Introduction JavaScript

Script means small piece of code. Scripting languages are two kinds one is client-side other one is servers-side scripting. In general client-side scripting is used for verifying simple validation at client side, server-side scripting is used for database verifications. VBScript, java script and J script are examples for client-side scripting and ASP, JSP, Servlets,PHP etc. are examples of server-side scripting.

JavaScript (originally known as "LiveScript") is a scripting language that runs inside the browser to manipulate and enhance the contents of Web pages. Java Script is designed to add interactivity to HTML pages. W eb pages are two types

5. Static web page
6. Dynamic webpage

- Static web page where there is no specific interaction with the client
- Dynamic web page which is having interactions with client and as well as validations can be added.

Simple HTML script is called static web page, if you add script to HTML page it is called dynamic page. Netscape navigator developed java script. Microsoft's version of JavaScript is Jscript.

- Java script code as written between <script>-----</script>tags
- All java script statements end with a semicolon
- Java script ignores whitespace
- Java script is case sensitivelanguage
- Script program can save as either. Js or.html

Benefits of JavaScript

- It is widely supported by webbrowsers;
- It gives easy access to the document objects and can manipulate most of them.
- Java Script gives interesting animations with long download times associated with many multimedia data types;
- Web surfers don't need a special plug-in to use your scripts
- Java Script relatively secure - you can't get a virus infection directly from Java Script.
- JavaScript code resembles the code of C Language; the syntax of both the language is very close to each other. The set of tokens and constructs are same in both the language.

Problems with JavaScript

- Most scripts relay upon manipulating the elements of DOM;
- Your script does not work then your page is useless
- Because of the problems of broken scripts many web surfers disable java script support in their browsers
- Script can run slowly and complex scripts can take long time to start up

Similarities between java script and java:

1. Both java script and java having same kind of operators
2. Java script uses similar control structures of java
3. Nowadays both are used as languages for use on internet.
4. Labeled break and labeled continue both are similar
5. Both are case-sensitivelanguages

Difference between java script and java:

Java Script	Java
Java Script is scripting language	Java is a programming language
Java Script is object-based programming language.	Java is object –oriented programming language
Java script code is not compiled, only	Java is compiled as well as interpreted

interpreted.	language
Java Script is Weekly-typed language	Java is Strongly-typed language
JavaScript code is run on abrowser only.	**Java** creates applications that run in a virtual machine or browser

The syntax of the script tag is asfollows:

```
<script language=""scripting language name"">
----------------------------------------
</script>
```

The language attribute specifies the scripting language used in the script. Both Microsoft internet explorer and Netscape navigator use java script as the default scripting language. The script tag may be placed in either the head or the body or the body of an HTML document.

```
Ex: <script language=""javascript"">
        ----------
        ----------
        ----------
</script>
```

Comments in JavaScript:

Single line comment- //

Multi-line comment- <!-- comment -->

Operators in JavaScript:

- Arithmetic operators(+,-,*,/,%)
- Relational operators(<,>,!=,<=,>=)
- Logical operators(&&,||,!)
- Assignment operator(=)
- Increment decrement operators(++,--)
- Conditional/Ternary operator(?:)
- Bitwise operators(&,|,!)

Control structures:

- If statement
- Switch
- While
- Do-while
- For
- Break
- Continue

Control structures syntax and working as same as java language.

Variables

Variables are like storage units/place holders to hold values. A variable is a memory location to hold certain different types of data. In Javascript, A variable can store all kinds of data. It is important to know the proper syntax to which variables must conform:

- They must start with a letter or underscore ("_")
- Subsequent characters can also be digits (0-9) or letters (A-Z and/or a-z). Remember, JavaScript is case-sensitive. (That means that MyVar and myVar are two different names to JavaScript, because they have different capitalization.)
- You cannot use reserved words as variable names.
- You cannot use spaces in names.
- Names are case-sensitive.

Syntax:

var v_name = value;

Examples of legal variable names are fname, temp99, and _name.

When you declare a variable by assignment outside of a function, it is called global variable, because it is available everywhere in the document. When you declare a variable within a function, it is called local variable, because it is available only within the function. To assign a value to a variable, you use the following notation:

```
var num = 8; var
real= 4.5;
var myString = "Web Technologies";
```

Values of Variables(Data types)

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159
- Boolean values, either true orfalse
- Strings, such as "Howdy!"
- NULL, a special keyword which refers tonothing.

Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure -a set of statements that performs a specific task when called. A function definition has these basic parts:

- The *function* keyword
- A function name
- A comma-separated list of arguments to the function in parentheses
- The statements in the function in curly braces: { }

Defining a Function

Defining the function means, name the function and specifies what to do when the function is called. You define a function within the <SCRIPT>...</SCRIPT> tags within the <HEAD> ... </HEAD> tags. While defining a function, you can also declare the variables which you will be calling in that function. Here's an example of *defining* a function:

```
function msg()
{
    window.alert("This is an alert box.");
}
```

Here's an example of a function that takes a parameter:

```
function welcome(string)
{
        window.alert("Hi"+string);
 }
```

When you call this function, you need to pass a parameter (such as the word that the user clicked on) into thefunction.

Calling a Function

Calling the function actually performs the specified actions. When you call a

function, this is usually within the BODY of the HTML page, and you usually pass a parameter into the function on which the function will act.
Here's an example of calling the same function:

msg();

For the other example, this is how you may call it:

<inupt type="button" name=”welcome” onClick="*msg1(“Vijay”)*"/>

OBJECTS IN JAVA SCRIPT

When you load a document in your Web browser, it creates a number of JavaScript objects with properties and capabilities based on the HTML in the document and other information. These objects exist in a hierarchy that reflects the structure of the HTML page itself. The pre-defined objects that are most commonly used are the window and document objects. Some of the useful Objects are:

1. Document
2. Window
3. Browser
4. Form
5. Math
6. Date

The DocumentObject

A document is a web page that is being either displayed or created. The document has a number of properties that can be accessed by JavaScript programs and used to manipulate the content of the page.

write or writeln

Html pages can be created on the fly using JavaScript. This is done by using the write or writeln methods of the documentobject.

Syntax:

document.write (“String”); document.writeln (“String”);

In this document is object name and write () or writeln () are methods. Symbol period is used as connector between object name and method name. The difference between these two methods is carriage form feed character that is new line character automatically added into thedocument.

Exmaple:
```
document.write(“<body>”);
document.write(“<h1> Hello</h1>”);
```

bgcolor and fgcolor

These are used to set background and foreground(text) color to webpage. The methods accept either hexadecimal values or common names for colors.

Syntax:

```
document.bgcolor=”#1f9de1”;
document.fgcolor=“silver”;
```

anchors

The anchors property is an array of anchor names in the order in which they appear in the HTML Document. Anchors can be accessed like this:

Syntax:

```
document.anchors[0];
    :
document.anchors[n-1];
```

Links

Another array holding all links in the order in which they were appeared on the Webpage

Forms

Another array, this one contains all of the HTML forms. By combining this array with the individual form objects each form item can be accessed.

The Window Object

The window object is used to create a new window and to control the properties of window.

Methods:

1. *open("URL","name") :* This method opens a new window which contains the document specified by URL and the new window is identified by its name.
2. *close():* this shutdowns thecurrent window.

Properties:

toolbar = [1|0] location= [1|0] menubar = [1|0] scrollbars = [1|0] status = [1|0] resizable = [1|0]

where as 1 means *on* and 0 means *off*

height=pixels, width=pixels : These properties can be used to set the window size.

The following code shows how to open a new window

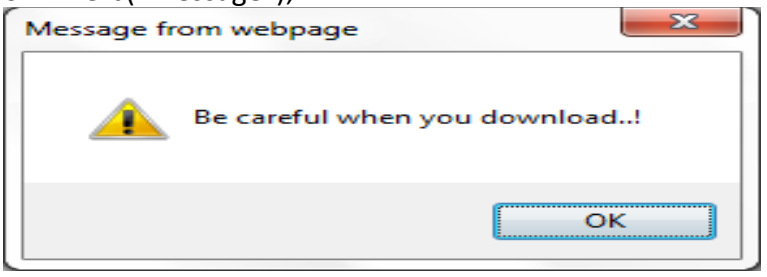
newWin = open("first.html","newWin","status=0,toolbar=0,width=100,height=100");

Window object supports three types of message boxes.

1. Alert box
2. Confirm box
3. Prompt box

Alert box is used to display warning/error messages to user. It displays a text string with *OK* button.

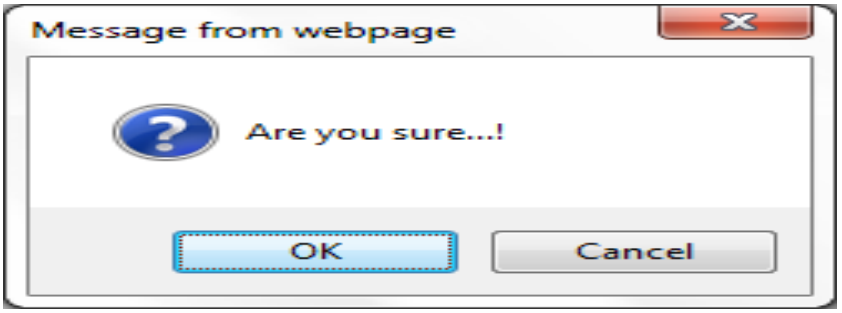
Syntax: window. Alert ("Message");



Confirm Box is useful when submitting form data. This displays a window containing message with two buttons: *OK* and *Cancel.* Selecting *Cancel* will abort the any pending action, while *OK* will let the action proceed.

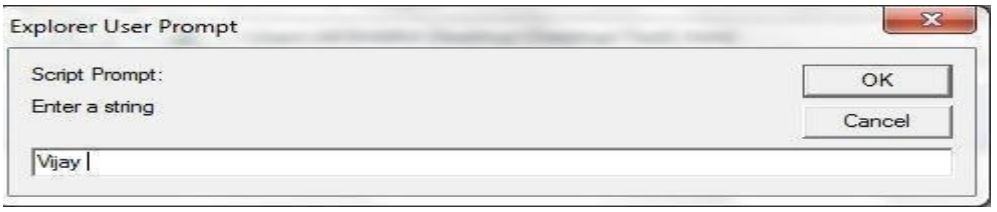
Syntax

window.confirm("String");



Prompt box used for accepting data from user through keyboard. This displays simple window that contains a prompt and a text field in which user can enter data. This method has two parameters: a text string to be used as a prompt and a string to use as the default value. If you don‟t want to display a default then simply use an empty string. Syntax

Variable=window.prompt(“string”,”default value”);



The Form Object

Two aspects of the form can be manipulated through JavaScript. First, most commonly and probably most usefully, the data that is entered onto your form can be checked at submission. Second you can actually build forms through JavaScript.

Form object supports three events to validate the form

onClick = “method()”

This can be applied to all form elements. This event is triggered when the user clicks on the element.

onSubmit = “method()”

This event can only be triggered by form itself and occurs when a form is submitted.

onReset = “method()”

This event can only be triggered by form itself and occurs when a form is reset.

Example: HTML program that applies a random background color when you click on button

```
<html>

<head>

<script language = "javascript">
        function change()

        {

                var clr = document.bgColor=parseInt(Math.random()*999999);
                document.f1.color.value=clr;
         }

</script>

</head>

<body>
```

```
<form name="f1">
        <input type="text" name="color"/>
        <input type="button" value="Cick me" onclick="change()"/>
</form> </body></html>
```

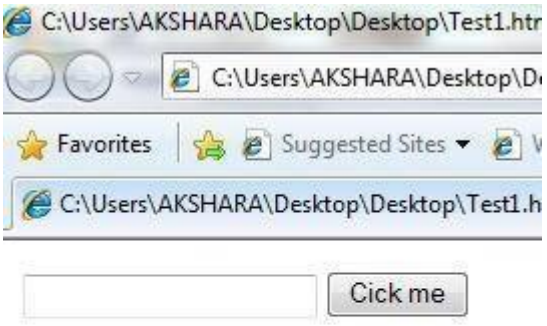


Fig.1: On first run backgroundis white

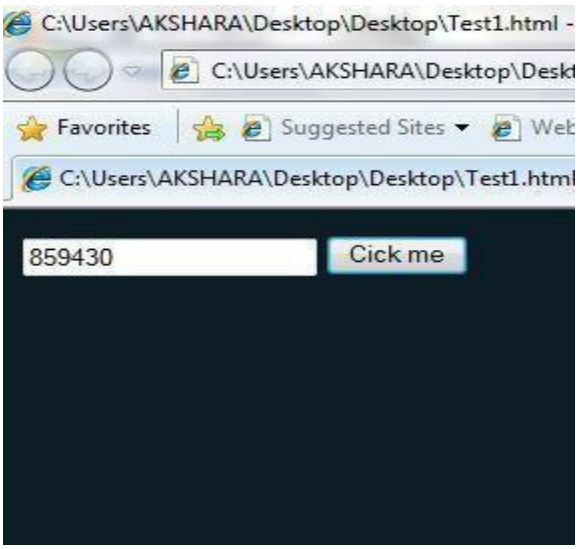


Fig.2: After clicking on button"Clickme": background changed to "black"

The browser Object

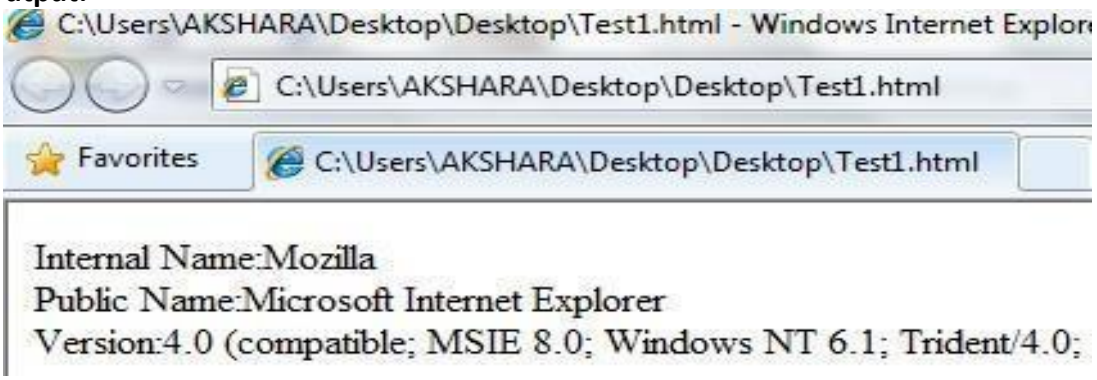
The browser is JavaScript object that can be used to know the details of browser. Some of the properties of the browser object is as follows:

Property	Description
navigator.appCodeName	It returns the internal name for the browser. For major browsers it is *Mozilla*
navigator.appName	It returns the public name of the browser – navigator or Internet Explorer
navigator.appVersion	It returns the version number, platform on which the browser is running.
navigator.userAgent	The strings appCodeName and appVersion concatenated together
navigator.plugins	An array containing details of all installed plug-ins
Navigator.mimeTypes	An array of all supported MIME Types

Example: Write javascript to display internal details of a browser (***Test.html***)

```
<script language = "javascript">
<!--
   document.writeln("Internal      Name:"+navigator.appCodeName);
   document.writeln("<br/>Public Name:"+navigator.appName);
   document.writeln("<br/>Version:"+navigator.appVersion);
--></script>
```

Output:



The Math Object

The *Math* object holds all mathematical functions and values. All the functions and attributes used in complex mathematics must be accessed via this object.

Syntax:

```
Math.methodname();
Math.value;
```

Method	Description	Example
Math.abs(x)	Returns the absolute value	Math.abs(-20) is 20
Math.ceil(x)	Returns the ceil value	Math.ceil(5.8) is 6 Math.ceil(2.2) is 3
Math.floor(x)	Returns the floor value	Math.floor(5.8) is 5 Math.floor(2.2) is 2
Math.round(x)	Returns the round value, nearest integer value	Math.round(5.8) is 6 Math.round(2.2) is 2
Math.trunc(x)	Removes the decimal places it returns only integer value	Math.trunc(5.8) is 5 Math.trunc(2.2) is 2
Math.max(x,y)	Returns the maximum value	Math.max(2,3) is 3 Math.max(5,2) is 5
Math.min(x,y)	Returns the minimum value	Math.min(2,3) is 2 Math.min(5,2) is 2
Math.sqrt(x)	Returns the square root of x	Math.sqrt(4) is 2

Math.pow(a,b)	This method will compute the a^b	Math.pow(2,4) is 16
Math.sin(x)	Returns the sine value of x	Math.sin(0.0) is 0.0
Math.cos(x)	Returns cosine value of x	Math.cos(0.0) is 1.0
Math.tan(x)	Returns tangent value of x	Math.tan(0.0) is 0
Math.exp(x)	Returns exponential value i.e e^x	Math.exp(0) is 1
Math.random(x)	Generates a random number in between 0 and 1	Math.random()
Math.log(x)	Display logarithmic value	Math.log(2.7) is 1
Math.PI	Returns a ∏ value	a = Math.PI; a = 3.141592653589793

The Date Object

This object is used for obtaining the date and time. In JavaScript, dates and times represent in milliseconds since 1st January 1970 UTC. JavaScript supports two time zones: UTC and local. UTC is Universal Time, also known as Greenwich Mean Time(GMT), which is standard time throughout the world. Local time is the time on your System. A JavaScript *Date* represents date from -1,000,000,000 to -1,000,000,000 days relative to 01/01/1970.

Date Object Constructors:

new Date(); Constructs an empty date object.

new Date("String"); Creates a Date object based upon the contents of a text string.

new Date(year, month, day[,hour, minute, second]); Creates a Date object based upon the numerical values for the year, month and day.

```
var dt=new Date();
document.write(dt);          // TueDec 23 11:23:45 UTC+0530 2015
```

Methods in Date object:

Java script date object provides several methods, they can be classified is string form, get methods and set methods. All these methods are provided in the following table.

Method	Description
getDate()	Returns day of the month i.e. 1 to 31
getDay()	Returns an integer representing day of the week(0 - 6), Sunday to Saturday respectively.
getMonth()	Returns month of the year from 0 to 11, January to December respectively.
getFullYear()	Returns four-digit year number
getHours()	Returns hour field of the Date Object in 24 hours time format (0 to 23)
getMinitues()	Returns minute field of the Date Object from 0 to 59
getseconds()	Returns seconds field of the Date Object 0 to 59
setDate(v)	To set the date, day, month and full year
setDay(v)	
setMonth(v)	
setFullYear(y,m,d)	
setHours(v)	To set the hours, minutes, seconds of time
setMinutes(v)	
setSeconds(v)	
toString()	Returns the Date as a string

1.16 DYNAMIC HTML

DHTML is combination of HTML, CSS and JavaScript. It gives pleasant appearance to web page.

Difference between HTML and DHTML

HTML	DHTML
HTML is used to create static web pages.	DHTML is used to create dynamic web pages.
HTML is consists of simple html tags.	DHTML is made up of HTML tags+cascading style sheets+javascript.
Creation of html web pages is simplest but less interactive.	Creation of DHTML is complex but more interactive.

Creating Date Objects

The Date object lets us work with dates.

A date consists of a year, a month, a week, a day, a minute, a second, and a millisecond.

Date objects are created with the **new Date()** constructor.

There are **4 ways** of initiating a date:

new Date()
new Date(milliseconds)
new Date(dateString)
new Date(year, month, day, hours, minutes, seconds, milliseconds)

Using new Date(), **without parameters**, creates a new date object with the **current date and time**:

Example

```
<script>
var d = new Date();
document.getElementById("demo").innerHTML = d;
</script>
```

getDate()	Get the day as a number (1-31)
getDay()	Get the weekday a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	Get the hour (0-23)
getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)
getMonth()	Get the month (0-11)
getSeconds()	Get the seconds (0-59)
getTime()	Get the time (milliseconds since January 1, 1970)

Date Set Methods

Set methods are used for setting a part of a date. Here are the most common (alphabitically):

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day yyyy.mm.dd)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

Introduction to Dynamic HTML

RegExp Object

A regular expression is an object that describes a pattern of characters.

Regular expressions are used to perform pattern-matching and "search-and-replace" functions on text.

Syntax

```
var patt=new RegExp(pattern,modifiers);

or more simply:

var patt=/pattern/modifiers;
```

- pattern specifies the pattern of an expression
- modifiers specify if a search should be global, case-sensitive, etc.

Modifiers

Modifiers are used to perform case-insensitive and global searches:

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
M	Perform multiline matching

Brackets

Brackets are used to find a range of characters:

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character not between the brackets
[0-9]	Find any digit from 0 to 9
[A-Z]	Find any character from uppercase A to uppercase Z
[a-z]	Find any character from lowercase a to lowercase z
[A-z]	Find any character from uppercase A to lowercase z
[adgk]	Find any character in the given set
[^adgk]	Find any character outside the given set
(red|blue|green)	Find any of the alternatives specified

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
.	Find a single character, except newline or line terminator
\w	Find a word character(alphabets & _)
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match not at the beginning/end of a word
\0	Find a NUL character
\n	Find a new line character
\f	Find a form feed character
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character

\xxx	Find the character specified by an octal number xxx
\xdd	Find the character specified by a hexadecimal number dd
\uxxxx	Find the Unicode character specified by a hexadecimal number xxxx

Quantifiers

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n
n{X}	Matches any string that contains a sequence of *X n*'s
n{X,Y}	Matches any string that contains a sequence of X to Y *n*'s
n{X,}	Matches any string that contains a sequence of at least X *n*'s
n$	Matches any string with n at the end of it
^n	Matches any string with n at the beginning of it
?=n	Matches any string that is followed by a specific string n
?!n	Matches any string that is not followed by a specific string n

RegExp Object Methods

Method	Description
compile()	Compiles a regular expression
exec()	Tests for a match in a string. Returns the first match
test()	Tests for a match in a string. Returns true or false

Database programming with JDBC

Syllabus: Database drivers, the java.sql package: connection management, database access, data types, database metadata, exceptions and warnings, loading a database driver and opening connections, establishing a connection, creating and executing sql statements, querying the database, prepared statements, mapping sql types to java, transaction support, save points.

Introduction:

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 commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

- Java Applications
- Java Applets
- Java Servlets
- Java Server Pages (JSPs)
- Enterprise JavaBeans (EJBs)

All of these different executables are able to use a JDBC driver to access a database and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

JDBC Architecture:

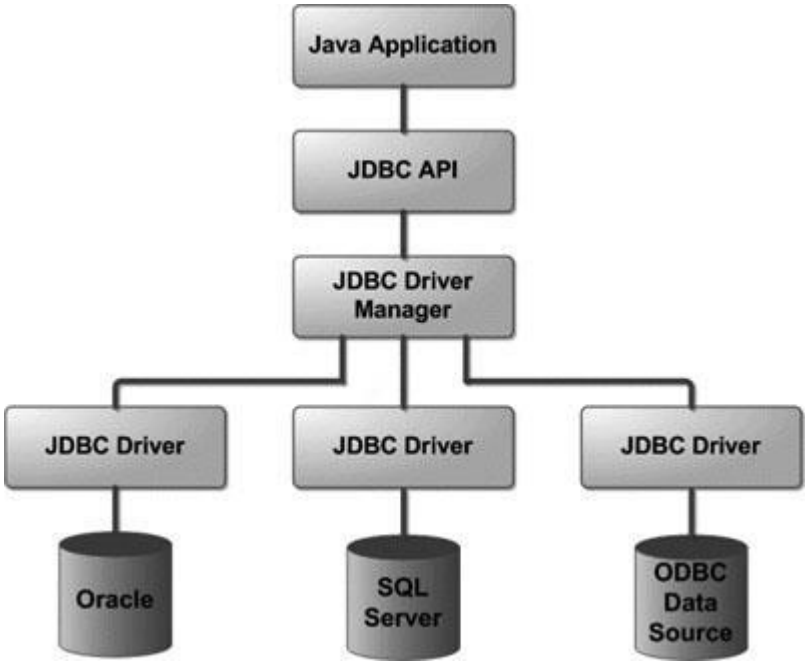
The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application:



Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manage objects of this type. It also abstracts the details associated with working with Driver objects
- **Connection:** This interface provides all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Database Drivers:

- 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.

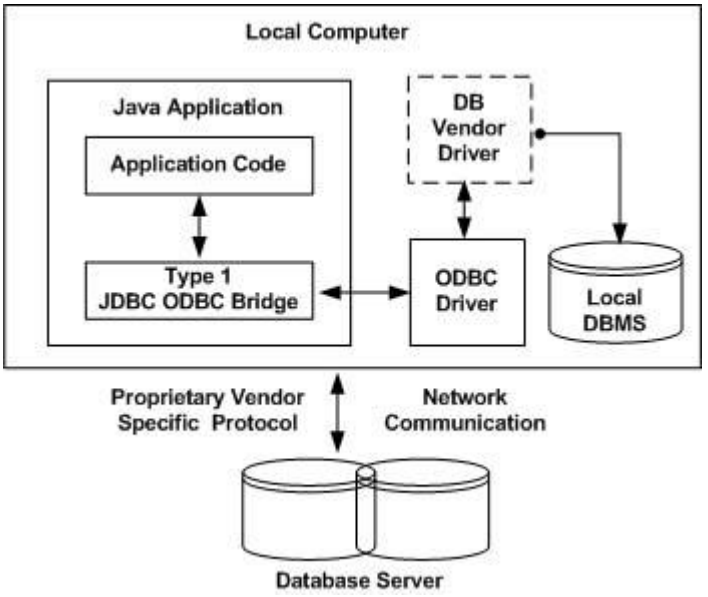
- The *Java.sql* package that ships with JDK contains various classes with their behaviors defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

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 explained below:

Type 1: JDBC-ODBC Bridge Driver:

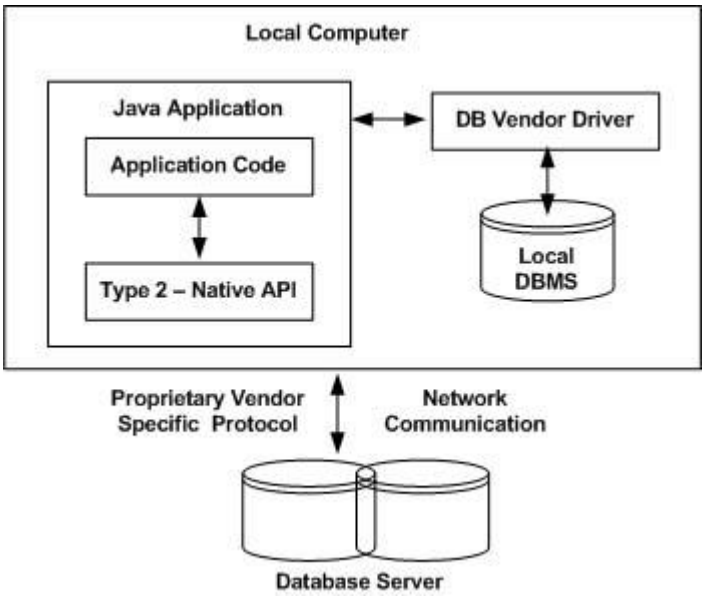
- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this was a useful driver because most databases only supported 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:

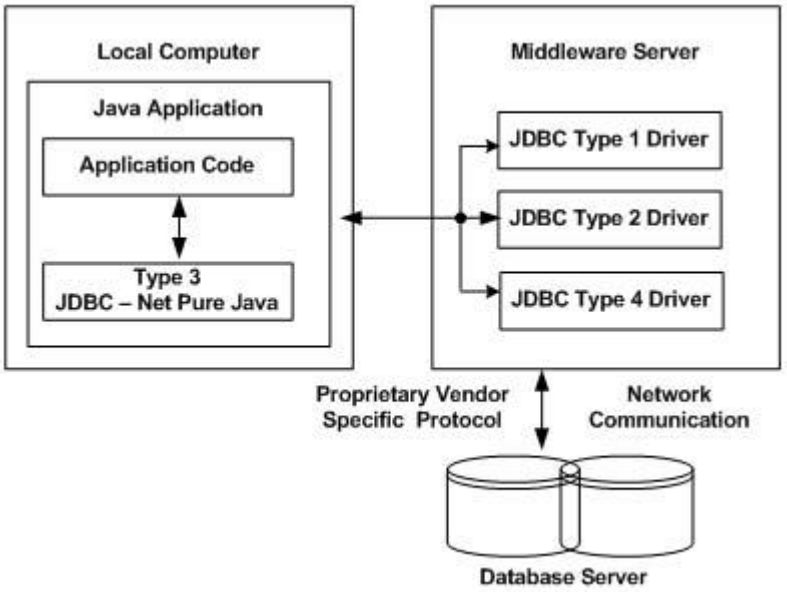
- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.
- If we change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

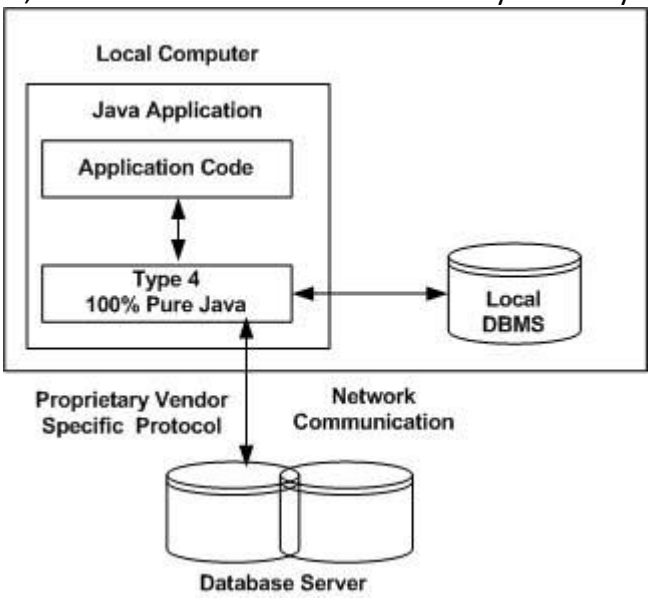
- In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and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

The java.sql Package

The classes in the java.sql package can be divided into the following groups on their functionality.

- Connection Management
- Database Access
- Data Types
- Database Metadata
- Exceptions and Warnings

Connection Management

Class/interface	Description

java.sql.DriverManger class	This class provides the functionality necessary for managing one or more database drivers. Each driver in turn lets you connect to a specific database
java.sql.Driver interface	This is an interface that abstracts the vendor-specific connection protocol.
java.sql.DriverPropertyInfo class	Since each database may require a distinct set of properties to obtain a connection, you can use this class to discover the properties required to obtain the connection.
Java.sql.Connection interface	This interface abstracts most of the interaction with the database. Using a connection, you can send SQL statements to the database, and read the results of execution.

Database Access

Class/interface	Description
java.sql.Statement	This interface lets you execute SQL statements over the underlying connection and access the results.
java.sql.Driver PreparedStatement	This is a variant of the java.sql.Statement interface following for parameterized SQL statements include markers (as "?"), which can be replaced with actual values later on.
Java.sql.CallableStatement interface	This interface lets you execute stored procedures.
Java.sql.ResultSet interface	This interface abstracts results of executing SQL SELECT statements. This interface provides methods to access the results row by row. You can use this interface to access various fields in each rows.

DataTypes

The java.sql package also provides several java data types that correspond to some of the SQL types. You can use one of the following types as appropriate depending on what a field in a result row correspond in a database.

Class/interface	Description
java.sql.Array interface	This interface provides a java language abstraction of ARRAY, a collection of SQL data types.
java.sql.Blob interface	This interface provides a java language abstraction of the SQL type BLOB.
java.sql.Clob interface	This interface provides a java language abstraction of the SQL type CLOB.
java.sql.Date class	This class provides a java language abstraction of the SQL type DATE.
java.sql.Time class	This interface provides a java language abstraction of the SQL type TIME.
Java.sql.Types class	This class holds a set of constant integers, each corresponds to a SQL type.

Database Metadata

The JDBC API also includes facilities to obtain metadata about the database, parameters to statements, and results.

Class/interface	Description
java.sql.DatabaseMetadata interface	You can find out about database features using this interface.
java.sql.ResultSetMetaData interface	This interface provides methods to access metadata of the ResultSet, such as names of columns, their types, the corresponding table name, and other properties.
java.sql.ParameterMetadata interface	This interface allows you access the database types of parameters in prepared statements.

Exceptions and Warnings

Class/interface	Description
java.sql.SQLException	This exception represents all JDBC-related exception conditions. This exception also embeds all driver/database-level exceptions and error codes.
java.sql.SQLWarning	This exception represents database access warnings.
java.sql.BatchUpdateException	This is special case of java.sql.SQLException meant for batch updates.
java.sql.DataTruncation	This is special case of java.sql.SQLWarning meant for data truncation errors.

Loading a Database Driver and Opening Connections

The java.sql.Connection interface represents a connection with a database. The JDBC API provides two different approaches for obtaining connections.

The first uses java.sql.DriverManager and is suitable for non-managed applications such as standalone java database clients.

The second approach is based on the java.sql package that introduces the notation of data sources and is suitable for access in J2EE applications.

DriverManager Class:

The purpose of the java.sql.DriverManager class is to provide a common access layer on top of different database drivers used in application. This class provides three static methods to obtain connections.

The DriverManager, requires each driver needed by the application must be registered before use, so that the DriverManager is aware of it.

The JDBC approach for registering a database driver is as follows.

Class.forName(“oracle.jdbc.driver.OracleDriver”);

Once a driver has been registered with the java.sql.DriverManger, we can use its static methods to get connections.

The java.sql.DriverManger class specifies the following types of methods:

- Methods to manage drivers
- Methods to obtain connections.
- Methods for logging.

Methods to manage drivers

- **public static void registerDriver(Driver driver)**

  This method is used to register a driver with the DriverManager.
- **public static void deregisterDriver(Driver driver)**

  This method deregisters a driver from the DriverManager.
- **public static Driver getDriver(String url)**

  Given a JDBC URL, this method returns a driver that can understand the URL.
- **public static Enumeration getDrivers()**

  This method returns an enumeration of all the registered JDBC drivers registered by classes using the same class loader.

Methods to obtain connections

The DriverManager has three variants of a static method getConnection() used to establish connections.The driver manager delegates these calls to the connect methods on the java.sql.Driver interface.

- **public static Connection getConnection(String url) throws SQLException**

  URL is specified in the form of jdbc:<subprotocol>:<subname>

  for example: jdbc:odbc:Data where Data is a DSN setup using our ODBC driver administrator.

  Whether we get connection with this method or not is depends on whether the database accepts connection request without authentication.
- **public static Connection getConnection(String url, java.util.Properties info) throws SQLException**

  This method requires URL and a java.util.Properties object. It contains required parameters for the specified database. These parameters differs from database to database. Two common parameters are autocommit=true and create =false.
- **public static Connection getConnection(String url,String user,String password) throws SQLException**

  This method takes user and password as the arguments in addition to the URL.
- **public static void setLoginTimeout(int seconds)**

This method can be used to set the login timeout.

- **public static void getLoginTimeout()**

  This method can be used to get the login timeout.

Methods for logging

The following methods access or set a PrintWriter object for logging purpose.

public static void setLogWriter(PrintWriter out)

public static PrintWriter getLogWriter()

In addition, client applications can also log messages using the following method:

public static void println(String message)

Establishing a Connection

To communicate with a database using JDBC, we first establish a connection to the database using **java.sql.Connection** interface.

It has the following public methods:

Function	Methods
Creating statements	createStatement() prepareStatement() prepareCall()
Obtaining database information	getMetaData()
Transaction support	setAutoCommit() getAutoCommit() commit() rollback() setTransactionIsolation() getTransactionIsolation()
Connection status and closing	isClosed() close()
Setting various properties	setReadOnly() isReadOnly() clearWarnings() getWarnings()

Creating and Executing SQL Statements:

We can use a connection object to execute SQL statements by creating a statement, a PreapredStatement, or a CallableStatement. Once we obtain one of these statement objects, we can execute the statement and read the results through ResultSet object.

Creating statement is as follows.

Statement st=conn.createStatement();

PreparedStatement pst=conn.prepareStatement();

CallableStatement cst=conn.prepareCall(String sql): This method is used to call a stored procedure.

The **Statement** interface has the following methods:

Function	Method
Executing statements	execute():for stored procedures. executeQuery();for SELECT statements. executeUpdate():for CREATE,UPDATE,INSERT statements.
Batch updates	addBatch() executeBatch() clearBatch()
Resultset fetch size	setFetchSize() getFetchSize()

Creating JDBC Application:

There are following six steps involved in building a JDBC application:

- **Import the packages**: Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code:

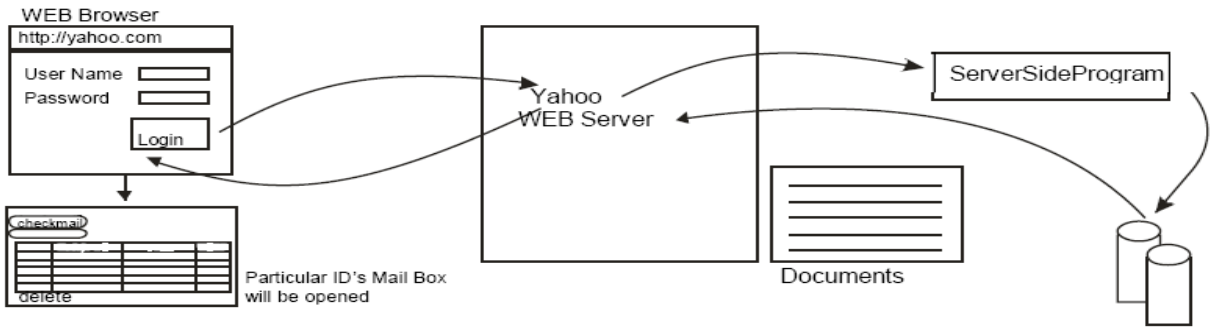
```
// STEP 1. Import required packages
import java.sql.*;
public class FirstExample
{
   public static void main(String[] args)
  {
    Connection conn = null;
   Statement stmt = null;
   try{
       //STEP 2: Register JDBC driver
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

     //STEP 3: Open a connection

        conn = DriverManager.getConnection("jdbc:odbc:Data");
       System.out.println("Connected to database...");
     //STEP 4: Execute a query
     System.out.println("Creating statement...");
     stmt = conn.createStatement();
     /*String sql2;
      sql2="create table std_db(sno Number,name varchar2(10))";
     stmt.execute(sql2);
     System.out.println("table created");*/
   String sql;
     sql = "select empId, empName, age,city from emp_data";
     String sql1;
      sql1="insert into emp_data values(9,'xyz',29,'delhi')";
      stmt.executeUpdate(sql1);
      System.out.println("row inserted");
      String Sql2="insert into emp_data values(10,'mmm',35,'banglore')";
      stmt.executeUpdate(sql1);
      System.out.println("row inserted");
     ResultSet rs = stmt.executeQuery(sql);

     //STEP 5: Extract data from result set
     while(rs.next())
    {
       //Retrieve by column name
       int  id  =  rs.getInt("empId");
       int age = rs.getInt("age");
       String name = rs.getString("empName");
       String city = rs.getString("city");

       //Display values
       System.out.print("empID: " +id);
       System.out.print(", Age: " +age);
       System.out.print(", Empname: " +name);
       System.out.println(", City: " +city);
     }
```

```
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
    }catch(SQLException se)
     {
      //Handle errors for JDBC
      se.printStackTrace();
      }
     catch(Exception e)
      {
      //Handle errors for Class.forName
      e.printStackTrace();
      }
    }//end main
}//end FirstExample
```

JAVA SERVLETS

What is server-side programming?

1. For many web applications server side processing is necessary. i.e., whenever a web Browser sends the data to a web Server the web Server forwards the same to a program on the server which is referred as server side program.
2. The ServerSideProgram receives the data from the web Server, process the data and returns the output back to the web Server which will be given to the web Browser.
3. The web browser receives the data and presents the data on the document.
4. These ServerSidePrograms can be written in any language but the current technologies are ASP,JSP etc.



1. Java servlets are small independent Java programs that can run on the server.
2. Just as an **applet** runs on the **client-side** in a Java-enabled web **browser**, *servlets* execute on

What is Servlet? OR Define servlet

a Java-enabled web *server*.

3. A servlet executes on the server but its output is returned to the client in the form of a HTML page.
4. Although applets display in a graphical user interface, servlets do not display in GUI environment.
5. When a browser sends a request, the server may forward it to a servlet. The servlet processes the request, and constructs an appropriate message and sends it back to the client (browser). This model is based on the HTTP protocol.
6. The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cyclemethods.

What can servlets do?

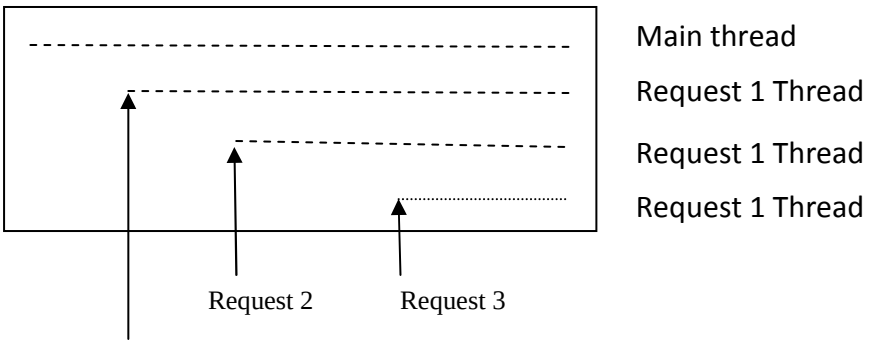
Servlets allow a two-way interaction between the client and server.

1. They can process the user input passed by an HTML file and return a response.
2. They can be used to dynamically build web pages based on the nature of client request.
3. Allow communication between group of people by publishing information submitted by many clients.
4. Forward requests from one server to another for load balancing purpose.

5. Automatically attach web page elements such as header and footer to all pages returned by the server.
6. Interact with server resources such as databases and other applications and return useful information to the client.
7. Provide user authentication and other security mechanisms.

What are the advantages of servlets?

1. **Capable of running in-process** – Servlets run in-process and therefore they are loaded only once. Due to the multithreaded nature of servlets, all client requests can be serviced by separate threads. It is not necessary to create a separate process to handle each client request.



2. **Compiled:** Unlike scripting languages, servlets are compiled into Java byte-code. Therefore they can execute much more quickly than scripting languages. Compilation also offers the advantage of error and type checking. Compilation makes servlets more stable and easier to develop and debug than scripting languages. Also, compiled code is more compact.

3. **Crash-resistant:** Servlets are written in Java and executed by a Java Virtual Machine (JVM). JVM does not allow direct memory access and hence crashes do not occur. Also, before execution, JVM verifies that the compiled Java class files are valid and do not perform any illegal operations.

4. **Cross-platform:** the "write once-run anywhere" capability allows servlets to be easily distributed without rewriting for each platform. Servlets operate identically without modification whether they are running on Unix, or Windows.

5. **Durable:** Servlets are durable object. That is they remain in memory until they are destroyed. Thus, servlets are instantiated only once in order to service many requests. A servlet can create other objects, e.g., a servlet can create a database connection when it is first loaded. This connection can then be shared across all requests.

6. **Dynamic Loading:** Servlets can be dynamically loaded locally or across a network. This ensures that the unused servlets are not occupying system resources. They are loaded only when needed.

7. **Multithreaded:** Servlets support multithreading. Thus, client request can be handled by separate threads within a single process. This approach requires fewer resources and executes much more quickly.
8. **Protocol Independent:** Servlets are protocol independent. Thus, servlet can support FTP commands, SMTP(Simple Mail Transfer Protocol), POP3 (Post Office Protocol), telnet, HTTP and other protocols.
9. **Cross server:** Servlets can run on almost all popular web servers.
10. **Written in Java:** Since servlets are written in Java, they offer many advantages. These advantages are: true OOPs-based language, strong type checking, multithreading support, built-in security, optimized code, automatic garbage collection, built-in network support, and built-in internationalization through Unicode.

Distinguish between Applets and Servlets

Applet	Servlet
1. An applet is a Java program that runs within a Web browser on the client machine.	1. A servlet is a Java program that runs on the web server.
2. An applet can use the interface classes such as AWT or Swing.	2. A servlet does not have a user interface.
3. Applets do not communicate with the server.	3. Servlets run on the server.
4. Client-side java programs that run in browser are called applets.	4. Server-side Java programs are called Servlets.

The client sends a request to the server.

How do servlets work?

1. The server loads the servlet and creates a thread for the servlet process. The servlet is loaded when the first request is made. It stays loaded in memory until it is removed or server is shut down.
2. The server sends the requested information to the servlet.
3. The servlet builds a response and passes it to the server.
4. The server sends the response back to the client.

Write a note on the Servlet Life Cycle

Each servlet has the same life cycle:

- A server loads and initializes the servlet [init() method]
- The servlet handles zero or more client requests [service() method]
- The server removes the servlet (some servers do this step only when they shut down) [destroy()method]

➢ **Step 1:** A user enters a URL to a browser. The browser generates an HTTP request for this URL and this request is sent to the appropriate server.

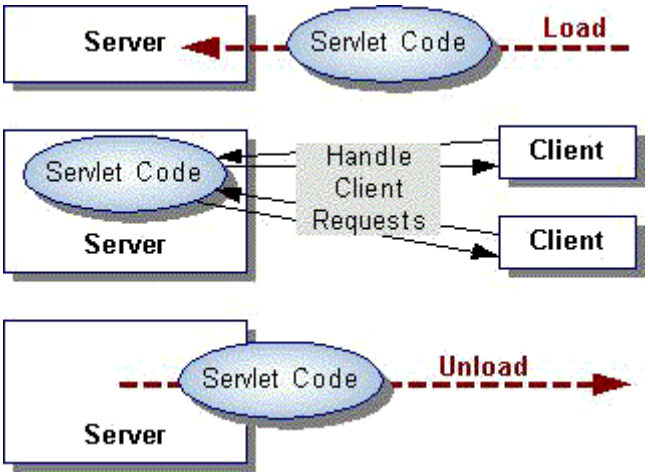
➢ **Step 2:** The HTTP request is received by the web server. The server maps this request toa particular servlet. This servlet is dynamically retrieved and loaded into the server.

- **Step 3:** The server invokes the init() method of the servlet. This method is invoked only when the servlet is first loaded into the memory. We can pass initialization parameters to the servlet.

- **Step 4:** The server invokes the service() method of the servlet. This method is called to process the request HTTP request. The servlet can read data that has been provided in the HTTP request. The service method can also create a HTTP response for the client. The servlet remains in the server's address space and is available to process any other requests from other clients. The service method is called for each request.

- **Step 5:** The server calls the destroy() method when a servlet has to be unloaded from the server memory. Once this method is called, the servlet will give up all file handles that were allotted to it. Important data may be saved to a persistent store. The memory allocated to the servlet and its objects is released.

  Sometimes it may be necessary to store the current state of a servlet in some text file or in a database. This is required if the servlet wants the data from the previous state. This stored data is called *persistent data* and his technique is called *persistence*.



Initializing a servlet:

1. When a server loads a servlet, the server runs the servlet's initmethod.
2. Initialization is completed before client requests are handled.
3. The server calls the init() method once, when the server loads the servlet, and will not call the init() method again unless the server is reloading the servlet.
4. The server will reload a servlet only after it has destroyed the servlet by running the destroy() method.
5. If an initialization error occurs and the servlet cannot handle client requests, the program throws an UnavailableException. E.g., not able to establish a required network connection.
6. If a servlet uses a database, the init() method could try to open a connection and throw the UnavailableExceptionif it was unsuccessful.

Interacting with Clients:

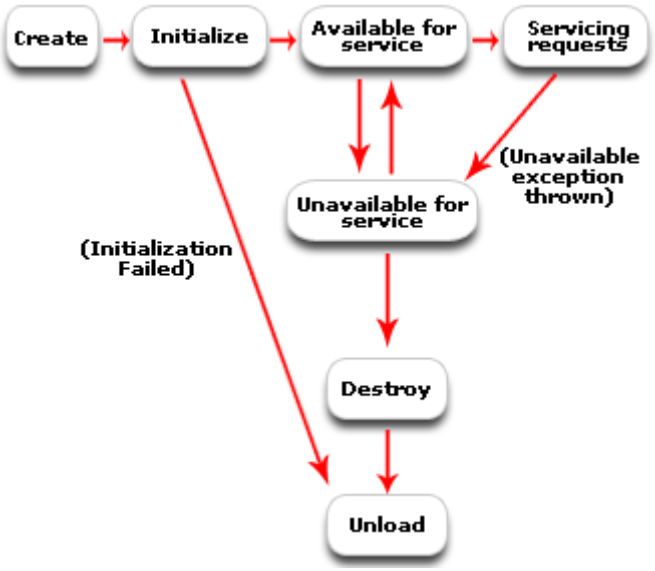
1. A Servlet handles client requests through its service() method. The service method supports

standard HTTP client requests by dispatching each request to a method designed to handle that request.

2. Each client request is done through a servlet object of type ServletRequest. The response of the servlet is sent back through the servlet response object of type ServletResponse.
3. When the client makes a request, the servlet engine passes both the servlet request object and the servlet response object as parameters to the servlet. The ServletRequest gives the servlet the following: form data, and protocol methods. The ServletResponse allows the servlet to set the response headers.

Destroying a Servlet:

1. A server calls the destroy()method after all service calls have been completed. This method can also be called after a pre-defined time.
2. When the servlet engine decides to destroy a servlet, it invokes the destroy() method. The servlet releases system resources.



Servlet Life Cycle Diagram

How are HTTP Request and Response handled by servlets?

a) The HttpServlet class contains methods that handle the various types of HTTP requests.
b) These methods are doGet(), doDelete(), doPost(), doPut(), doHead(). The GET and POST requests are commonly used when handling form input.

Creating a Servlet :

To create a servlet, we create a class that extends the "HttpServlet" class and overrides the following methods (i) doGet( ) (ii) doPost( )

- If the WEB Browser sends the data in the Get( ) method then doGet( ) method of servlet will be executed.
- If the WEB Browser sends the data in the Post method then the doPost( ) method of the servlet will be executed.
- If the WEB Browser does not specify any method, then the doGet( ) will be executed

HTTP requests are of two type: GET and POST. A web browser after receiving the details from the user, can forward these to the web server by one of the following ways: GET, or POST

✓ **How to handle HTTP GET requests:**

When a web Browser is required to send data to the web Server, the browser converts the data in a particular format, which is known as URL coding. The server side program receives the converted data, undoes the conversion, and processes the data.

The following 4 rules will be used by the WEB Browsers for URL coding.

1. All the fields will be separated by & symbol.
2. Each field contains the name of the field and the value of the field separated by = symbol.
3. All spaces will be converted to + symbols.
4. All special characters such as +, & etc., will be converted to hexadecimal values prefixed with % symbol.

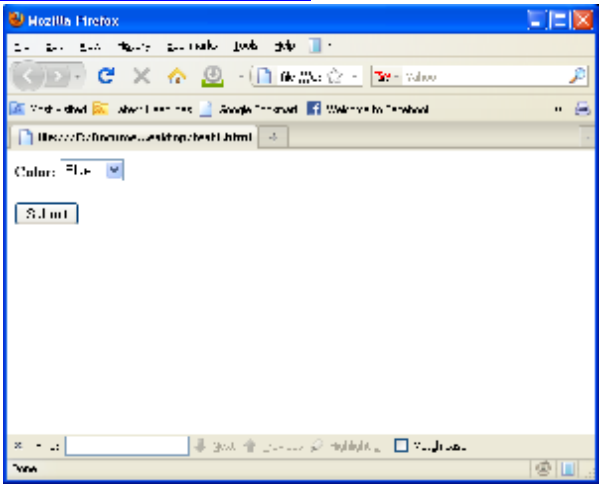
Example 1:

http://myserver.com/SSPname?Empno = 101&Ename = abc+xyz&Job

= clk Server param1 value1 p2 v2 p3 v3

Example 2: http://localhost:8080/servlet/ColorServlet?color=Red
The characters to the right of the question mark are called the *query string*.

```
<HTML>
<BODY>
<FORM name = "form1" action = http://localhost:8080/servlet/ColorServlet>
<B>Color: </B>
<SELECT Name = "color", Size = "1">
        <option value = "Red">Red</option>
        <option value = "Blue">Blue</option>
        <option value = "Green">Green</option>
</SELECT>
<BR><BR>
<INPUT TYPE = Submit value = "Submit">
</FORM>
</BODY>
</HTML>
```



The output of this code is shown here.

The servlet source code is shown

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorServlet extends HttpServlet
{
```

below:

```
      {
            String color = req.getParameter("color");

            res.setContentType("text/"html");
            PrintWriter out = res.getWriter();
            out.println("The selected color is   " );
            out.println(color);
            out.close();
      }
}
```

In the Http form, if the user selects the Red color and clicks on the SUBMIT button, the URL sent to the servlet is http://localhost:8080/servlet/ColorServlet?color=Red

✓ **How to handle the HTTP POST requests:**

When the browser sends an HTTP request using the POST method, the doPost() method is

called. Consider the following statement in the HTML form code at the client side:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet
{
     public void doPost (HttpServletRequest req,
                         HttpServletResponse res)
          throws ServletExceptio, IOException
     {
          String color = req.getParameter("color");

          res.setContentType("text/"html");
          PrintWriter out = res.getWriter();
          out.println("The selected color is   " );
          out.println(color);
          out.close();
     }
}
```

<FORM name = "form1" METHOD = "POST" action =

http://localhost:8080/servlet/ColorPostServlet> The rest of the code remains as in the

previous HTML file.The source code for the ColorPsotServlet.java is as shown below:

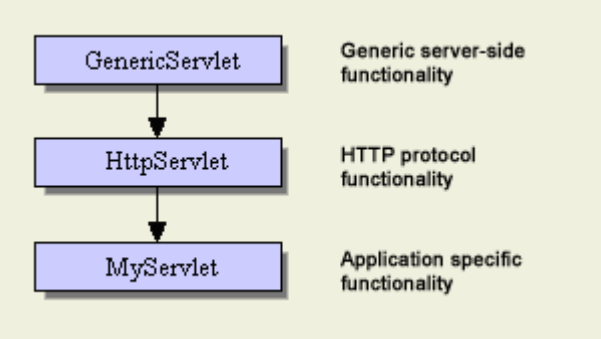
When the user clicks on the SUBMIT button in the HTML form, the URL sent from the browser is:
http://localhost:8080/servlet/ColorPostServlet. The parameter names and values are sent in the body of the HTTP request unlike the query string that was created in GET method.

What is the difference between the GET and POST methods?

- In "GET" the name-value pairs are submitted as a query string in the URL, while in POST all the name-value pairs are submitted in the message body of the request.
- GET is not secure as it is visible in plain text form in the address bar of the browser (as a URL). POST method is secure because the name-value pairs cannot be seen in the address bar of the browser.
- In GET, the length of the URL string is limited while in the POST method, there is no such restriction.
- If GET method is used and if the page is refreshed, it will not prompt before the request is submitted again. If POST method is used and the page is refreshed, it will prompt before the request is resubmitted.
- If the method is not mentioned in the form tag, the default method is GET.

Explain the Java Servlet architecture.

- A servlet is an object that extends either the *javax.servlet.GenericServlet* class or the *javax.servlet.http.HttpServlet* class.
- The *javax.servlet.GenericServlet* class defines methods for building protocol-independent servlets.
- The *javax.servlet.http.HttpServlet* class extends this class to provide HTTP-specificmethods.
- The diagram below illustrates the hierarchy of a typical HTTP servlet.



The GenericServlet class defines generic server-side operations; the HttpServlet extends this to define HTTP-specific operations; and application-specific servlets extend this to provide application-specific operations.

- **Client Interaction:**
  - When a servlet accepts a call from a client, it receives two objects:
    - A ServletRequest, which encapsulates the communication from the client to the server.
    - A ServletResponse, which sends the response from the servlet back to the client.
  - ServletRequest and ServletResponse are interfaces defined by the javax.servlet package.

- **The ServletRequest Interface:**
  - The ServletRequestinterface allows the servlet access to following data:
  - names of the parameters passed by the client, the protocol being used by the client, and the names of the remote host that made the request and the server that receivedit.
  - the input stream, ServletInputStream. Servlets use the input stream to get data from clients that use application protocols such as the HTTP methods (POST and PUT).
- **The Service Response Interface:**
  - The ServletResponse interface gives the servlet methods for replying to the client.
  - It allows the servlet to set the content length and MIME type of the reply.
  - Provides an output stream, ServletOutputStream, and a Writer through which the servlet can send the reply data.

Write a short note on the doXXX methods.

The HttpServlet class contains methods that handle the various types of HTTP requests. These methods are doGet(), doDelete(), doPost(), doPut(), doOptions(), and doTrace(). Which method is used depends on the type of HTTP request.

Method	Description
doPut()	This method is used for uploading a file.
doDelete()	This method is used for deleting a document from the server. The documented to be deleted is indicated in the URL section of the request.
doOptions()	This is called by the server to allow a servlet to handle an OPTIONS request. The OPTIONS request determines which HTTP methods the server supports.
doTrace()	This is used for debugging
doGet()	This method is called in response to an HTTP GET request. This happens when the user clicks on a link, or enters a URL in the browser address bar. It also happens when the HTML form uses the GET method.
doPost()	This method is called in response to an HTTP GET request. This happens when the HTML form uses the POST method.

What is a cookie?

1. Cookies are small files which are stored on a user's computer by the server.
2. They can hold small amounts of data for a specific client and website.
3. Cookies can be accessed either by the web server or the client computer. The server can send a page custom-made for a particular client, or location, or time of day. Thus, we can say that cookies are used for session management.
4. A cookie can be read back by the server. Thus the server can "remember" the client. This is important because HTTP itself is a stateless protocol. Once the data is delivered by the server to the client browser, the server will not keep any further information about the client.

5. Cookies have a name and a single value. They may have optional attributes such as version number, expiry date, a comment for the user, etc.
6. Cookies are assigned by the server to the client. They is sent using fields added to the HTTP response header. Cookies are passed back to the server using fields added to the HTTP request headers.

Cookie methods:

Method	Description
getName()	Gets the name of the cookie. The name of the cookie cannot be changed after it is created.
getValue()	Returns the value of the cookie
setValue(String)	Sets the value of the cookie
getMaxAge()	Returns the maximum specified age of the cookie.
setMaxAge()	Sets the maximum age of the cookie. The unit of measurement is seconds. If it is set to 0, the cookie will be deleted..
getDomain()	Returns the domain that this cookie belongs to
setDomain(String)	Sets the domain to which this cookie belongs.The cookie will be visible only to the specified domain.
Cookie(string, string)	This is a constructor of the class. It defines a cookie with an initial name-value pair.

What is session tracking? Why is it useful?

1. HTTP is a stateless protocol. Each request is independent of the previous one. But in some applications such as online shopping, banking, etc, it is necessary to save the state information so that the information can be collected from the user over several interactions. Sessions provide this mechanism.
2. A session can be created by the getSession() method of HttpServletRequest. This method returns an HttpSession object. The setAttribute(), getAttribute(), removeAttribute() and getAttributeNames() methods of the HttpSession manage the bindings between the names and objects.

Program 1: Write a servlet that illustrates how to use the session state.

```
import java.io.*;
import javax.servlet.*;
import java.util.*;
public class HelloServlet extends GenericServlet
{
 public void service(ServletRequest req, ServletResponse res)
                throws ServletException, IOException
  {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.println("<HTML>");
      out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<B>Hello World</B>");
      out.println("</BODY></HTML>");
      out.close();
  }
}
```

```
import javax.servlet.http;
public class DateServlet extends HttpServlet
{
     public void doGet(HttpServlet req,
                       HttpResponse res) throws
                       ServletException, IOException
     { //get the http response object
       HttpSession hs =
       req.getSession(true);

       //get writer
       res.setContentType("text/htm
       l"); PrintWriter out =
       res.getWriter();

       //Display date and time of last
       access Date dt = (Date)
       hs.getAttribute("date");
```

```
       {
            out.println("Last access was on " + dt);
       }

       //display current date
       dt = new Date();
       hs.setAttribute("date", dt);
       out.println("Current date is : " + dt);
     }
}
```

Program 2: Write a program that prints the message "Hello World"

- We import the javax.servlet package. This package contains the classes and interfaces required to build the srvlet.
- We define HelloServlet as a subclass of GenericServlet. The GenericServlet class simplifies the creation of a servlet. The GenericServlet class provides init() and destroy() methods and we have to only write the service() method.

- We have overridden the service() method. This method handles the client requests. In this method, the first argument is the ServletRequest object. Through this object we can read the data provided by the client. The second argument is the ServletResponse object and this object enables the servlet to send a response to the client.
- The call to setContentType() indicates that the browse should interpret the content as HTML source code.
- The getWriter() method obtains a PrintWriter. The println() method is used to send HTML source code as the HTTP response.

Program 3: Write a program to display the "Hello, World" message.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException
      {
          res.setContentType("text/html");
          PrintWriter out = res.getWriter();

          out.println("<HTML>");
          out.println("<HEAD><TITLE>Hello World</TITLE></HEAD>");
          out.println("<BODY>");
          out.println("<BIG>Hello World</BIG>");
          out.println("</BODY></HTML>");}}
```

a) We import the javax.servlet package. This package contains classes and interfaces required to build servlets.
b) We define HelloWorld as a subclass of HttpServlet.
c) The HttpServlet class provides specialized methods that handle the various types of HTTP requests. These methods are doGet(), doDelete(), doPost(), doPut(), doHead(). The GET and POST requests are commonly used when handling form input. In this program we have used the doGet() method.
d) PrintWriter – This is a class for character stream I/O. We can use the System.out to write to the console, but PrintWriter is preferred because it can be used to internationalize the output. This class supports the print and println methods.

Program 4: Write a servlet that counts and displays the number of times it has been accessed since the last server reboot:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleCounter extends HttpServlet
{
        int count = 0;

        public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException
        {
                res.setContentType("text/plain");
                PrintWriter out = res.getWriter();
                count++;
                out.println("Since loading, this servlet has been accessed " +
                             count + " times.");}}
```

Program 5: Write a servlet that counts the number of times it has been accessed, the number of instances created by the server, and the total times all of them have been accessed.

```
import java.io.*; import
java.util.*; import
javax.servlet.*;
import javax.servlet.http.*;

public class SCounter extends HttpServlet
{
   static int classCount = 0;                  // shared by all instances int count
   = 0;                                        // separate for each servlet
   static Hashtable instances = new Hashtable();                    // also shared

   public void doGet(HttpServletRequest req, HttpServletResponse res) throws
                      ServletException, IOException
   {
          res.setContentType("text/plain"); PrintWriter
          out = res.getWriter();

          count++;
          out.println("Since loading, this servlet instance has been accessed " + count + "
                             times.");

      // Keep track of the instance count by putting a reference to this
      // instance in a Hashtable. Duplicate entries are ignored.
```

```
        // The size() method returns the number of unique instances stored.

            instances.put(this, this);
            out.println("There are currently " + instances.size() + " instances.");


        classCount++;
        out.println("Across all instances, this servlet class has been " + "accessed " + classCount +
                            " times.");
    }
}
```

Introduction to JSP

Introducing JSP:

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 %>.

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.

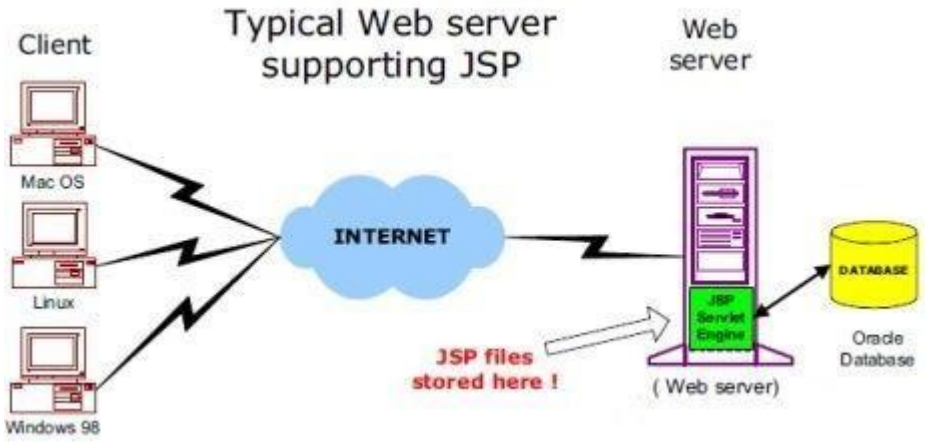
Using JSP, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

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.

The web server needs a JSP engine ie. container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This document makes use of Apache which has built-in JSP container to support JSP pages development.

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

Following diagram shows the position of JSP container and JSP files in a Web Application.



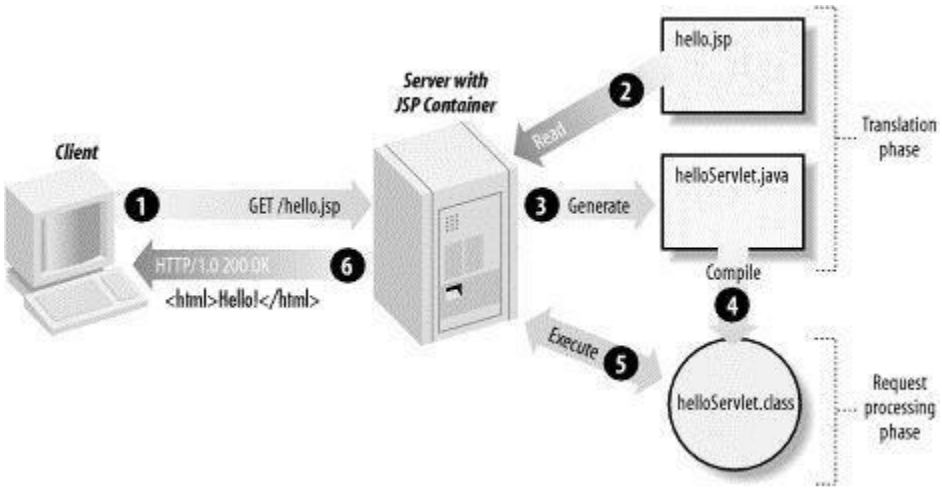
JSP Processing:

The following steps explain how the web server creates the web page using JSP:

- As with a normal page, your browser sends an HTTP request to the web server.
- The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of .html.

- The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page.
- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
- A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

All the above mentioned steps can be shown below in the following diagram:



Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with other scripting languages (such as PHP) and therefore faster.

So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet

JSP Directives:

JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of JSP processing.

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

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

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.

The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.

There are three types of directive tag:

Directive	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

The page Directive:

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive:

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

Attributes:

Following is the list of attributes associated with page directive:

Attribute	Purpose
Buffer	Specifies a buffering model for the output stream.
autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
Extends	Specifies a superclass that the generated servlet must extend
Import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
Info	Defines a string that can be accessed with the servlet's getServletInfo() method.
isThreadSafe	Defines the threading model for the generated servlet.
Language	Defines the programming language used in the JSP page.
Session	Specifies whether or not the JSP page participates in HTTP sessions
isELIgnored	Specifies whether or not EL expression within the JSP page will be ignored.
isScriptingEnabled	Determines if scripting elements are allowed for use.

The include Directive:

The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.

The general usage form of this directive is as follows:
<%@ include file="relative url" >

The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

The taglib Directive:

The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.
The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.

The taglib directive follows the following syntax:
<%@ taglib uri="uri" prefix="prefixOfTag" >

JSP Scripting Elements:

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 %>
Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple and first example for JSP:

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World!<br/>
<%
out.println("Your IP address is " + request.getRemoteAddr());
%>
</body>
</html>
```

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 of JSP Declarations:

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

Following is the simple example for JSP Declarations:

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

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 %>
```

You can write XML equivalent of the above syntax as follows:

```
<jsp:expression>
  expression
</jsp:expression>
```

Following is the simple example for JSP Expression:

```
<html>
<head><title>A Comment Test</title></head>
<body>
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
</body>
</html>
```

This would generate following result:

Today's date: 11-Sep-2010 21:24:25

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" part of your JSP page.

Following is the syntax of JSP comments:

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

Following is the simple example for JSP Comments:

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- This comment will not be visible in the page source --%>
</body>
</html>
```

A Test of Comments

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary:

Syntax	Purpose
<%-- comment --%>	A JSP comment. Ignored by the JSP engine.
<!-- comment -->	An HTML comment. Ignored by the browser.
<\%	Represents static <% literal.
%\>	Represents static %> literal.

\'	A single quote in an attribute that uses single quotes.
\"	A double quote in an attribute that uses double quotes.

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 and there are following JSP actions available:

Syntax	Purpose
jsp:include	Includes a file at the time the page is requested
jsp:useBean	Finds or instantiates a JavaBean
jsp:setProperty	Sets the property of a JavaBean
jsp:getProperty	Inserts the property of a JavaBean into the output
jsp:forward	Forwards the requester to a new page
jsp:plugin	Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin
jsp:element	Defines XML elements dynamically.
jsp:attribute	Defines dynamically defined XML element's attribute.
jsp:body	Defines dynamically defined XML element's body.
jsp:text	Use to write template text in JSP pages and documents.

Common Attributes:

There are two attributes that are common to all Action elements: the **id** attribute and the **scope** attribute.

- **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext
- **Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b) request, (c) session, and (d) application.

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.

Following is the list of attributes associated with include action:

Attribute	Description
Page	The relative URL of the page to be included.
Flush	The boolean attribute determines whether the included resource has its buffer flushed before it is included.

Example:

Let us define following two files (a) date.jsp and (b) main.jsp as follows:

Following is the content of date.jsp file:

```
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

Here is the content of main.jsp file:

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

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 bean properties.

Following is the list of attributes associated with useBean action:

Attribute	Description
Class	Designates the full package name of the bean.
Type	Specifies the type of the variable that will refer to the object.
beanName	Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

The <jsp:setProperty> Action

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action.

Following is the list of attributes associated with setProperty action:

Attribute	Description
Name	Designates the bean whose property will be set. The Bean must have been previously defined.
Property	Indicates the property you want to set. A value of "*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods.
Value	The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action is ignored.
Param	The param attribute is the name of the request parameter whose value the property is to receive. You can't use both value and param, but it is permissible to use neither.

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 ans simple syntax is as follows:

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

Following is the list of required attributes associated with setProperty action:

Attribute	Description
Name	The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
Property	The property attribute is the name of the Bean property to be retrieved.

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.

The simple syntax of this action is as follows:

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

Following is the list of required attributes associated with forward action:

Attribute	Description
Page	Should consist of a relative URL of another resource such as a

static page, another JSP page, or a Java Servlet.

Example:

Let us reuse following two files (a) date.jps and (b) main.jsp as follows:

Following is the content of date.jsp file:

```
<p>
  Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

Here is the content of main.jsp file:

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
</html>
```

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like as below. Here it discarded content from main page and displayed content from forwarded page only.

Today's date: 12-Sep-2010 14:54:22

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.

Following is the typical syntax of using plugin action:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                    width="60" height="80">
  <jsp:param name="fontcolor" value="red" />
  <jsp:param name="background" value="black" />

  <jsp:fallback>
    Unable to initialize Java Plugin
  </jsp:fallback>

</jsp:plugin>
```

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

The <jsp:element> Action

The <jsp:attribute> Action

The <jsp:body> Action

The <jsp:element>, <jsp:attribute> and <jsp:body> actions are used to define XML elements dynamically. The word dynamically is important, because it means that the XML elements can be generated at request time rather than statically at compile time.
Following is a simple example to define XML elements dynamically:

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
    xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
  Value for the attribute
</jsp:attribute>
<jsp:body>
  Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

This would produce following HTML code at run time:

```
<html xmlns="http://www.w3c.org/1999/xhtml"
    xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
  Body for XML element
</xmlElement>
</body>
</html>
```

The <jsp:text> Action:

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

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

The body fo the template cannot contain other elements; it can only contain text and EL expressions (Note: EL expressions are explained in 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.

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

If you need to include a DOCTYPE declaration, for instance for XHTML, you must also use the <jsp:text> element as follows:

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>
<books><book><jsp:text>
   Welcome to JSP Programming
</jsp:text></book></books>
</body>
</html>
```

JSP Implicit 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.

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 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.

Scope of Implicit Objects:

The availability of a JSP object for use from a particular place of the application is defined as the scope of that JSP object. Every object created in a JSP page will have a scope. Object scope in JSP is segregated into four parts and they are page, request, session and application.

Syntax: < jsp:action id="" scope="page/request/session/application />

- **page**
  'page' scope means, the JSP object can be accessed only from within the same page where it was created. The default scope for JSP objects created using <jsp:useBean> tag is page. JSP implicit objects out, exception, response, pageContext, config and page have 'page' scope.
- **request**
  A JSP object created using the 'request' scope can be accessed from any pages that serves that request. More than one page can serve a single request. The JSP object will be bound to the request object. Implicit object request has the 'request' scope.
- **session**
  'session' scope means, the JSP object is accessible from pages that belong to the same session from where it was created. The JSP object that is created using the session scope is bound to the session object. Implicit object session has the 'session' scope.
- **application**
  A JSP object created using the 'application' scope can be accessed from any pages across the application. The JSP object is bound to the application object. Implicit object application has the 'application' scope.

JSP pages as XML documents:

When you send XML data via HTTP, it makes sense to use JSP to handle incoming and outgoing XML documents for example RSS documents. As an XML document is merely a bunch of text, creating one through a JSP is no more difficult than creating an HTML document.

Sending XML from a JSP:

You can send XML content using JSPs the same way you send HTML. The only difference is that you must set the content type of your page to text/xml. To set the content type, use the <%@page%> tag, like this:

```
<%@ page contentType="text/xml" %>
```

Following is a simple example to send XML content to the browser:

```
<%@ page contentType="text/xml" %>

<books>
  <book>
    <name>Padam History</name>
    <author>ZARA</author>
    <price>100</price>
  </book>
</books>
```

Try to access above XML using different browsers to see the document tree presentation of the above XML.

Processing XML in JSP:

Before you proceed with XML processing using JSP, you would need to copy following two XML and XPath related libraries into your <Tomcat Installation Directory>\lib:

- **XercesImpl.jar:** Download it from http://www.apache.org/dist/xerces/j/
- **xalan.jar:** Download it from http://xml.apache.org/xalan-j/index.html

Let us put following content in books.xml file:

```
<books>
<book>
  <name>Padam History</name>
  <author>ZARA</author>
  <price>100</price>
</book>
<book>
  <name>Great Mistry</name>
  <author>NUHA</author>
  <price>2000</price>
</book>
</books>
```

Now try the following main.jsp, keeping in the same directory:

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

<html>
<head>
  <title>JSTL x:parse Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:import var="bookInfo" url="http://localhost:8080/books.xml"/>

<x:parse xml="${bookInfo}" var="output"/>
<b>The title of the first book is</b>:
<x:out select="$output/books/book[1]/name" />
<br>
<b>The price of the second book</b>:
<x:out select="$output/books/book[2]/price" />

</body>
</html>
```

Now try to access above JSP using http://localhost:8080/main.jsp, this would produce following result:

Formatting XML with JSP:

Consider the following XSLT stylesheet style.xsl:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
  <html>
  <body>
   <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="books">
  <table border="1" width="100%">
    <xsl:for-each select="book">
      <tr>
        <td>
          <i><xsl:value-of select="name"/></i>
        </td>
        <td>
          <xsl:value-of select="author"/>
        </td>
        <td>
          <xsl:value-of select="price"/>
        </td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Now consider the following JSP file:

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

<html>
<head>
  <title>JSTL x:transform Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:set var="xmltext">
  <books>
    <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
    </book>
    <book>
```

```
    <name>Great Mistry</name>
    <author>NUHA</author>
    <price>2000</price>
   </book>
  </books>
</c:set>

<c:import url="http://localhost:8080/style.xsl" var="xslt"/>
<x:transform xml="${xmltext}" xslt="${xslt}"/>

</body>
</html>
```

Introduction to MVC Architecture:

Refer Text book