] + "<BR>"); out.println("Checking
Account 3 holds $" + accounts[1][3]);
%>
</BODY>
</HTML>
```

**Finding a Factorial**

```
<HTML>
<HEAD>
<TITLE>Finding a Factorial</TITLE>
</HEAD>
<BODY>
<H1>Finding a Factorial</H1>
<%
int value = 6, factorial = 1, temporaryValue = value;
while (temporaryValue > 0) { factorial *= temporaryValue; temporaryValue--;
}
out.println("The factorial of " + value + " is " + factorial + ".");
%>
</BODY>
</HTML>
```

**Get Form Button Value**

```
<HTML>
<HEAD>
<TITLE>Using Buttons</TITLE>
</HEAD>
<BODY>
<H1>Using Buttons</H1>
<FORM NAME="form1" ACTION="basic.jsp" METHOD="POST">
<INPUT TYPE="HIDDEN" NAME="buttonName">
<INPUT TYPE="BUTTON" VALUE="Button 1" ONCLICK="button1()">
<INPUT TYPE="BUTTON" VALUE="Button 2" ONCLICK="button2()">
<INPUT TYPE="BUTTON" VALUE="Button 3" ONCLICK="button3()">
</FORM>
<SCRIPT LANGUAGE="JavaScript">
<!--
function button1()
{
document.form1.buttonName.value = "button 1" form1.submit()
}
```

```
function button2()
{
document.form1.buttonName.value = "button 2" form1.submit()
}
function button3()
{
document.form1.buttonName.value = "button 3" form1.submit()
}
// -->
</SCRIPT>
</BODY>
</HTML>
```

**basic.jsp**

```
<HTML>
<HEAD>
<TITLE>Determining Which Button Was Clicked</TITLE>
</HEAD>
<BODY>
<H1>Determining Which Button Was Clicked</H1> You clicked
<%= request.getParameter("buttonName") %>
</BODY>
</HTML>
```

**Read Form Checkboxes index.jsp**

```
<HTML>
<HEAD>
<TITLE>Submitting Check Boxes</TITLE>
</HEAD>
<BODY>
<H1>Submitting Check Boxes</H1>
<FORM ACTION="basic.jsp" METHOD="post">
<INPUT TYPE="CHECKBOX" NAME="check1" VALUE="check1" CHECKED>
Checkbox 1
<BR>
<INPUT TYPE="CHECKBOX" NAME="check2" VALUE="check2">
Checkbox 2
<BR>
<INPUT TYPE="CHECKBOX" NAME="check3" VALUE="check3">
Checkbox 3
```

```
<BR>
<INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

**basic.jsp**

```
<HTML>
<HEAD>
<TITLE>Reading Checkboxes</TITLE>
</HEAD>
<BODY>
<H1>Reading Checkboxes</H1>
<%
if(request.getParameter("check1") != null) { out.println("Checkbox 1 was checked.<BR>");
}
else {
out.println("Checkbox 1 was not checked.<BR>");
}
if(request.getParameter("check2") != null) { out.println("Checkbox 2 was checked.<BR>");
}
else {
out.println("Checkbox 2 was not checked.<BR>");
}
if(request.getParameter("check3") != null) { out.println("Checkbox 3 was checked.<BR>");
}
else {
out.println("Checkbox 3 was not checked.<BR>");
}
%>
</BODY>
</HTML>
```

**Model View Controller**

JSP technology can play a part in everything from the simplest web application to complex enterprise applications. How large a part JSP plays differs in each case, of course. Let introduce a design model called Model- View-Controller (MVC), suitable for both simple and complex applications.

MVC was first described by Xerox in a number of papers published in the late 1980s. The key point of using MVC is to separate logic into three distinct units: the Model, the View, and the Controller. In a server application, we commonly classify the parts of the application as business logic, presentation, and request processing.

Business logic is the term used for the manipulation of an application's data, such as customer, product, and order information. Presentation refers to how the application data is displayed to the user, for example, position, font, and size. And finally, request processing is what ties the business logic and presentation parts together.

In MVC terms, presentation should be separated from the business logic. Presentation of that data (the View) changes fairly often. Just look at all the face- lifts many web sites go through to keep up with the latest fashion in web design. Some sites may want to present the data in different languages or present different subsets of the data to internal and external users.

**cookies:**

- A **cookie** is a small piece of information created by a JSP program that is stored in the client‘s hard disk by the browser. Cookies are used to store various kind of information such as username, password, and user preferences, etc.
- **Different methods in cookie class are:**
    1. **String getName()**- Returns a name of cookie
    2. **String getValue()**-Returns a value of cookie
    3. **int getMaxAge()**-Returns a maximum age of cookie in millisecond
    4. **String getDomain(**)-Returns a domain
    5. **boolean getSecure()**-Returns true if cookie is secure otherwise false
    6. **String getPath()**-Returns a path ofcookie
    7. **void setPath(Sting)**- set the path of cookie
    8. **void setDomain(String)**-set the domain of cookie **9.void setMaxAge(int)**-set the maximum age of cookie **10**.**void setSecure(Boolean)**-set the secure of cookie.

**Creating cookie:**
Cookie are created using cookie class constructor.
Content of cookies are added the browser using addCookies() method.

**Reading cookies:**
Reading the cookie information from the browser using getCookies() method. Find the length of cookie class.
Retrive the information using different method belongs the cookie class

**PROGRAM: To create and read the cookie for the given cookie name as "EMPID" and its value as"AN2356".**

**JSP program to create a cookie**

```
<%!
Cookie c=new Cookie(–EMPID‖,‖AN2356‖); response.addCookie(c);
%>
```

**JSP program to read a cookie**

```
<%!
Cookie c[]=request.getCookies(); for(i=0;i<c.length;i++)
{
String name=c[i].getName(); String value=c[i].getValue(); out.println(–name=‖+name);
out.println(–value=‖+value);
}
%>
```

**Session object(session tracking or session uses)**

- The HttpSession object associated to the request
- Session object has a session scope that is an instance of javax.servlet.http.HttpSession class. Perhaps it is the most commonly used object to manage the state contexts.
- This object persist information across multiple user connection. Created automatically by
- Different methods of HttpSession interface are as follows:

1. **object getAttribute(String)-**Returns the value associated with the name passed as argument.
2. **long getCreationTime()-**Returns the time when session created. 3.**String getID()-**Returns the session ID
4. **long getAccessedTIme()-**returns the time when client last made a request for this session.
5. **void setAttribute(String,object)-**Associates the values passed in the object name passed.

**Program:**

```
<%!

HttpSession h=req.getSesssion(true); Date d=(Date) h.getAttribute("Date"); out.println(-last date and time‖+d);

Date d1=new Date(); d1=h.setAttribute("date",d1); out.println(-current date and time=‖+d1);

%>
```

**Database Access:** Database Programming using JDBC, JDBC drivers, Studying Javax.sql.* package, Connecting to database in PHP, Execute Simple Queries, Accessing a Database from a Servlet and JSP

**What is JDBC Driver?**

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The Java.sql package that ships with JDK, contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver.

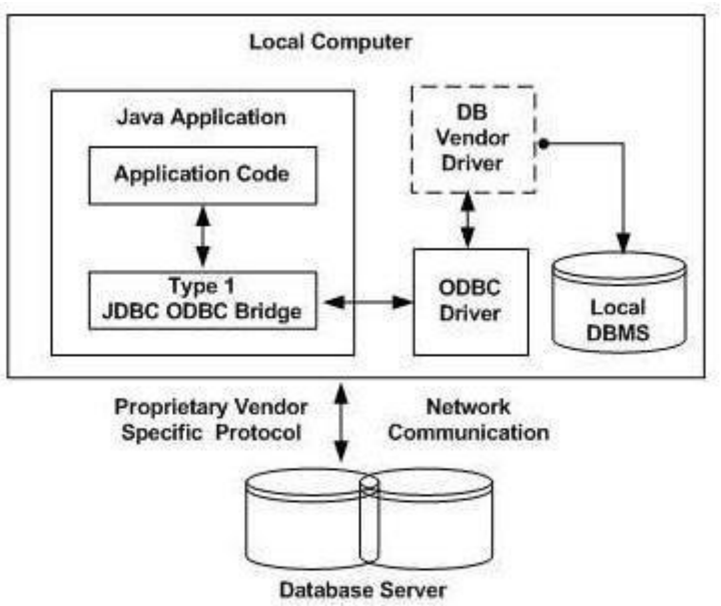
**JDBC Drivers Types**

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is exp lained below −

**Type 1: JDBC-ODBC Bridge Driver**

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

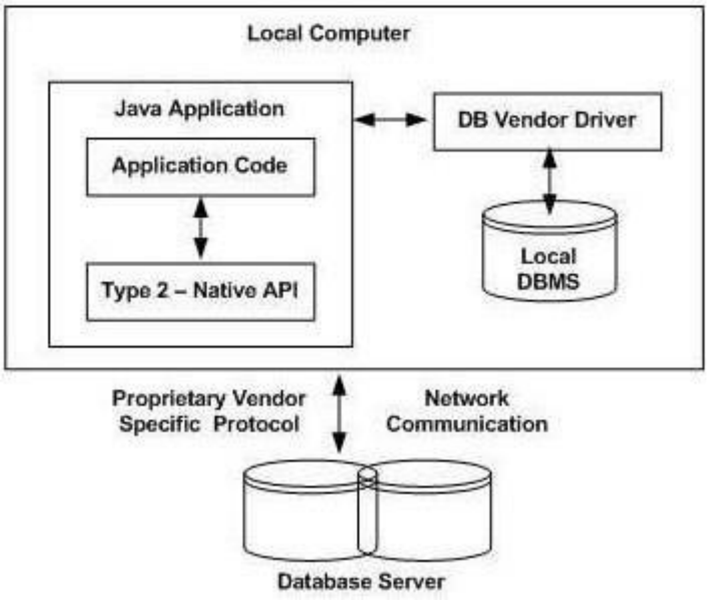


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

**Type 2: JDBC-Native API**

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

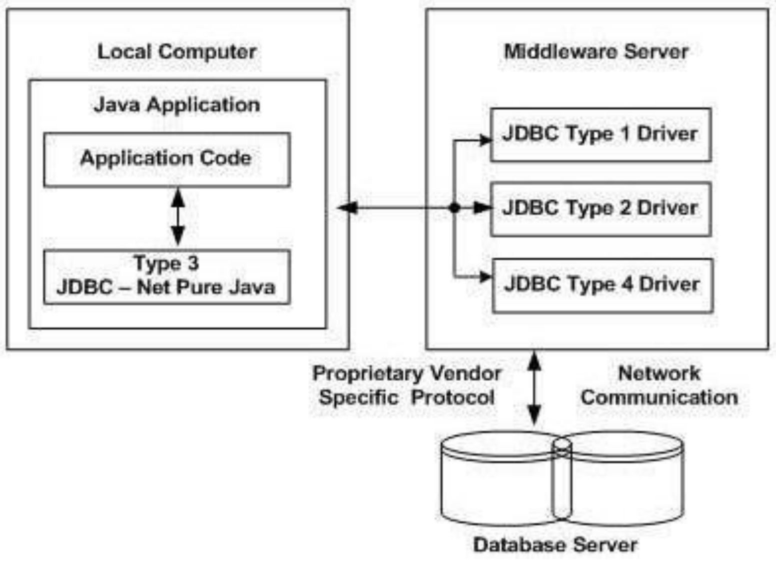


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

**Type 3: JDBC-Net pure Java**

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



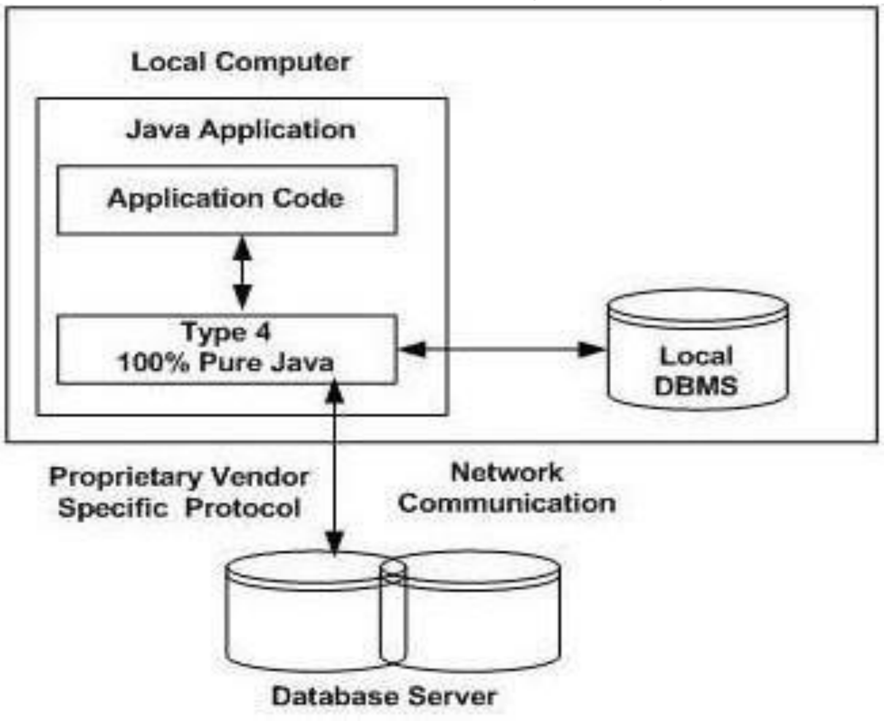
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

**Type 4: 100% Pure Java**

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

**Which Driver should be Used?**

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment- level driver, and is typically used for development and testing purposes only.

**JDBC( Java Database Connectivity):**

The first thing you need to do is check that you are set up properly. This involves the following steps:

1. **Install Java and JDBC on yourmachine.**

To install both the Java tm platform and the JDBC API, simply follow the instructions for downloading the latest release of the JDK tm (Java Development Kit tm ). When you download the JDK, you will get JDBC as well.

2. **Install a driver on yourmachine.**

Your driver should include instructions for installing it. For JDBC drivers written for specific DBMSs, installation consists of just copying the driver onto your machine; there is no special configuration needed.

The JDBC-ODBC Bridge driver is not quite as easy to set up. If you download JDK, you will automatically get the JDBC-ODBC Bridge driver, which does not itself require any special configuration. ODBC, however, does. If you do not already have ODBC on your machine, you will need to see your ODBC driver vendor for information on installation and configuration.

3. **Install your DBMS ifneeded.**

If you do not already have a DBMS installed, you will need to follow the vendor's instructions for installation. Most users will have a DBMS installed and will be working with an established database.

**Configuring Database:**

Configuring a database is not at all difficult, but it requires special permissions and is normally done by a database administrator.

First, open the control panel. You might find "Administrative tools" select it, again you may find shortcut for "Data Sources (ODBC)". When you open the ―Data Source (ODBC)" 32bit ODBC‖ icon, you‗ll see a "ODBC Data Source Administrator" dialog window with a number of tabs, including ―User DSN,‖ ―System DSN,‖ ―File DSN,‖ etc., in which ―DSN‖ means ―Data Source Name.‖ Select ―System DSN,‖. and add a new entry there, Select appropriate driver for the data source or directory where database lives. You can name the entry anything you want, assume here we are giving our data source name as "MySource".

**JDBC Database Access**

JDBC was designed to keep simple things simple. This means that the JDBC API makes everyday database tasks, like simple SELECT statements, very easy.

**Import a package java.sql.* :** This package provides you set of all classes that enables a network interface between the front end and back end database.

•DriverManager will create a Connectionobject.

•java.sql.Connection interface represents a connection with a specific database. Methods of connection is close(), creatStatement(), prepareStatement(), commit(), close() and prepareCall()

•Statement interface used to interact with database via the execution of SQL statements. Methods of this interface are executeQuery(), executeUpdate(), execute() and getResultSet().

•A ResultSet is returned when you execute an SQL statement. It maintains a pointer to a row within the tablur results. Mehods of this interface are next(), getBoolean(), getByte(), getDouble(), getString() close() and getInt().

## Establishing a Connection

The first thing you need to do is establish a connection with the DBMS you want to use. This involves two
steps: (1) loading the driver and (2) making the connection.

**Loading Drivers:** Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Your driver documentation will give you the class name to use. For instance, if the class name is jdbc.DriverXYZ , you would load the driver with the following line of code:

```
Class.forName("jdbc.DriverXYZ");
```

**Making the Connection:** The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

```
Connection con = DriverManager.getConnection(url,"myLogin", "myPassword");
```

If you are using the JDBC-ODBC Bridge driver, the JDBC URL will start with jdbc:odbc: . The rest of the URL is generally your data source name or database system. So, if you are using ODBC to access an ODBC data source called "MySource, " for example, your JDBC URL could be jdbc:odbc:MySource . In p lace of " myLogin " you put the name you use to log in to the DBMS; in place of " myPassword " you put your password for the DBMS. So if you log in to your DBMS with a login name of " scott " and a password of "tiger" just these two lines of code will establish a connection:

```
String url = "jdbc:odbc:MySource";
Connection con = DriverManager.getConnection(url, "scott", "tiger");
```

The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS. In the previous example, con is an open connection, and we will use it in the dorth coming examples.

## Creating JDBC Statements

A Statement object is what sends your SQL statement to the DBMS. You simply create a Statement object and then execute it, supplying the appropriate execute method with the SQL statement you want to send. For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate .

It takes an instance of an active connection to create a Statement object. Inthe following example, we use our Connection object con to create the Statement object stmt :

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

At this point stmt exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute stmt .

For example, in the following code fragment, we supply executeUpdate with the SQL statement from the example above:

```
stmt.executeUpdate("CREATE TABLE STUDENT " +
"(S_NAME VARCHAR(32), S_ID INTEGER, COURSE VARCHAR2(10), YEAR
VARCHAR2(3)‖);
```

Since the SQL statement will not quite fit on one line on the page, we have split it into two strings concatenated by a plus sign (+) so that it will compile. Executing Statements.

Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. The method executeUpdate is also used to execute SQL statements that update a table. In practice, executeUpdate is used far more often to update tables than it is to create them because a table is created once but may be updated many times.

The method used most often for executing SQL statements is executeQuery . This method is used to execute SELECT statements, which comprise the vast majority of SQL statements.

**Entering Data into a Table**

We have shown how to create the table STUDENT by specifying the names of the columns and the data types to be stored in those columns, but this only sets up the structure of the table. The table does not yet contain any data. We will enter our data into the table one row at a time, supplying the information to be stored in each column of that row. Note that the values to be inserted into the columns are listed in the same order that the columns were declared when the table was created, which is the default order.

The following code inserts one row of data, Statement stmt = con.createStatement();

```
stmt.executeUpdate( "INSERT INTO STUDENT VALUES ('xStudent', 501, ‗
B.Tech‘,‘IV‘)");
```

Note that we use single quotation marks around the student name because it is nested within double quotation marks. For most DBMSs, the general rule is to alternate double quotation marks and single quotation marks to indicate nesting.

The code that follows inserts a second row into the table STUDENT . Note that we can just reuse the Statement object stmt rather than having to create a new one for each execution.

```
stmt.executeUpdate("INSERT INTO STUDENT " + "VALUES ('yStudent', 502,
‗B.Tech‘.‘III‘)");
```

**Getting Data from a Table**

Now that the table STUDENT has values in it, we can write a SELECT statement to access those values. The star (*) in the following SQL statement indicates that all columns should be selected. Since there is no WHERE clause to narrow down the rows from which to select, the following SQL statement selects the whole table:

```
SQL> SELECT * FROM STUDENT;
```

**Retrieving Values from Result Sets**

We now show how you send the above SELECT statements from a program written in the Java programming language and how you get the results we showed.

JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results. The following code demonstrates declaring the ResultSet object rs and assigning the results of our earlier queryto it:

```
ResultSet rs = stmt.executeQuery( "SELECT S_NAME, YEAR FROM STUDENT");
```

The following code accesses the values stored in the current row of rs. Each time the method next is invoked, the next row becomes the current row, and the loop continues until there are no more rows in rs.

```
String query = "SELECT COF_NAME, PRICE FROM STUDENT";
ResultSet rs = stmt.executeQuery(query); while (rs.next())
{
String s = rs.getString("S_N AME"); Integer i = rs.getInt("S_ID");
String c = rs.getString("COURSE"); String y = rs.getString(–YEAR‖);
System.out.println(i + " " + s + " " + c + " " + y);
}
```

**Updating Tables**

Suppose that after a period of time we want update the YEAR column in the table STUDENT. The SQL statement to update one row might look like this:

```
String updateString = "UPDATE STUDENT " +
"SET YEAR = IV WHERE S-NAME LIKE 'yStudent'";
```

Using the Statement object stmt , this JDBC code executes the SQL statement contained in updateString :

```
stmt.executeUpdate(updateString);
```

**Using try and catch Blocks:**

Something else all the sample applications include is try and catch blocks. These are the Java programming language's mechanism for handling exceptions. Java requires that when a method throws an exception, there be some mechanism to handle it. Generally a catch block will catch the exception and specify what happens (which you may choose to be nothing). In the sample code, we use two try blocks and two catch blocks. The first try block contains the method Class.forName, from the java.lang package. This method throws a ClassNotFoundException, so the catch block immediately following it deals with that exception. The second try block contains JDBC methods, which all throw SQLExceptions, so one catch block at the end of the application can handle all of the rest of the exceptions that might be thrown because they will all be SQLException objects.

**Retrieving Exceptions**

JDBC lets you see the warnings and exceptions generated by your DBMS and by the Java compiler. To see exceptions, you can have a catch block print them out. For example, the following two catch blocks from the sample code print out a message explaining the exception:

```
Try
{
// Code that could generate an exception goes here.
// If an exception is generated, the catch block below
// will print out information about it.
} catch(SQLException ex)
{
System.err.println("SQ LException: " + ex.getMessage());
}
```

## JavaBeans

**JavaBe
ans:**

JavaBeans is architecture for both using and building components in Java. This architecture supports the features of software reuse, component models, and object orientation. One of the most important features of JavaBeans is that it does not alter the existing Java language.

Although Beans are intended to work in a visual application development tool, they don't necessarily have a visual representation at run-time (although many will). What this does mean is that Beans must allow their property values to be changed through some type of visual interface, and their methods and events should be exposed so that the development tool can write code capable of manipulating the component when the application is executed.

**Bean Development Kit (BDK)** is a tool for testing whether your JavaBeans meets the JavaBean specification.

### Features of JavaBeans

**Compact and Easy:** JavaBeans components are simple to create and easy to use. This is an important goal of the JavaBeans architecture. It doesn't take very much to write a simple Bean, and such a Bean is lightweight, it doesn't have to carry around a lot of inherited baggage just to support the Beans environment.

**Portable:** Since JavaBeans components are built purely in Java, they are fully portable to any platform that supports the Java run-time environment. All platform specifics, as well as support for JavaBeans, are implemented by the Java virtual machine.

**Introspection:** Introspection is the process of exposing the properties, methods, and events that a JavaBean component supports. This process is used at run-time, as well as by a visual development tool at design-time. The default behavior of this process allows for the automatic introspection of any Bean. A low- level reflection mechanism is used to analyze the Bean's class to determine its methods. Next it applies some simple design patterns to determine the properties and events that are supported. To take advantage of reflection, you only need to follow a coding style that matches the design pattern. This is an important feature of JavaBeans. It means that you don't have to do anything more than code your methods using a simple convention. If you do, your Beans will automatically support introspection without you having to write any extra code.

**Customization:** When you are using a visual development tool to assemble components into applications, you will be presented with some sort of user interface for customizing Bean attributes. These attributes may affect the way the Bean operates or the way it looks on the screen. The application tool you use will be able to determine the properties that a Bean supports and build a property sheet dynamically. This property sheet will contain editors for each of the properties supported by the Bean, which you can use to customize the Bean to your liking. The Beans class library comes with a number of property editors for common types such as float, boolean, and String. If you are using custom classes for properties, you will have to create custom property editors to associate with them.

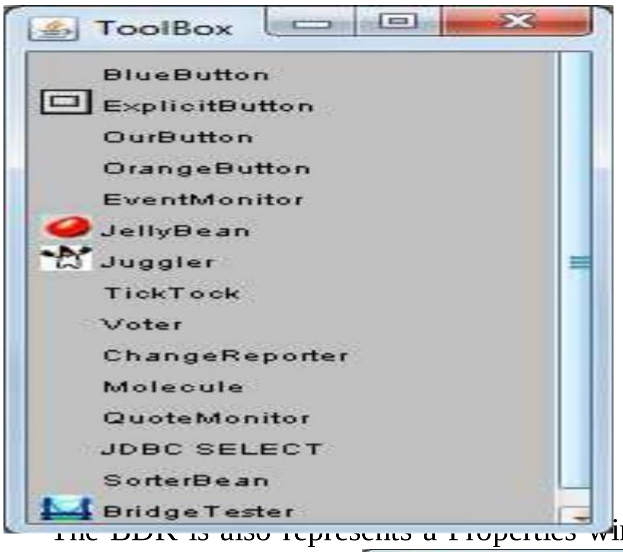
**Persistence:** It is necessary that Beans support a large variety of storage mechanisms. This way, Beans can participate in the largest number of applications. The simplest way to support persistence is to take advantage of Java Object Serialization. This is an automatic mechanism for saving and restoring the state of an object. Java Object Serialization is the best way to make sure that your Beans are fully portable, because you take advantage of a standard feature supported by the core Java platform. This, however, is not always desirable. There may be cases where you want your Bean to use other file formats or mechanisms to save and restore state. In the future, JavaBeans will support an alternative externalization mechanism that will allow the Bean to have complete control of its persistence mechanism.

**BDK (Bean Development Kit:**

The Bean Development kit is a tool that allows the user to configure and interconnect a set of beans. The user can change the properties of a Bean, link two or more Beans and execute Beans.

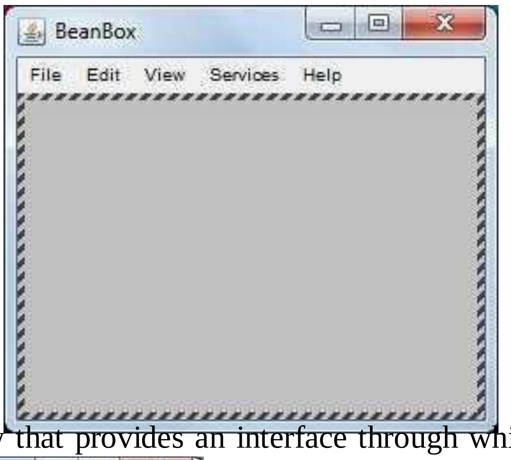
Start the "beanbox" by running "$bdk \beanbox\run.bat".

The Bean Development K it (BDK) represents a Toolbox, a Bean Box and a Property Window. The Toolbox window that lists the demonstration Beans.

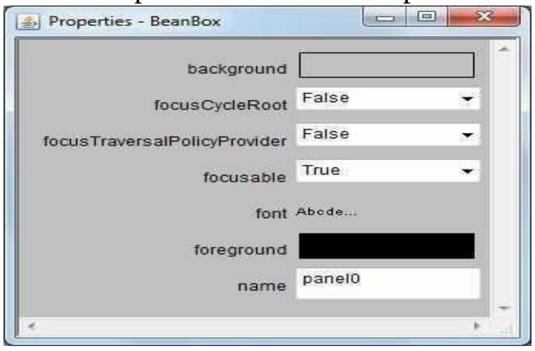


BlueButton
ExplicitButton
OurButton
OrangeButton
EventMonitor
JellyBean
Juggler
TickTock
Voter
ChangeReporter
Molecule
QuoteMonitor
JDBC SELECT
SorterBean
BridgeTester

The BeanBox window that provides an area in which the user can connect the components.



The BDK is also represents a Properties window that provides an interface through which the user configure a Bean.



background
focusCycleRoot False
focusTraversalPolicyProvider False
focusable True
font Abcde...
foreground
name panel0

**Deploying java beans in jsp:**

A JavaBean can be defined as a reusable software component. A Java Bean is a java class that should follow following conventions:

•It should have a no-arg constructor.

•It should be Serializable.

•It should provide methods to set and get the values of the properties, known as getter and setter methods.

Java beans- development phases

The Construction Phase

The Build Phase

The Execution Phase

JavaBeans component design conventions govern the properties of the class, and the public methods that give access to the properties.

**A JavaBeans component property can be:**

Read/write, read-only, or write-only.

It means it contains a single value, or indexed, i.e. it represents an array of values.

There is no requirement that a property be implemented by an instance variable; the property must simply be accessible using public methods that conform to certain conventions:

For each readable property, the bean must have a method of the form: PropertyClass getProperty () { ... }

For each writable property, the bean must have a method of the form: setProperty (PropertyClass pc) { ... }

In addition to the property methods, a JavaBeans component must define a constructor that takes no parameters.

## Steps to deploy and run this JSP using JavaBean Project

Write a java file and name it as **FindAuthor.java**
Write a jsp file and name it as **GetAuthorName.jsp**
Write a html file and name it as **WelcomePage.html**