AJAX

AJAX is an acronym for **Asynchronous JavaScript and XML**. AJAX is a new technique for creating better, faster and interactive web applications with the help of JavaScript, DOM, XML, HTML, CSS etc. AJAX allows you to send and receive data asynchronously without reloading the entire web page. So it is fast.

AJAX allows you to send only important information to the server not the entire page. So only valuable data from the client side is routed to the server side. It makes your application interactive and faster.

Ajax is the most viable Rich Internet Application(RIA) technique so far.

Where it is used?

There are too many web applications running on the web that are using AJAX Technology. Some are:

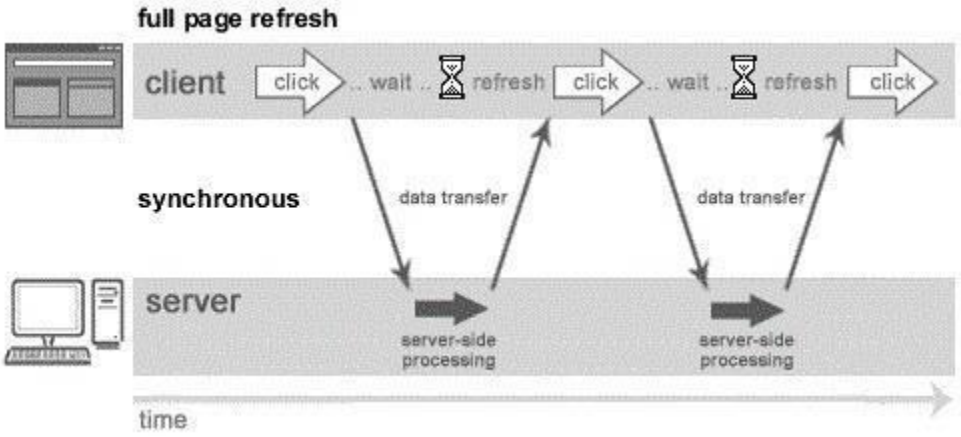
1. Gmail
2. Face book
3. Twitter
4. Google maps
5. YouTube etc.,

Synchronous Vs. Asynchronous Application

Before understanding AJAX, let's understand classic web application model and AJAX Web application model.

- **Synchronous (Classic Web-Application Model)**

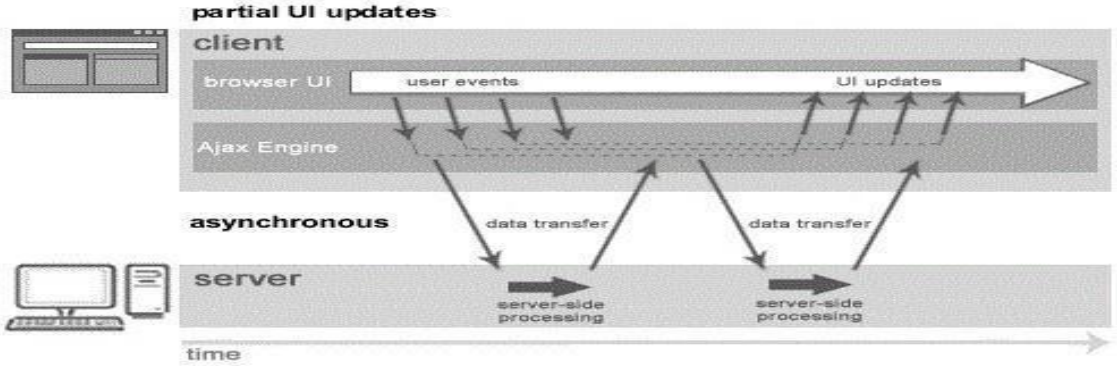
  A synchronous request blocks the client until operation completes i.e. browser is not unresponsive. In such case, JavaScript Engine of the browser is blocked.



As you can see in the above image, full page is refreshed at request time and user is blocked until request completes.

- **Asynchronous (AJAX Web-Application Model)**

  An asynchronous request doesn't block the client i.e. browser is responsive. At that time, user can perform other operations also. In such case, JavaScript Engine of the browser is not blocked.



As you can see in the above image, full page is not refreshed at request time and user gets response from the AJAX Engine. Let's try to understand asynchronous communication by the image given below.

AJAX Components

AJAX is not a technology but group of inter-related technologies. AJAX Technologies includes:

- HTML/XHTML and CSS
- DOM
- XML or JSON(JavaScript Object Notation)
- XMLHttpRequest Object
- JavaScript

- **HTML/XHTML and CSS**

  These technologies are used for displaying content and style. It is mainly used for presentation.

- **DOM**

  It is used for dynamic display and interaction with data.

- **XML or JSON(Javascript Object Notation)**

  For carrying data to and from server. JSON is like XML but short and faster than XML.

- **XMLHttpRequest Object**

  For asynchronous communication between client and server.

- **JavaScript**

  It is used to bring above technologies together. Independently, it is used mainly for client-side validation.

Understanding XMLHttpRequest

It is the heart of AJAX technique. An object of XMLHttpRequest is used for asynchronous communication between client and server.it provides a set of useful methods and properties that are used to send HTTP Request to and retrieve data from the web server. It performs following operations:

1. Sends data from the client in the background
2. Receives the data from the server
3. Updates the webpage without reloading it.

- **Methods of XMLHttpRequest object**

Method	Description
void open(method, URL)	Opens the request specifying get or post method and url.
void open(method, URL, async)	Same as above but specifies asynchronous or not.
void open(method, URL, async, username, password)	Same as above but specifies username and password.
void send()	Sends GET request.
void send(string)	Sends POST request.
setRequestHeader(header,value)	It adds request headers.

Syntax of open() method:

xmlHttp.open("GET","conn.php",tru
e); which takes three attributes

1. An HTTP method such as GET ,POST , or HEAD
2. The URL of the Server resource
3. A boolean Flag that indicates whether the request should be asynchronously(true) or synchronously(false)

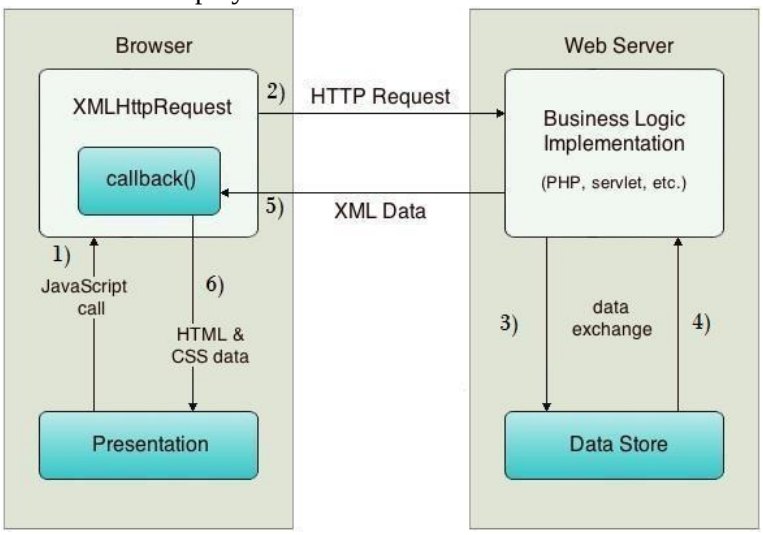
Properties of XMLHttpRequest Object:

Property	Description
readyState	Represents the state of the request. It ranges from 0 to 4. **0 UN INITIALIZED** – After creating XMLHttpRequest Object before calling *open()* method. **1 CONNECTION ESTABLISHED** – open() is called but send() is not called. **2 REQUEST SENT**- send() is called. **3 PROCESSING** - Downloading data; responseText holds the data. **4 DONE** - The operation is completed successfully.
onReadyStateChange	It is called whenever *readystate* attribute changes. It must not be used with synchronous requests.
reponseText	Returns response as TEXT.
responseXML	Returns response as XML

How AJAX Works?

AJAX communicates with the server using XMLHttpRequest object. Let's understand the flow of AJAX with the following figure:

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.



Introduction to Web Services

Technology keep on changing, users were forces to learn new application on continuous basis. With internet, focus is shifting to-wards services based software. Users may access these services using wide range of devices such as PDAs, mobile phones, desktop computers etc. Service oriented software development is possible using man known techniques such as COM, CORBA, RMI, JINI, RPC etc. some of them are capable of delivering services over web and some or not. Most of these technologies uses

particular protocols for communication and with no standardization. **Web service** is the concept of creating services that can be accessed over web. Most of these

What are Web Services?

A web services may be defines as: An application component accessible via standard web protocols. It is like unit of application logic. It provides services and data to remote clients and other applications. Remote clients and application access web services with internet protocols. They use XML for data transport and SOAP for using services. Accessing service is independent of implementation.

With component development model, web service must have following characteristics:

- Registration with lookup service
- Public interface for client to invoke service

Web services should also process following characteristics:

- It should use standard web protocols for communication
- It should be accessible over web
- It should support loose coupling between uncoupled distributed systems

Web services receive information from clients as messages, containing instructions about what client wants, similar to method calls with parameters. These message delivered by web services are encoded using XML.XML enabled web services are interoperable with other web services.

Web Service Technologies:

Wide variety of technologies supports web services. Following technologies are available for creation of web services. These are vendor neutral technologies. They are:

- Simple Object Access Protocol(SOAP)
- Web Services Description Language(WSDL)
- UDDI(Universal Description Discovery and Integration)

Simple Object Access Protocol (SOAP):

SOAP is a light weight and simple XML based protocol. It enables exchange of structured and typed information on web by describing messaging format for machine to machine communication. It also enables creation of web services based on open infrastructure.

- SOAP is application communication protocol designed to communicate via Internet.
- SOAP is a format for sending and receiving messages.
- SOAP provides data transport for Web services.
- SOAP is platform and language-independent.
- SOAP enables client applications to easily connect to remote services and invoke remote methods.
- SOAP can be used in combination with variety of existing internet protocols and formats including HTTP, SMTP etc.

A SOAP message is an ordinary XML document which consists of three parts:

- **SOAP Envelope**: defines what is in message, who is the recipient, whether message is optional or mandatory
- **SOAP Encoding Rules**: defines set of rules for exchanging instances of application defined data types
- **SOAP RPC Representation**: defines convention for representing remote procedure calls

and response

SOAP Message Structure

The following block depicts the general structure of a SOAP message −

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
SOAP- ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <SOAP-ENV:Header>
    ...
  </SOAP-ENV:Header>
 <SOAP-ENV:Body>...
```

```
        <SOAP-ENV:Fault>
        ...
        </SOAP-ENV:Fault>
    ...
```

```
  </SOAP-ENV:Body>
</SOAP_ENV:Envelope>
```

If we are creating web service that offered latest stock quotes, we need to create WSDL file on server that describes service. Client obtains copy of this file, understand contract, create SOAP request based on contract and dispatch request to server using HTTP post. Server validates the request, if found valid executes request. The result which is latest stock price for requested symbol is then returned to client as SOAP response.

Typical SOAP message is shown below:

```
<IVORY:Envelope xmlns:IVORY="http://schemas.xmlsoap.org/soap/envelope"
                IVORY:encodingStyle="http://schemas.xmlsoap.org/soap/encoding
                ">
<IVORY:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
    <symbol>DIS</symbol>
    </m:GetLastTradePrice>
</IVORY:Body>
</IVORY:Envelope>
```

The consumer of web service creates SOAP message as above, embeds it in HTTP POST request and sends it to web service for processing:

```
POST /StockQuote HTTP/1.1
Host:
www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-
URI"
    ....
SOAP Message
....
```

The message now contains requested stock price. A typical returned SOAP message may look like following:

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope" SOAP-ENV:encodingStyle="
http://schemas.xmlsoap.org/soap/encoding" />
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <Price>34.5</Price>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Interoperability:

The major goal in design of SOAP was to allow for easy creation of interoperable distributed web services. Few details of SOAP specifications are open for interpretation; implementation may differ across different vendors. SOAP message though it is conformant XML message, may not strictly follow SOAP specification.

Implementations:

SOAP technology was developed by DevelopMentor, IBM, Lotus, Microsoft etc. More than 50 vendors have currently implemented SOAP. Most popular implementations are by Apache which is open source java based implementation and by Microsoft in .NET platform. SOAP specification has

been submitted to W3C, which is now working on new specifications called XMLP (XML Protocol)

SOAP Messages with Attachments (SwA)

SOAP can send message with an attachment containing of another document or image etc. On Internet, GIF, JPEG data formats are treated as standards for image transmission. Second iteration of SOAP specification allowed for attachments to be combined with SOAP message by using multipart MIME structure. This multi part structure is called as **SOAP Message Package**. This new specification was developed by HP and Microsoft. Sample SOAP message attachment is shown here:

```
MIME-Version: 1.0
Content-Type: Multipart/Related;
boundary=MIME_boundary; type=text/xml;
start=”<myimagedoc.xml@mystie.com>” Content-
Description: This is the optional message description.
--MIME_boundary
Content-Type: text/xml;
charset=UTF-8 Content-Transfer-
Encoding: 8bit
Content-ID: <myimagedoc.xml@mysite.com>
<?xmll version="1.0" ?>
<SOAP-ENV: Envelope xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope”
<SOAP-ENV:Body>
        ...
        <theSignedForm href=”cid:myimage.tiff@mysite.com” />
        ...
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
        --MIME_boundary
        Content-Type:
        image/tiff
        Content-Transfer-Encoding: binary
        Content-ID: <myimagedoc.xml@mysite.com>
        …binary TIFF image…
        --MIME_boundary--
```

Web Services Description Language (WSDL)

WSDL is an XML format for describing web service interface. WSDL file defines set of operations permitted on the server and format that client must follow while requesting service. WSDL file acts like contract between client and service for effective communication between two parties. Client has to request service by sending well formed and conformant SOAP request.

Features of WSDL

- WSDL is an XML-based protocol for information exchange in decentralized and distributed environments.
- WSDL definitions describe how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of Universal Description, Discovery, and Integration (UDDI), an XML- based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL is pronounced as 'wiz-dull' and spelled out as 'W-S-D-L'.

WSDL Document:

WSDL document is an XML document that contains of set of definitions. First we declare name spaces required by schema definition:

```
<schema xmlns=”http://www.w3.org/2000/10/XMLSchema”
xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”
targetNameSpace=http://schemas.xmlsoap.org/wsdl/ elementFormDefault=”qualified”>
```

The root element is definitions as shown below:

```
<wsdl:defiinitions name="nmtoken"? targetNameSpace="uri"?>
        <import namespace="uri" location="uri"/>
<wsdl:documentation ..... />?
        ...
</wsdl:definitions>
```

The *name* attribute is optional and can serve as light weight form of documentation. The *nmtoken* represents name token that are qualified strings similar to CDATA, but character usage is limited to letters, digits, underscores, colons, periods and dashes. A *targetNamespace* may be specified by providing uri. The *import* tag may be used to associate namespace with document locations. Following code segment shows how declared namespace is associated with document location specified in *import* statement:

```
<definitions name="StockQuote"
        targetNameSpace="http://example.com/stockquote/defiinitio
        ns" xmlns:tns="http://example.com/stockquote/definitions"
        xmlns:xsdl="http://example.com/stockquote/schemas"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
<import namespace="http://example.com/stockquote/schemas"
        Location="http://example.com/stockquote/stockquote.xsd
        "/>
```

Finally, optional *wsdl:documentation* element is used for declaring human readable documentation. The element may contain any arbitrary text. There are six major elements in document structure that describes service. These are as follows:

- **Types Element:** it provides definitions for data types used to describe how messages will exchange data. Syntax for types element is as follows:

```
<wsdl:types> ?
        <wsdl:documentation .../>
        <xsd:schema .../>
        <-- extensibility element -->
</wsdl:types>
```

  The *wsdl:documentation* tag is optional as in case of *definitions*. The *xsd* type system may be used to define types in message. WSDL allows type systems to be added via extensibility element.

- **Message Element:** It represents abstract definition of data begin transmitted. Syntax for message element:

```
<wsdl:message name="nktoken"> *
        <wsdl;documentation .../>
            <part name="nmtoken" element="qname"? type="qname"? /> *
</wsdl:message>
```

  The *message name* attribute is used for defining unique name for message with in document scope. The *wsdl:documentation* is optional and may be used for declaring human readable documentation. The message consists of one or more logical parts. The *part* describes logical abstract content of message. Each part consists of name and optional element and type attributes.\

- **Port Type Element:** It defines set of abstract operations. An operation consists of both input and output messages. The *operation* tag defines name of operation, *input* defines input for operation and *output* defines output format for result. The *fault* element is used for describing contents of SOAP fault details element. It specifies abstract message format for error

messages that may be output as result of operation:

```
<wsdl:portType name="nmtoken">*
        <wsdl:documentation ..../>?
    <wsdl:operation name="nmtoken">*
        <wsdl:documentation ..../>?
            <wsdl:input name="nmtoken"? message="qname">?
        <wsdl:documentation ..../>?
        </wsdl:input>
            <wsdl:output name="nmtoken"? message="qname">?
        <wsdl:documentation ..../>?
        </wsdl:output>
            <wsdl:fault name="nmtoken"? message="qname">?
            <wsdl:documentation ..../>?
        </wsdl:fault>
    </wsdl:operation>
</wsdl:portType>
```

- **Binding Element:** It defines protocol to be used and specifies data format for operations and messages defined by particular *portType*. The full syntax for binding is given below:

```
<wsdl:binding name="nmtoken" type="qname"> *
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    <wsdl:operation name="nmtoken">*
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    <wsdl:input> ?
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:input>
    <wsdl:output> ?
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:output>
    <wsdl:fault name="nmtoken"> *
        <wsdl:documentation ..../>?
        <--Extensibility element -->*
    </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

The operation in WSDL file can be document oriented or remote procedure call (RPC) oriented. The style attribute of <*soap:binding*> element defines type of operation. If operation is document oriented, input and output messages will consist of XML documents. If operation is RPC oriented, input message contains operations input parameters and output message contains result of operation.

- **Port Element:** It defines individual end point by specifying single address for binding:

```
<wsdl:port name="nmtoken" binding="qname"> *
    <--Extensibility element (1) -->
</wsdl:port>
```

The *name* attribute defines unique name for port with current WSDL document. The *binding* attribute refers to binding and extensibility element is used to specify address information for port.

- **Service Element:** it aggregates set of related ports. Each port specifies address for binding:

```
<wsdl:service name="nmtoken"> *
        <wsdl:documentation ..../>?
  <wsdl:port name="nktoken" binding="qname"> *
        <wsdl:documentation .../> ?
        <--Extensibility element -->
  </wsdl:port>
        <--Extensibility element -->
 </wsdl:service>
```

Universal Description, Discovery and Integration (UDDI)

UDDI is an XML-based standard for describing, publishing, and finding web services.

- UDDI stands for **Universal Description, Discovery, and Integration.**
- UDDI is a specification for a distributed registry of web services.
- UDDI is a platform-independent, open framework.
- UDDI can communicate via SOAP, CORBA, Java RMI Protocol.
- UDDI uses Web Service Definition Language(WSDL) to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.
- UDDI is an open industry initiative, enabling businesses to discover each other and define how they interact over the Internet.

UDDI has two sections:

- A registry of all web service's metadata, including a pointer to the WSDL description of a service.
- A set of WSDL port type definitions for manipulating and searching that registry.

A business or a company can register three types of information into a UDDI registry. This information is contained in three elements of UDDI. These three elements are:

- White Pages,
- Yellow Pages, and
- Green Pages.

White Pages

White pages contain:

- Basic information about the company and its business.
- Basic contact information including business name, address, contact phone number, etc.
- A Unique identifiers for the company tax IDs. This information allows others to discover your web service based upon your business identification.

Yellow Pages

- Yellow pages contain more details about the company. They include descriptions of the kind of electronic capabilities the company can offer to anyone who wants to do business with it.
- Yellow pages uses commonly accepted industrial categorization schemes, industry codes, product codes, business identification codes and the like to make it easier for companies to search through the listings and find exactly what they want.

Green Pages

Green pages contains technical information about a web service. A green page allows someone to bind to a Web service after it's been found. It includes:

- The various interfaces
- The URL locations
- Discovery information and similar data required to find and run the Web service.

Implementation:

This is global, public registry called UDDI business registry. It is possible for individuals to set up private UDDI registries. The implementations for creating private registries are available from IBM, Idoox etc. Microsoft has developed UDDI SDK that allows visual basic programmer to write program code to interact with UDDI registry. The use of SDK greatly simplifies interaction with registry and shields programmer from local level details of XML and SOAP.

UNIT-1

Introduction to HTML

1. Create a simple HTML page which demonstrates use of three types of lists? (Or)Explain the concept of list?

Ans.

Lists: Lists is one of the most effective ways of structuring a website or its contents.HTML provides three types of lists, they are:

i. Ordered list ii. Unordered list iii. Definition list

i. Ordered Lists: An **ordered list** has a number instead of bullet in front of each list item.

Ordered list must be enclosed within
… tag.

- Create an Ordered List using …….:

Example:

```
<ol type="1">
    <li>Apple</li>
    <li>Orange</li>
    <li>Grapefruit</li>
</ol>
```

- Attribute values for type are 1, A, a, I, or i

Example:

1. Apple
2. Orange
3. Grapefruit

ii. Unordered list: The basic **unordered list** has a bullet in front of each list item.

Everything between tags must be enclosed within
… tag.

Create an Unordered List using ……. :

Example:

```
<ul type="disk">
    <li>Apple</li>
    <li>Orange</li>
    <li>Grapefruit</li>
</ul>
```

- Attribute values for type are: disc, circle or square

Example:

- Apple
- Orange
- Pear

iii. Definition list: These are different to the previous types in that they do not use list items to contain their members.

- Create definition lists using <dl>
  - Pairs of text and associated definition; text is in <dt> tag, definition in <dd> tag

Example:

```
<dl>
    <dt>HTML</dt>
        <dd>A markup language …</dd>
    <dt>CSS</dt>
        <dd>Language used to …</dd>
</dl>
```

- An **Example** which illustrate three list:

```
<ol type="1">
    <li>Apple</li>
    <li>Orange</li>
    <li>Grapefruit</li>
</ol>
<ul type="disc">
    <li>Apple</li>
    <li>Orange</li>
    <li>Grapefruit</li>
</ul>
<dl>
    <dt>HTML</dt>
        <dd>A markup lang…</dd>
</dl>
```

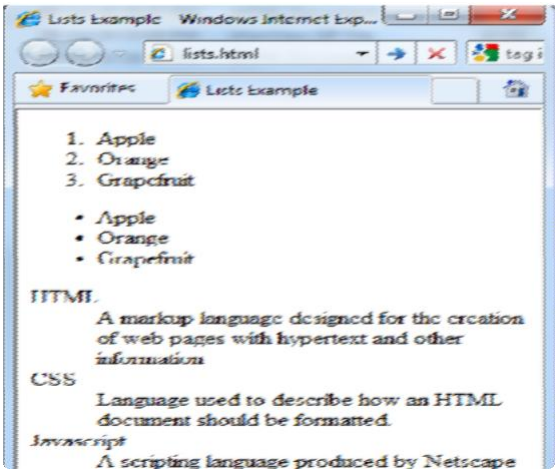


Figure 1. Example for lists

2. Write a short note about CSS and its advantages?

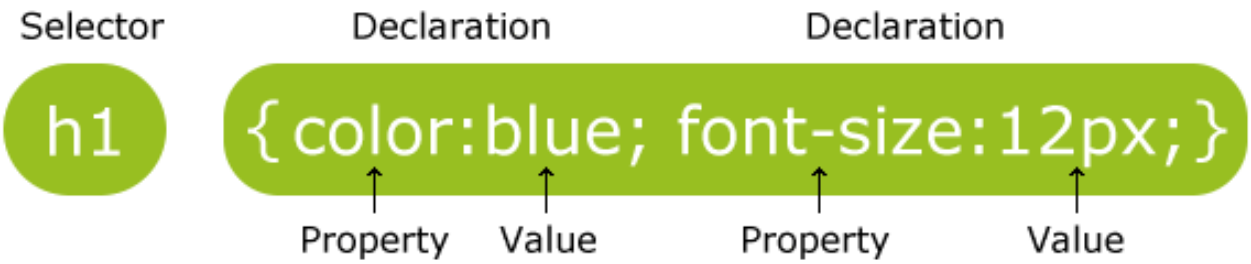
Ans.

Definition:

- Cascading Style Sheets (CSS) form the presentation layer of the user interface.
  - Structure (XHTML)
  - Behavior (Client-Side Scripting)
  - Presentation (CSS)
- Tells the browser agent *how* the element is to be presented to the user.
- A style is simply a set of formatting instructions that can be applied to a piece of text.

CSS Syntax:

- A CSS rule has two main parts: a selector, and one or more declarations:
- The selector is normally the HTML element you want to style.
- Each declaration consists of a property and a value.
- The property is the style attribute you want to change. Each property has a value.



Styling HTML with CSS

- CSS was introduced together with HTML 4, to provide a better way to style HTML elements.

CSS can be added to HTML in the following **3 ways**:

- ❖ Inline - using the style **attribute** in HTML elements
- ❖ Internal - using the <style> **element** in the <head> section
- ❖ External - using an external CSS **file**

i. Inline Styles

- An inline style can be used if a unique style is to be applied to one single occurrence of an element.

  This is a paragraph.

Example:

```
<!DOCTYPE html>
<html>
<body style="background-color:yellow;">
<h2 style="background-color:red;">This is a heading</h2>
<p style="background-color:green;">This is a paragraph.</p>
</body>
</html>
```

Output:

ii. Internal Style Sheet

An internal style sheet can be used if one single document has a unique style. Internal styles are defined in the <head> section of an HTML page, by using the <style> tag, like this:

Example:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {background-color:yellow;}
h1 {color:red;}
h2 {color:blue;}
```

```
p {color:green;}
</style></head>
<body>
    <h1>All header 1 elements will be red</h1>
    <h2>All header 2 elements will be blue</h2>
    <p>All text in paragraphs will be green.</p>
</body></html>
```

Output:

iii. External Style Sheet

An external style sheet is ideal when the style is applied to many pages. With an external style sheet, you can change the look of an entire Web site by changing one file. Each page must link to the style sheet using the <link> tag. The <link> tag goes inside the <head> section:

```
<head>
    <link rel="stylesheet" type="text/css" href="mystyle.css">
</head>
```

HTML Style Tags:

Tag	Description
<style>	Defines style information for a document
<link>	Defines the relationship between a document and an external resource

Advantages of CSS:

- CSS removes the presentation attributes from the structure allowing reusability, ease of maintainability, and an interchangeable presentation layer.
- HTML was never meant to be a presentation language. Proprietary vendors have created tags to add presentation to structure.
  - <font>
  - <b>
  - <i>
- CSS allows us to make global and instantaneous changes easily.

3. Write short notes on HTML hyperlinks? (Or) Discuss the difference between relative and absolute paths in hyperlinks.

Ans.

HTML Hyperlinks (Links)

The HTML **<a> tag** defines a hyperlink.

- A hyperlink (or link) is a word, group of words, or image that you can click on to jump to another document.
- When you move the cursor over a link in a Web page, the arrow will turn into a little hand.
- The most important attribute of the **<a> element is the href** attribute, which indicates the link's destination.

By default, links will appear as follows in all browsers:

- ✓ An unvisited link is underlined and blue
- ✓ A visited link is underlined and purple

An active link is underlined and red

HTML Link Syntax

Link text

Example: Visit W3Schools

HTML Links - the target Attribute

The target attribute specifies where to open the linked document.

The example below will open the linked document in a new browser window or a new tab:

Example

```
<a href="http://www.w3schools.com/" target="_blank">Visit
 W3Schools!</a>
<a href=http://www.telerik.com/  title="Telerik">Link to Telerik
 Web site</a>
```

HTML Links - the id Attribute

The id attribute can be used to create a bookmark inside an HTML document.

Tip: Bookmarks are not displayed in any special way. They are invisible to the reader.

Example

An anchor with an id inside an HTML document:

Useful Tips Section

Create a link to the "Useful Tips Section" inside the same document:

Visit the Useful Tips Section

Or, create a link to the "Useful Tips Section" from another page:

Visit the Useful Tips Section

Two types of Links: Absolute & Relative

- Absolute: shows entire path to file "elephant.jpg"
  (http://www.site.com/web/images/elephant.jpg)
- Relative: seen when on website (zoo.html)
  ("images/elephant.jpg")
- Relative: as if going to page on your site
  Welcome

  Absolute: as if going to another site
  IBM Home

Another type of link is:

- Local: an anchor to and from a particular spot (not on test)
  (Go to Bottom)
  ()

Example:

```
<!DOCTYPE HTML>

<html>

        <head>

                <title>Simple Tags Demo</title>

        </head>

        <body>

<a href=http://www.telerik.com/ title="Telerik site">This is a link.</a>

<br />

        <img src="logo.gif" alt="logo" />

<br />

        <strong>Bold</strong> and <em>italic</em> text.

</body>       </html>
```

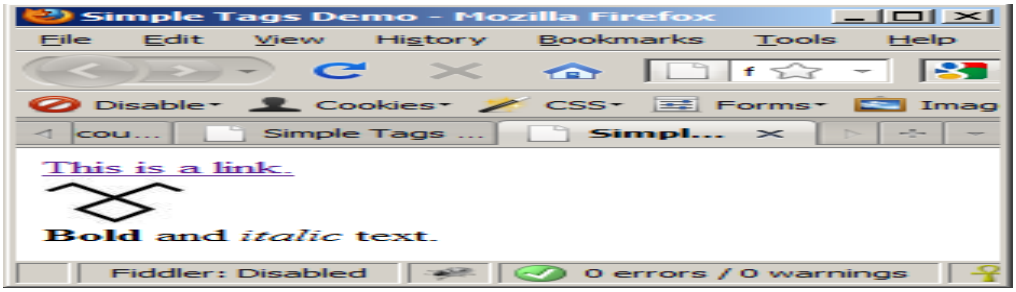


Figure 2. Example for Hyperlinks

4. Write short note on HTML Forms

Ans.

- HTML forms are used to pass data to a server.
- An HTML form can contain input elements like *text fields, checkboxes, radio-buttons, submit buttons* and more.
- A form can also contain *select lists, textarea, fieldset, legend, and label* elements.

The <form> tag is used to create an HTML form:

```
<form>
        …..
    input elements
        ……..
</form>
```

- Forms are the primary method for gathering data from site visitors
- Create a form block with: <form></form>
- **Example:**

```
<form name="myForm" method="post"    action="path/to/some-script.php">

  ...

</form>
```

Syntax:

```
<input type="text"

    name="string" [value="string"] [checked] [size="n"] [maxlength="n"]
        [src="url"] [align="top"|"bottom"|"middle"|"left"|"right"]>
```

HTML Forms - The Input Element

Text Fields

Defines a one-line input field that a user can enter text into:

Example:

```
<form>
First name: <input type="text" name="firstname"><br>
Last name: <input type="text" name="lastname">
</form>
```

Note: The form itself is not visible. Also note that the default width of a text field is 20 characters.

2. Password Field

Example:

```
<form>
Password: <input type="password" name="pwd">
</form>
```

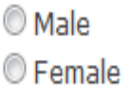
Note: The characters in a password field are masked (shown as asterisks or circles).

3. Radio Buttons

<input type="radio"> defines a radio button. Radio buttons let a user select ONLY ONE of a limited number of choices:

Example:

```
<form>
<input type="radio" name="gender" value="male">Male<br>
<input type="radio" name="gender" value="female">Female
</form>
```

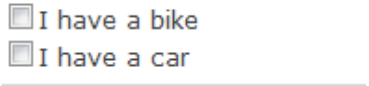


4. Checkboxes

<input type="checkbox"> defines a checkbox. Checkboxes let a user select ZERO or MORE options of a limited number of choices.

Example:

```
<form>
<input type="checkbox" name="vehicle" value="Bike">I have a bike<br>
<input type="checkbox" name="vehicle" value="Car">I have a car
</form>
```



5. Submit Button

<input type="submit"> defines a submit button.

A submit button is used to send form data to a server. The data is sent to the page specified in the form's action attribute. The file defined in the action attribute usually does something with the received input:

Example:

<form name="input" action="demo_form_action.asp‟ method="get">

Username: <input type="text" name="user">

<input type="submit" value="Submit">

</form>

6. Button

Example:

<!DOCTYPE html>

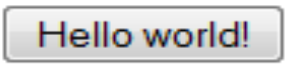
<html>

<body>

<form action="">

<input type="button" value="Hello world!">

</form> </body></html>



7. Reset button – brings the form to its initial state

<input type="reset" name="resetBtn" value="Reset the form" />

8. Image button – acts like submit but image is displayed and click coordinates are sent

<input type="image" src="submit.gif" name="submitBtn" alt="Submit" />

9. Hidden fields contain data not shown to the user:

<input type="hidden" name="Account" value="This is a hidden text field" />

Fieldset:

- Fieldsets are used to enclose a group of related form fields:

<form method="post" action="form.aspx">

<fieldset> <legend>Client Details</legend>

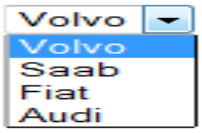
```
    <input type="text" id="Name" />
     <input type="text" id="Phone" />
   </fieldset>
</form>
```

Other form controls:

- Dropdown menus:

Example:

```
<!DOCTYPE html>
<html>
<body>
<form action="">
<select name="cars">
        <option value="volvo">Volvo</option>
        <option value="saab">Saab</option>
        <option value="fiat">Fiat</option>
        <option value="audi">Audi</option>
</select>
</form>
</body>
</html>
```



Labels:

- Form labels are used to associate an explanatory text to a form field using the field's ID.

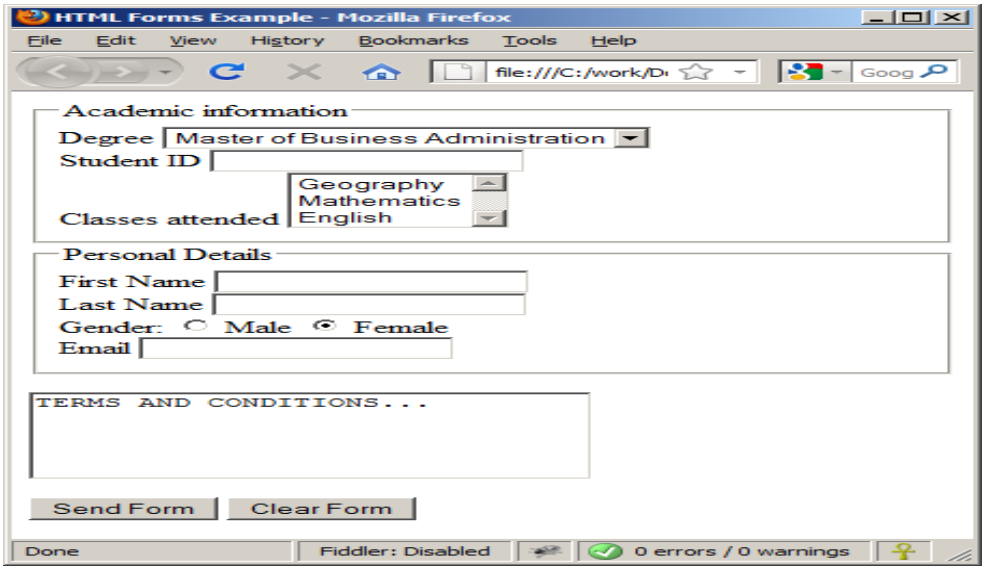
```
<label for="fn">First Name</label>
<input type="text" id="fn" />
```

Example: form.html

```
<form method="post" action="apply-now.php">
    <input name="subject" type="hidden" value="Class" />
<fieldset><legend>Academic information</legend>
    <label for="degree">Degree</label>
```

```
                <select name="degree" id="degree">
                        <option value="BA">Bachelor of Art</option>
                        <option value="BS">Bachelor of Science</option>
                        </select>  <br />
                <label for="studentid">Student ID</label>
        <input type="password" name="studentid" />
  </fieldset>
  <fieldset><legend>Personal Details</legend>
          <label for="fname">First Name</label>
         <input type="text" name="fname" id="fname" /> <br />
         <label for="lname">Last Name</label>
          <input type="text" name="lname" id="lname" /><br />
          Gender:
         <input name="gender" type="radio" id="gm" value="m" />
                 <label for="gm">Male</label>
                <input name="gender" type="radio" id="gf" value="f" />
                        <label for="gf">Female</label> <br />
                        <label for="email">Email</label>
                <input type="text" name="email" id="email" />
        </fieldset>
          <p> <textarea name="terms" cols="30" rows="4"
          readonly="readonly">TERMS AND CONDITIONS...</textarea> </p>
  <p><input type="submit" name="submit" value="Send Form" />
           <input type="reset" value="Clear Form" /> </p>
</form>
```

Output:



5. Explain how a basic table is created in HTML.

Ans:

- Tables represent tabular data
  - A table consists of one or several rows
  - Each row has one or more columns
- Tables comprised of several core tags:

  <table></table>: begin / end the table
  <tr></tr>: create a table row
  <td></td>: create tabular data (cell)

- Tables should not be used for layout. Use CSS floats and positioning styles instead.
- Start and end of a table

  <table>.. </table>
- Start and end of a row

  <tr>...</tr>
- Start and end of a cell in a row

  <td>...</td>

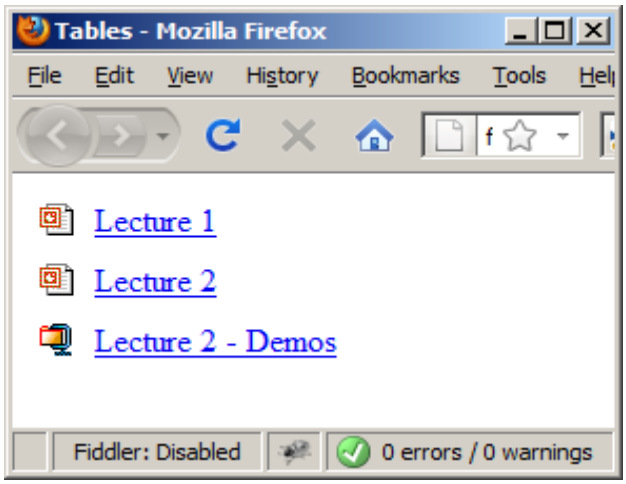
Example:

```
<table cellspacing="0" cellpadding="5">

 <tr> <td><img src="ppt.gif"></td>

<td><a href="lecture1.ppt">Lecture 1</a></td></tr>

<tr> <td><img src="ppt.gif"></td>

<td><a href="lecture2.ppt">Lecture 2</a></td> </tr>
```

<tr><td><img src="zip.gif"></td>

<td> Lecture 2 - Demos</td></tr></table>

Output:



Note:

Have a glance on the topics:

Basic HTML, Class and Id selectors, HTML tables, Frames, layers.

UNIT -2

JAVA SCRIPT

1. Write a javascript logic for validating email-id of a registration form.

Ans.
JavaScript is a **scripting** language. A scripting language is a lightweight **programming** language.

JavaScript can manipulate the DOM (change HTML contents).

It allows interactivity such as:

- Implementing form validation
- React to user actions, e.g. handle keys
- Changing an image on moving mouse over it
- Sections of a page appearing and disappearing
- Content loading and changing dynamically
- Performing complex calculations
- Custom HTML controls, e.g. scrollable table
- Implementing AJAX functionality

In HTML, JavaScripts must be inserted between <script> and </script> tags. The <script> and </script> tells where the JavaScript starts and ends.

JavaScripts can be put in the <body> and in the <head> section of an HTML page.

JavaScript is often used to validate input.

The following is the java script code that validates the email id of the registration form:

```
<html>   <head>

        <script language="javascript">

function validate()

{               /*E-mail validation*/

                  var e=f5.t3.value;

                  var atpos=e.indexOf("@");

                  var dotpos=e.lastIndexOf(".");

                  if(atpos<1 || dotpos<atpos+2 || dotpos+2>=e.length)

                  alert("Not a valid e-mail address");
```

```
                else
                        window.alert("your mail id is:"+e);
}
</script>
</head>
<body bgcolor="pink">
                        <form name="f5">
                        <fieldset>
                        <legend>Registration Details</legend>
                        <center>
                        <table>       <tr>
                        <td>Email: </td><td><input type="text" name=t3></td>
                                        </tr>
/* <tr>
<td>mobile no:</td><td><input type="text" name=t4></td>
</tr>
<tr>
       <td>          DateOfBirth: </td>
<td> <select name=day>
<option value=1>1 </option>
<option value=2>2 </option>
<option value=3>3 </option>
<option value=4>4 </option>
<option value=5>5 </option>
<option value=6>6 </option>
<option value=7>7 </option>
<option value=8>8 </option>
```

<option value=9>9 </option>

<option value=10>10</option>

<option value=11>11</option>

<option value=12>12</option>

<option value=13>13</option>

<option value=14>14</option>

<option value=15>15</option>

<option value=16>16</option>

<option value=17>17</option>

<option value=18>18</option>

<option value=19>19</option>

<option value=20>20</option>

<option value=21>21</option>

<option value=22>22</option>

<option value=23>23</option>

<option value=24>24</option>

<option value=25>25</option>

<option value=26>26</option>

<option value=27>27</option>

<option value=28>28</option>

<option value=29>29</option>

<option value=30>30</option>

<option value=31>31</option>

</select>

<select name=month>

<option value=Jan>Jan</option>

<option value=Feb>Feb</option>

```
<option value=Mar>Mar</option>
<option value=Apr>Apr</option>
<option value=May>May</option>
<option value=Jun>Jun</option>
<option value=Jul>Jul</option>
<option value=Aug>Aug</option>
<option value=Sep>Sep</option>
<option value=Oct>Oct</option>
<option value=Nov>Nov</option>
<option value=Dec>Dec</option>
</select>
<select name=year>
<option value=1995>1995</option>
<option value=1996>1996</option>
<option value=1997>1997</option>
<option value=1998>1998</option>
<option value=1999>1999</option>
<option value=2000>2000</option>
<option value=2001>2001</option>
</select>
        </td>
                </tr> */
        </table>
<input type="submit" value="submit" onClick=validate()>
                <input type="reset" value="reset">
                </center>
                        </fieldset></form></body></html>
```

2. Write a JavaScript code to validate user id and password input fields with the following constraints:

User id-It should contain combination of alphabets, numbers and _. It should not allow spaces and special symbols.
Password-It should not be less than 8 characters in length.

Ans.
JavaScript is a **scripting** language. A scripting language is a lightweight **programming** language.

JavaScript can manipulate the DOM (change HTML contents).

It allows interactivity such as:

- Implementing form validation
- React to user actions, e.g. handle keys
- Changing an image on moving mouse over it
- Sections of a page appearing and disappearing
- Content loading and changing dynamically
- Performing complex calculations
- Custom HTML controls, e.g. scrollable table
- Implementing AJAX functionality

In HTML, JavaScripts must be inserted between <script> and </script> tags. The <script> and </script> tells where the JavaScript starts and ends.

JavaScripts can be put in the <body> and in the <head> section of an HTML page.

JavaScript is often used to validate input.

The following is the java script code that validates the user id and passwords of the registration form:

```
<html>

<head>

<script language="javascript">

function validate()

{

  var f=f6.t1.value;

  var l=f6.t2.value;

  var u=f6.t3.value;
```

```
  var p=f6.t4.value;
  var re1=/\d|\W|_/i;
  var re2=/\d|\w|/;
  if(f.length<3)
  {
      var x=document.getElementById("first");
     x.innerHTML="First name should More than 3 characters";
  }
else
  if(re1.test(f))
  {
       var x=document.getElementById("first");
     x.innerHTML="First name should only alphabets";
  }
else
  if(l.length<3)
  {
    var x=document.getElementById("last");
     x.innerHTML="enter More than 3 characters";
  }
else
  if(re1.test(l))
  {
       var x=document.getElementById("last");
     x.innerHTML="Last name should only alphabets";
  }
else
```

```
  if(re2.test(u))
   {
        var x=document.getElementById("uid");
     x.innerHTML="user name should only alphanumerics and _";
   }
else
      if(p.length<8)
   {
        var x=document.getElementById("pswd");
     x.innerHTML="password should be greater than 8";
   }
else
window.alert("successfully registered");
}
</script>
</head>
<body bgcolor="pink">
<form name="f6">
<fieldset>
<legend>Registration Details</legend>
<center>
<table>
<tr>
<td>FirstName:</td><td><input type="text" name=t1><span id="first"
style="color:red"></span></td>
</tr>
<tr>
```

```
<td>LastName:</td><td><input type="text" name=t2><span id="last"
style="color:red"></span></td>

</tr>

<tr>

<td>User id:</td><td><input type="text" name=t3><span id="uid"
style="color:red"></span></td>

</tr>

<tr>

<td>Password:</td><td><input type="password" name=t4><span id="pswd"
style="color:red"></span></td>

</tr>

</table>

<input type="button" value="submit" onClick=validate()>

<input type="reset" value="reset">

</center>

</fieldset>

</form>

</body>

 </html>
```

3. Functions in Java Script

Ans.

JavaScript Functions

Often, JavaScript code is written to be executed when an **event** occurs, like when the user clicks a button.

JavaScript code inside a **function** , can be invoked, when an event occurs.

Invoke a function = call upon a function (ask for the code in the function to be executed).

Function Declarations

```
function functionName(parameters) {
  code to be executed
}
```

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

Example

```
function myFunction(a, b) {
                                    return a * b;
                                  }
```

Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

Example

```
var x = function (a, b) {return a * b};
```

Function Parameters and Arguments

functionName(*parameter1, parameter2, parameter3*) {
  code to be executed
}

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** passed to (and received by) the function.

Invoking a JavaScript Function

The code in a function is not executed when the function is defined**. It is executed when the function is** invoked**.**

```
function myFunction(a, b) {
                                    return a * b;
                                  }
myFunction(10, 2);          // myFunction(10, 2) will return 20
```

4. Write a Java Script for password matching of two input fields.

Ans.
JavaScript is a **scripting** language. A scripting language is a lightweight **programming** language.

JavaScript can manipulate the DOM (change HTML contents).

It allows interactivity such as:

- Implementing form validation
- React to user actions, e.g. handle keys
- Changing an image on moving mouse over it

- Sections of a page appearing and disappearing
- Content loading and changing dynamically
- Performing complex calculations
- Custom HTML controls, e.g. scrollable table
- Implementing AJAX functionality

In HTML, JavaScripts must be inserted between <script> and </script> tags. The <script> and </script> tells where the JavaScript starts and ends.

JavaScripts can be put in the <body> and in the <head> section of an HTML page.

JavaScript is often used to validate input.

The following is the Java Script code for matching two password input fields:

```
<html>
        <head>
        <script type="text/javascript">
         function validation()
         {
         var c=document.f1.pass.value;
        if(c=="")
         {
                alert("PLease enter your Password");
                document.f1.pass.focus();
                return false;
        }
        var d=document.f1.cpass.value;
        if(d=="")
        {

                alert("PLease enter your Confirm Password");
                document.f1.cpass.focus();

                return false;
        }
        if(c!=d)
         {
                alert("Password and confirm Password Mismatched");
                return false;
        }
        }
         </script>
        </head>

        <body>
         <form name="f1" method="post" action="">
                 Password: <input type="password" name="pass" />
                 Confirm Password: <input type="password" name="cpass" />
                 <button type="reset" name="reset" >Reset</button>
                <buttontype="submit" name="submit"  onClick=”validation()”>
```

```
            Submit</button>

        </form>
  </body>

</html>
```

5. How is programming made easier in Javascript? Also mention the benefits and problems with Javascript.

Ans:

- JavaScript is a front-end scripting language developed by Netscape for dynamic content
  - Lightweight, but with limited capabilities
  - Can be used as object-oriented language
- Client-side technology
  - Embedded in your HTML page
  - Interpreted by the Web browser
- Simple and flexible
- Powerful to manipulate the DOM
  - Implementing JavaScript into Web pages
    - In <head> part
    - In <body> part
    - In external .js file

JavaScript Syntax

- JavaScript is a scripting language. It is a lightweight, but powerful, programming language.
- **Syntax definition:** "The principles by which sentences are constructed in a language."
- The sentences of a programming language are called *computer statements*, or just *statements*.
- The methods of the class are invoked by the Objects.

  Ex:- document.writeln("hello");
- HTML tags can be used within the " " to invoke the functionality of the tag

  Ex:- **document.writeln**("<h1 style="color:red">hello</h1>");
- User Input :- To take input from the user prompt() is used

  Synatx:- **prompt**("Enter the value",default value");

- Usage: - window.prompt("enter x","");

Advantages

JavaScript allows interactivity such as:

- Implementing form validation
- React to user actions, e.g. handle keys
- Changing an image on moving mouse over it
- Sections of a page appearing and disappearing
- Content loading and changing dynamically
- Performing complex calculations
- Custom HTML controls, e.g. scrollable table
- Implementing AJAX functionality

Disadvantages

- **Security.** Because the code executes on the users' computer, in some cases it can be exploited for malicious purposes. This is one reason some people choose to disable JavaScript.
- **Reliance on End User.** JavaScript is sometimes interpreted differently by different browsers. Whereas server-side scripts will always produce the same output, client-side scripts can be a little unpredictable.

Write Short notes on:

6. JavaScript Variables

- In a programming language (and in normal algebra), named variables store data values.
- JavaScript uses the **var** keyword to define variables, and an equal sign to assign values to variables (just like algebra):

Example

```
var x, length;
    x = 5;
    length = 6;
```

Note:

- A variable can have **variable values** during the execution of a JavaScript. A literal is always a **constant value**.
- A variable is a **name**. A literal is **value**.

Example on Variables:

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
```

```
<script>
        var length;
        length = 6;
        document.getElementById("demo").innerHTML = length;
</script>
</body>
</html>
```

7. String Manipulation

Method	Description
charAt()	Returns the character at the specified index (position)
concat()	Joins two or more strings, and returns a copy of the joined strings
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
match()	Searches a string for a match against a regular expression, and returns the matches
replace()	Searches a string for a value and returns a new string with the value replaced
search()	Searches a string for a value and returns the position of the match
slice()	Extracts a part of a string and returns a new string
split()	Splits a string into an array of substrings
substr()	Extracts a part of a string from a start position through a number of characters
substring()	Extracts a part of a string between two specified positions
toLowerCase()	Converts a string to lowercase letters
toString()	Returns the value of a String object
toUpperCase()	Converts a string to uppercase letters
trim()	Removes whitespace from both ends of a string
valueOf()	Returns the primitive value of a String object

8. Arrays

JavaScript arrays are used to store multiple values in a single variable.

Displaying Arrays

```
<!DOCTYPE html>

<html>

<body>
```

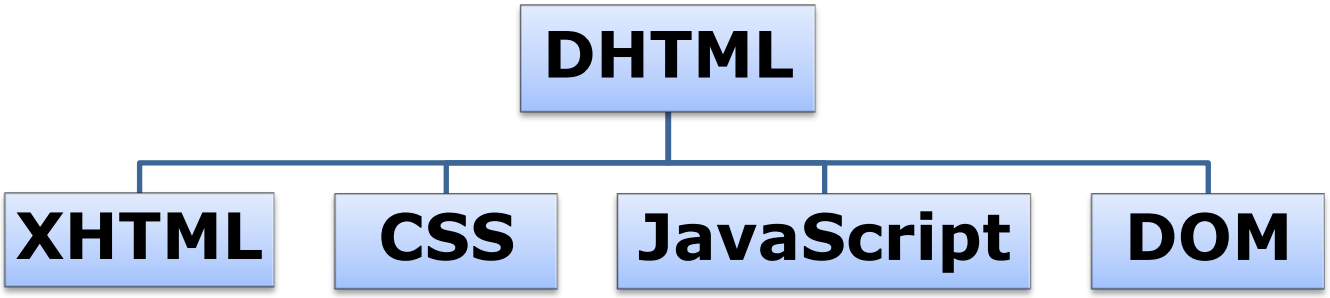
```
<p id="demo"></p>

<script>

var cars = ["Saab", "Volvo", "BMW"];

document.getElementById("demo").innerHTML = cars[0];

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

Table 1. Array of Objects

Method	Description
concat()	Joins two or more arrays, and returns a copy of the joined arrays
indexOf()	Search the array for an element and returns its position
join()	Joins all elements of an array into a string
lastIndexOf()	Search the array for an element, starting at the end, and returns its position
pop()	Removes the last element of an array, and returns that element
push()	Adds new elements to the end of an array, and returns the new length
reverse()	Reverses the order of the elements in an array
shift()	Removes the first element of an array, and returns that element
slice()	Selects a part of an array, and returns the new array
sort()	Sorts the elements of an array
splice()	Adds/Removes elements from an array
toString()	Converts an array to a string, and returns the result
unshift()	Adds new elements to the beginning of an array, and returns the new length
valueOf()	Returns the primitive value of an array

9. Introduction to DHTML

- Dynamic HTML (DHTML)
  - Makes possible a Web page to react and change in response to the user's actions
- DHTML = HTML + CSS + JavaScript



DTHML = HTML + CSS + JavaScript

- HTML defines Web sites content through semantic tags (headings, paragraphs, lists, …)
- CSS defines 'rules' or 'styles' for presenting every aspect of an HTML document
  - Font (family, size, color, weight, etc.)
  - Background (color, image, position, repeat)
  - Position and layout (of any object on the page)
- JavaScript defines dynamic behavior
  - Programming logic for interaction with the user, to handle events, etc.

Note:

Have a glance on the topics:

Objects in Javascript

UNIT-3

XML

1. Explain about Document Type Definition and explain about the elements?

Ans.

XML stands for EXtensible Markup Language.XML was designed to describe data.XML is a software- and hardware-independent tool for carrying information.XML is easy to learn.XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive.

XML documents are composed of three things:

1. Elements
2. Control information and
3. Entities

Elements:

The main building blocks of both XML and HTML documents are elements.

In XML, all elements **must** be properly nested within each other:

Ex: This text is bold and italic

XML documents form a tree structure that starts at "the root" and branches to "the leaves". XML documents must contain one element that is the **parent** of all other elements. This element is called the **root** element.

```
<root>
    <child>
        <subchild>.....</subchild>
    </child>
</root>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
        <to> Tove</to>
        <from>Jani</from>
        <heading>Reminder</heading>
   <body>Don't forget me this weekend!</body>
</note>
```

Document definition:A document definition is the easiest way to provide a reference to the legal elements and attributes of a document.A document definition also provides a common reference that many users (developers) can share.

There are different types of document definitions that can be used with XML:

1. The original Document Type Definition (DTD)
2. The newer, and XML based, XML Schema

i. DTD:

An XML document validated against a DTD is "Well Formed" and "Valid".
A well formed document id one which follows all of the rules of XML. Tags are matched and do not overlap, empty elements are ended properly and the document contains an XML declaration.
A valid XML document has its own DTD. The document is well formed but also conforms to the rules set out in the DTD.

Use of DTD:

- With a DTD, each of your XML files can carry a description of its own format.
- With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.
- Your application can use a standard DTD to verify that the data you receive from the outside world is valid.
- You can also use a DTD to verify your own data.

ii. DTD - XML Building Blocks:

The Building Blocks of XML Documents are seen from a DTD point of view. All XML documents (and HTML documents) are made up by the following building blocks:

- Elements
- Attributes
- Entities
- PCDATA
- CDATA

Elements:

In a DTD, XML elements are declared with an element declaration with the following syntax:

```
<!ELEMENT element-name category>
        Or
<!ELEMENT element-name (element-content)>
```

In the above syntax, element-name is nothing but the name of the tag used and the category field may be of among different types specified below:

a. Empty Elements
   Empty elements are declared with the category keyword EMPTY
   Syntax: <!ELEMENT element-name EMPTY>
b. Elements with Parsed Character Data
   Elements with only parsed character data are declared with #PCDATA inside parentheses:
   Syntax: <!ELEMENT element-name (#PCDATA)>
c. Elements with any Contents
   Elements declared with the category keyword ANY, can contain any combination of parsable data
   Syntax: <!ELEMENT element-name ANY>
d. Elements with Children (sequences)
   Elements with one or more children are declared with the name of the children elements inside parentheses

Syntax: <!ELEMENT element-name (child1)>

or

<!ELEMENT element-name (child1,child2,...)>

e. Declaring either/or Content

Example: !ELEMENT note (to,from,header,(message|body))>

f. Declaring Mixed Content

Example:<!ELEMENT note (#PCDATA|to|from|header|message)*>

Example1

XML Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

DTD Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT note (to,from,heading,body)>//since for "note" there are
4 children:to, from, heading, body. Hence it is specified like this.
<!ELEMENT to (#PCDATA)>//this if for "to" node and here the
category is of type "b" as given above.
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Based on the occurrence of the Element in XML, the syntax used in the DTD may change as shown below.

a. For Only One Occurrence of an Element: <!ELEMENT element-name (child-name)>
b. Minimum One Occurrence of an Element: <!ELEMENT element-name (child-name+)>
c. Zero or More Occurrences of an Element: <!ELEMENT element-name (child-name*)>
d. Zero or One Occurrences of an Element : <!ELEMENT element-name (child-name?)>

Example2

XML example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE address SYSTEM "name.dtd">
<address>
        <name>
                <fname>shaik</fname>
                <lname>K K Baseer</lname>
        </name>
```

```
        <name>
            <fname>shaik</fname>
            <lname>K K Baseer-SVEC</lname>
        </name>
    </address>
```

DTD Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT address (name+)>//Since the element "name" in the above example are used twice, hence "name+" is used in the category field.
<!ELEMENT name ( fname, lname ) >//fname and lname are children to parent element name.
<!ELEMENT fname (#PCDATA)>
<!ELEMENT lname (#PCDATA)>
```

2. Explain about DOM?

Ans.

Introduction to parsers

- The word *parser* comes from compilers
- In a compiler, a parser is the module that reads and interprets the programming language.
- In XML, a parser is a software component that sits between the application and the XML files.
- It reads a text-formatted XML file or stream and converts it to a document to be manipulated by the application.

Types of parsers:

a. Based on well formed and valid xml documents:
   1. Validating parsers:validating parsers knows not only following syntactical rules but also how to validate documents against their DTDs
   2. Non-validating parsers: Parsers enforce only syntactical rules.

b. Based on how the elements parsed:
   1. Event based parsers:

      An event-based API reports parsing events (such as the start and end of elements) directly to the application through callbacks.
      - The application implements handlers to deal with the different events.
      - Event-based parsers deal generally used for large documents.
      - Event-based parsers are more complex and give hard time for the programmer.
      - SAX is an example of event based parser.

   2. Tree based parsers:

      These map an XML document into an internal tree structure, and then allow an application to navigate that tree.

- Ideal for browsers, editors, XSL processors.
- Tree-based parsers deal generally small documents.
- Tree-based parsers are generally easier to implement.
- DOM is an example of tree based parsers.

DOM:

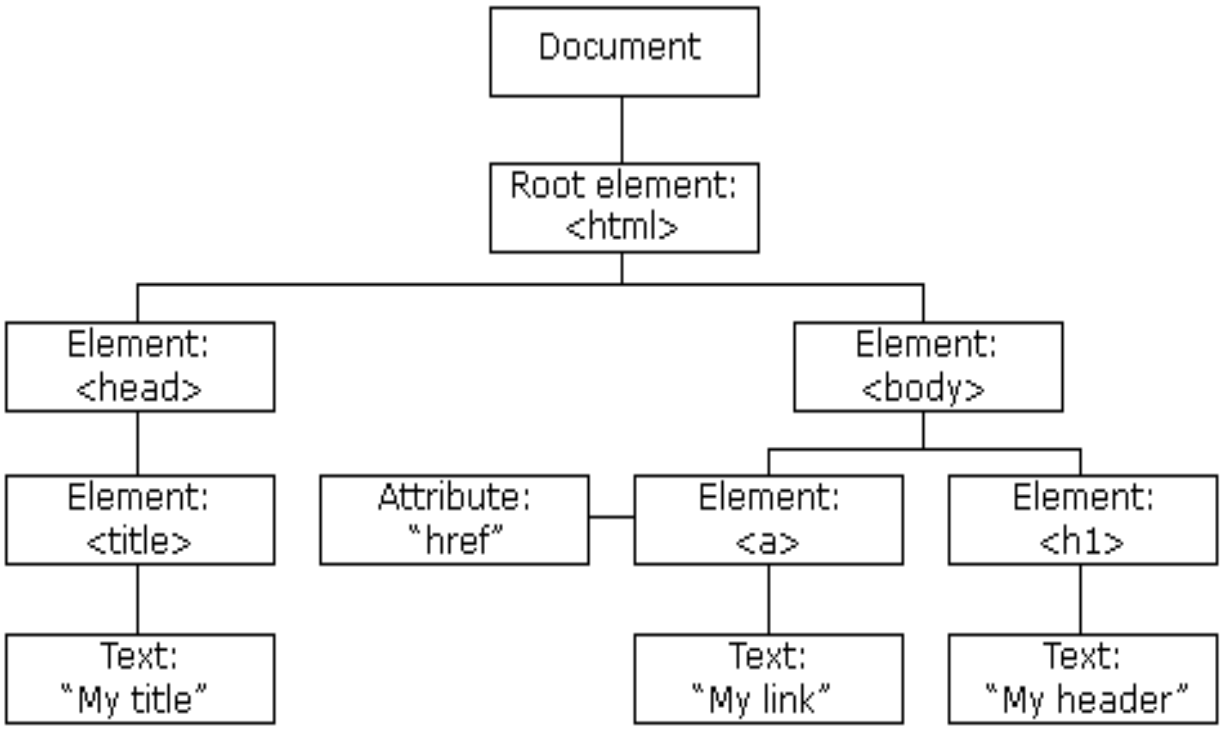
The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated

Properties of DOM:

- Programmers can build documents, navigate their structure, and add, modify, or delete elements and content.
- DOM provides a standard programming interface that can be used in a wide variety of environments and applications.
- structural isomorphism.

The Document Object Model is not a binary specification. The Document Object Model is not a way of persisting objects to XML or HTML. The Document Object Model does not define "the true inner semantics" of XML or HTML. The Document Object Model is not a set of data structures, it is an object model that specifies interfaces.

Generating a DOM Tree for XML:



Example:

XML:

```
<?xml version="1.0"?>
<products>
    <product>
        <name>XML Editor</name>
        <price>499.00</price>
    </product>
    <product>
        <name>DTD Editor</name>
        <price>199.00</price>
    </product>
    <product>
        <name>XML Book</name>
        <price>19.99</price>
    </product>
    <product>
        <name>XML Training</name>
        <price>699.00</price>
    </product>
</products>
```

DOM Tree:

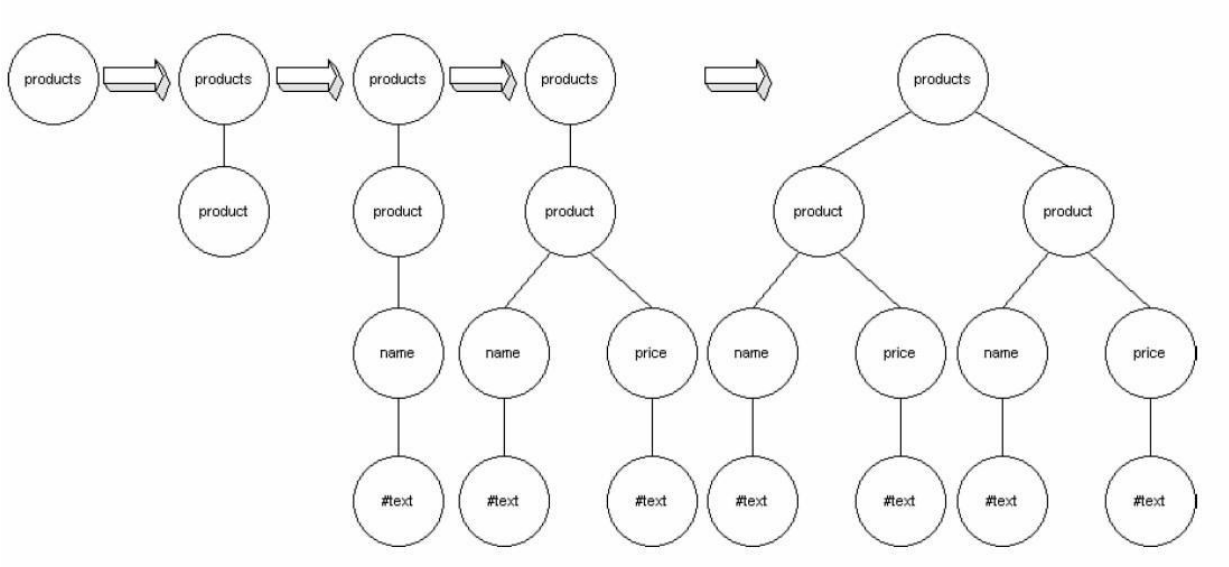


Figure 1: Building the tree of objects

DOM is used for manipulating the HTML elements like:

1. Creating New HTML Elements (Nodes): By using "appendChild()" method

   Example:

```
<!DOCTYPE html>
<html>
```

```
<body>
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
<script>
varpara = document.createElement("p");
var node = document.createTextNode("This is new.");
para.appendChild(node);
var element = document.getElementById("div1");
element.appendChild(para);
</script>
</body>
</html>
```

2. Removing Existing HTML Elements: By using "removeChild()" method.

   Example:

```
<!DOCTYPE html>
<html>
<body>
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
<script>
var parent = document.getElementById("div1");
var child = document.getElementById("p1");
parent.removeChild(child);
</script>
</body>
</html>
```

3. Replacing HTML Elements: By using "replaceChild()" method

   Example:

```
<!DOCTYPE html>
<html>
<body>
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
<script>
var parent = document.getElementById("div1");
var child = document.getElementById("p1");
varpara = document.createElement("p");
var node = document.createTextNode("This is new.");
para.appendChild(node);
```

```
parent.replaceChild(para,child);
</script>
</body>
</html>
```

4. Finding HTML Elements:
   - Finding HTML elements by id
     document.getElementById()
   - Finding HTML elements by tag name
     document.getElementsByTagName()
   - Finding HTML elements by class name
     document.getElementsByClassName()

Limitations:

- Memory consumption is very high
- Very complex for large xml documents

3. Explain about SAX?

Ans.

Introduction to parsers

- The word *parser* comes from compilers
- In a compiler, a parser is the module that reads and interprets the programming language.
- In XML, a parser is a software component that sits between the application and the XML files.
- It reads a text-formatted XML file or stream and converts it to a document to be manipulated by the application.

Types of parsers:

a. Based on well formed and valid xml documents:
   1. Validating parsers:validating parsers knows not only following syntactical rules but also how to validate documents against their DTDs
   2. Non-validating parsers: Parsers enforce only syntactical rules.

b. Based on how the elements parsed:
   1. Tree based parsers:
      These map an XML document into an internal tree structure, and then allow an application to navigate that tree.
      - Ideal for browsers, editors, XSL processors.
      - Tree-based parsers deal generally small documents.
      - Tree-based parsers are generally easier to implement.
      - DOM is an example of tree based parsers.

   2. Event based parsers:
      An event-based API reports parsing events (such as the start and end of elements) directly to the application through callbacks.
      - The application implements handlers to deal with the different events.

- Event-based parsers deal generally used for large documents.
- Event-based parsers are more complex and give hard time for the programmer.
- SAX is an example of event based parser.

SAX:
SAX (the Simple API for XML) is an event-based parser for xml documents.

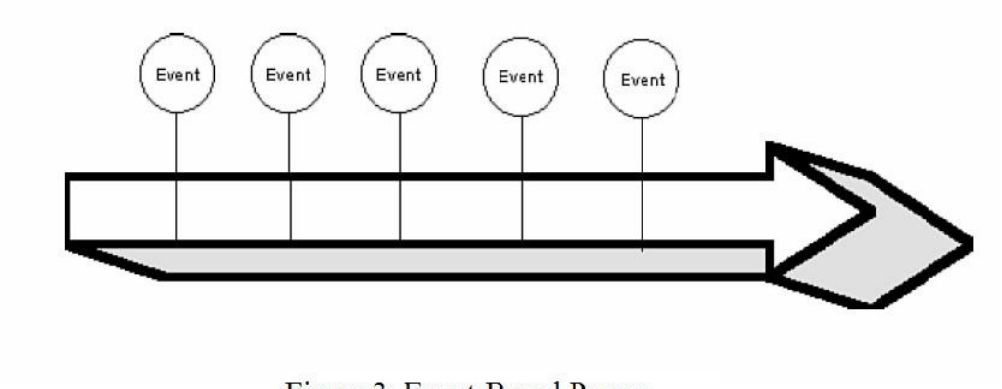


Figure 3: Event-Based Parser

The parser tells the application what is in the document by notifying the application of a stream of parsing events. Application then processes those events to act on data.

History:
SAX is a common, event-based API for parsing XML documents, developed as a collaborative project of the members of the XML-DEV discussion under the leadership of David Megginson.

Versions:

SAX 1 introduced in May 1998,
SAX 2.0 introduced in May 2000.

For applications that are not so XML-centric, an object-based interface is less appealing. Event-based interface consumes fewer resources than an object-based one. With an event-based interface, the application can start processing the document as the parser is reading it.

- Parser events are similar to user-interface events such as ONCLICK (in a browser) or AWT events (in Java).
- Events alert the application that something happened and the application might want to react.
  - When the parser finds the starting tag a certain action is done.
    Ex:
    function startElement(…)
    { //do something }
  - When the parser finds the ending tag some other action is done.
    Ex:
    function endElement(…)
    { //do something }
- Efficiency of SAX: Lower level than object-based interface

- Supported in:
  Java
  Perl
  C++
  Python

Components of SAX:

1. Element opening tags
2. Element closing tags
3. Content of elements
4. Entities
5. Parsing errors

Example:

```
<DocumentElementparam="value">
<FirstElement>&#xb6; Some Text </FirstElement>
<?some_pisome_attr="some_value"?>
<SecondElement param2="something"> Pre-Text
<Inline>Inlined text</Inline> Post-text.
</SecondElement>
</DocumentElement>
```

SAX Parser:

This XML document, when passed through a SAX parser, will generate a sequence of events like the following:

- XML Element start, named *DocumentElement*, with an attribute *param* equal to "value"
- XML Element start, named *FirstElement*
- XML Text node, with data equal to "¶ Some Text"
- XML Element end, named *FirstElement*
- Processing Instruction event, with the target *some_pi* and data *some_attr="some_value"*
- XML Element start, named *SecondElement*, with an attribute *param2* equal to "something"
- XML Text node, with data equal to "Pre-Text"
- XML Element start, named *Inline*
- XML Text node, with data equal to "Inlined text"
- XML Element end, named *Inline*
- XML Text node, with data equal to " Post-text"
- XML Element end, named *SecondElement*
- Document End.

Limitations of SAX:

- With SAX, it is not possible to navigate through the document as you can with a DOM.
- The application must explicitly buffer those events it is interested in.

4. XML Schema

- XML Schema is an XML-based alternative to DTD.
- An XML schema describes the structure of an XML document.
- The XML Schema language is also referred to as XML Schema Definition (XSD).

What is an XML Schema?

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema:

- Defines elements that can appear in a document
- Defines attributes that can appear in a document
- Defines which elements are child elements
- Defines the order of child elements
- Defines the number of child elements
- Defines whether an element is empty or can include text
- Defines data types for elements and attributes
- Defines default and fixed values for elements and attributes

XML Schemas are the Successors of DTDs

1. XML Schemas are extensible to future additions
2. XML Schemas are richer and more powerful than DTDs
3. XML Schemas are written in XML
4. XML Schemas support data types
5. XML Schemas support namespaces

Note: XML Schema is a W3C Recommendation

XML Schema became a W3C Recommendation 02. May 2001.

The following example is an XML Schema file called **"note.xsd"** that defines the elements of the XML document above **("note.xml"):**

XML Schema example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element></xs:schema>
```

5. Presenting XML

- XSL stands for EXtensible Stylesheet Language, and is a style sheet language for/presenting XML documents.
- XS stylesheets are not the same as HTML Cascading Stylesheets.
- Create a style for a specific XML element or class of elements, with XSL a template is created. This template is used to format XML elements which match a specified pattern.
- The template is a page design or the design of part of a page.

XSL consists of many parts:

- XSLT stands for XSL Transformations.
- XSLT to transform XML documents into other formats, like XHTML.
- XPath (the XML Path language) is a language for finding information in an XML document.
- XSL-FO stands for Extensible Stylesheet Language Formatting Objects.
- XSL-FO is a language for formatting XML data for output to screen, paper or other media.
- XLink (the XML Linking language) defines methods for creating links within XML documents.
- XPointer (the XML Pointer language) allows hyperlinks to point to specific parts (fragments) of XML documents.

Example: note.xsl

XSL Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>

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

<xsl:template match="/">
```

```
  <html>
        <body>
        <h2>NOTE</h2>
                 <table border="1">
                 <tr bgcolor="#9acd32">
                 <th>note</th>
                 </tr>
                 <tr bgcolor="#9acd44">
                 <th>from</th>
                 <th>to</th>
                 <th>heading</th>
                <th>body</th>
                 </tr>
<xsl:for-each select="note">
                <tr>
                 <td><xsl:value-of select="from"/></td>
                 <td><xsl:value-of select="to"/></td>
                <td><xsl:value-of select="heading"/></td>
                 <td><xsl:value-of select="body"/></td>
                 </tr>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

UNIT-IV

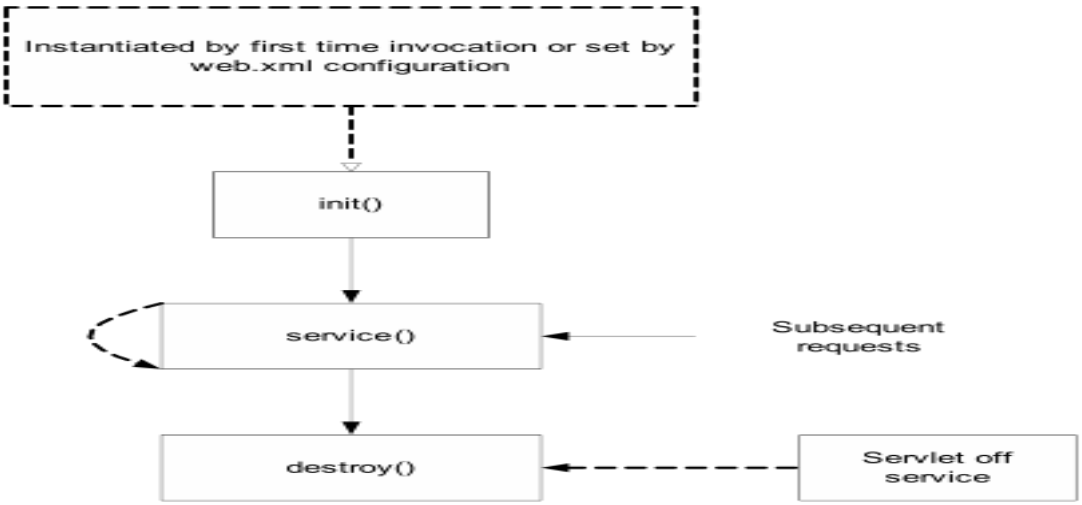
SERVLET PROGRAMMING

1. Explain about Servlet and its Life Cycle?

Ans.

Servlets: Servlets are small programs that execute on the server side of a web connection. Just as Applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.

There are three methods are central to the life cycle of a Servlet. These are init(),service() and destroy().They are implemented by every Servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.



Life cycle of a Servlet

i) Assume that a user enters a Uniform Resource Locator (URL) to a web browser .The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

ii) This HTTP request is received by the Web server. The server maps this request to a particular Servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

iii) The server invokes the init () method of the Servlet. This method is invoked only when the servlet is first loaded into the memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

iv) The server invokes the service() method of the Servlet. This method is called to process the HTTP request. You will see that it is possible for the Servlet to read data that has been provided in the HTTP reuest. It may also formulate an HTTP response for the client.

The servlet remains in the servers address space and is available to process any other HTTP requests received from the clients. The service () method is called for each HTTP request.

v) Finally, the server may decide to unload the Servlet from its memory. The destroy() method to relinquish any resources such as file handles that are allocated for the Servlet.

Important data may be saved to a persistent store .The memory allocated for the Servlet and its objects

can then be garbage collected.

2. Explain briefly about Servlet Implementation?

Ans:

Servlet Implementation

Servlet implementation can be done in 3 ways:

1. Servlet Interface
2. GenericServlet
3. HttpServlet

i) The Servlet Interface

syntax: public interface Servlet

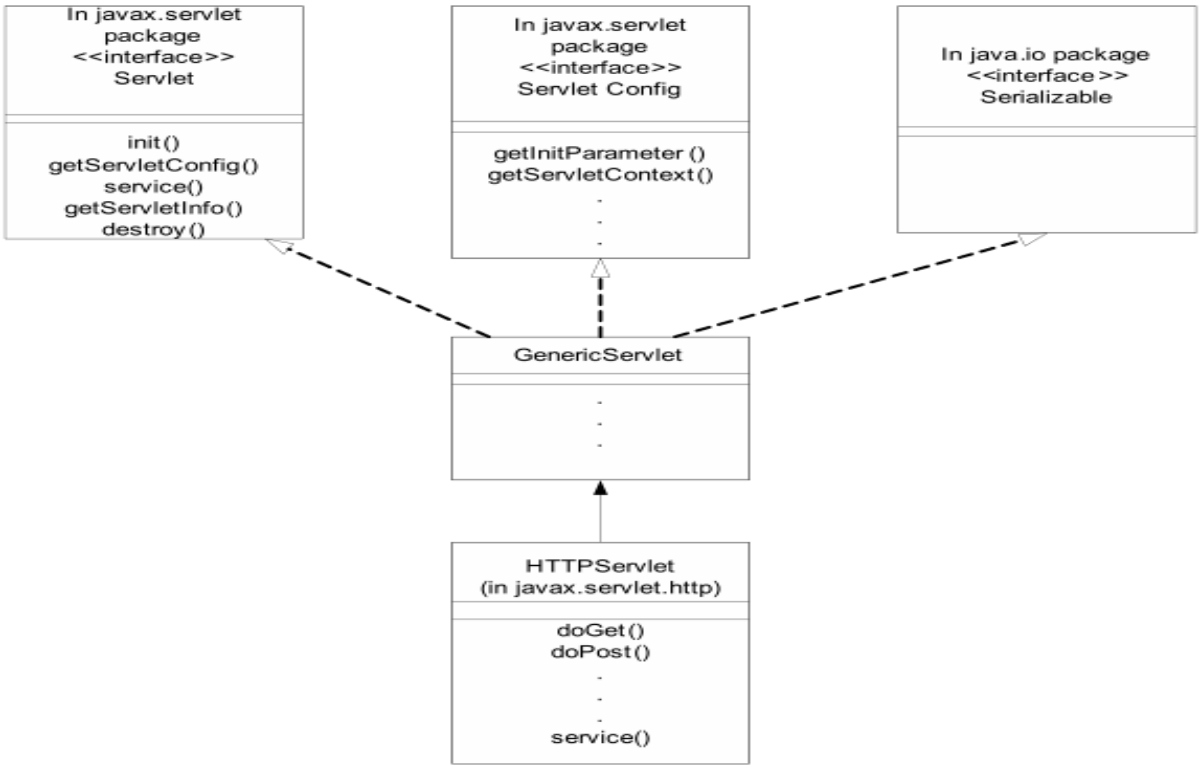
Interface Servlet and the Servlet Life Cycle

- Interface **Servlet**
  - All Servlets must implement this interface
  - All methods of interface **Servlet** are invoked by Servlet container
- Servlet life cycle
- A Servlet has a lifecycle just like a Java applet. The lifecycle is managed by the Servlet container.
- There are three methods in the Servlet interface, which each Servlet class must implement. They are init(), service(), and destroy().
  - Servlet container invokes the Servlets **init** method
  - Servlets **service** method handles requests
  - Servlets **destroy** method releases servlet resources when the Servlet container terminates the Servlet
  -

- Servlet implementation
  - **GenericServlet**
  - **HttpServlet**

Java Servlet Architecture

Java Servlet Class Hierarchy



- A Java Servlet is a typical Java class that extends the abstract class HttpServlet.
- The HttpServlet class extends another abstract class called GenericServlet.
- The GenericServlet class implements three interfaces:

✓ javax.servlet.Servlet

✓ javax.servlet.ServletConfig and

✓ java.io.Serializable.

Javax.servlet.Servlet interface, the following five methods must be implemented:

Method	Description
void init(ServletConfig config) throws ServletException	The servlet container calls this method once during a servlet's execution cycle to initialize the servlet. The ServletConfig argument is supplied by the servlet container that executes the servlet.
ServletConfig getServletConfig()	This method returns a reference to an object that implements interface ServletConfig. This object provides access to the servlet's configuration information such as servlet initialization parameters and the servlet's ServletContext, which provides the servlet with access to its environment (i.e., the servlet container in which the servlet executes).
String getServletInfo()	This method is defined by a servlet programmer to return a string containing servlet information such as the servlet's author and version.
void service(ServletRequest request, ServletResponse response) throws ServletException, IOException	The servlet container calls this method to respond to a client request to the servlet.
void destroy()	This "cleanup" method is called when a servlet is terminated by its servlet container. Resources used by the servlet, such as an open file or an open database connection, should be deallocated here.

ii) The GenericServlet Class

public abstract class GenericServlet implements Servlet, ServletConfig, Serializable

The methods are:

- public init()
- public void log(String message)
- public void log(String message, Throwable t)

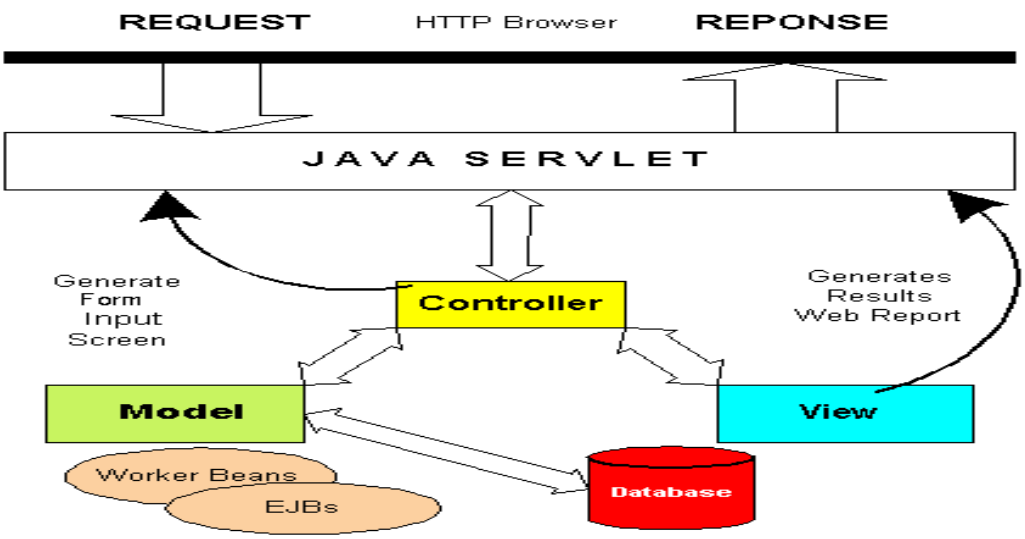
The SingleThreadModel Interface

public interface SingleThreadModel

javax.servlet.SingleThreadModel

During the lifetime of a servlet that doest not implement this interface, the container may send multiple service requests in different threads to a single instance. This means that implementation of the service() method should be thread-safe. However, whats the alternative if the service() method of a servlet is not thread-safe?

iii) The HttpServlet Class



public abstract class HttpServlet extends GenericServlet implements Serializable

The methods are:

- public void service(ServletRequest request, ServletResponse response) throws ServletException,IOException
- protected void service(HttpServletRequest request,
  HttpServletResponse response)
- protected void doGet(HttpServletRequest request,
  HttpServletResponse response)
- protected void doPost(HttpServletRequest request,
  HttpServletResponse response)
- protected void doHead(HttpServletRequest request,
  HttpServletResponse response)
- protected void doDelete(HttpServletRequest request,
  HttpServletResponse response)
- protected void doOptions(HttpServletRequest request,
  HttpServletResponse response)
- protected void doPut(HttpServletRequest request,
  HttpServletResponse response)
- protected void doTrace(HttpServletRequest request,
  HttpServletResponse response)
- protected long getLastModified(HttpServletRequest request)

3. Explain about Servlet Configuration and Servlet Exceptions?

Ans.

Servlet Configuration:

- In the Java Servlet API, javax.servlet.ServletConfig objects represent the configuration of a servlet.
- The configuration information contains initialization parameters (a set of name/value pairs), the name of the servlet and a javax.servlet.ServletContext object, which gives the servlet information about the container.
- The initialization parameters and the name of a servlet can be specified in the deployment descriptor (the web.xml file).

  Example:

```
<web-app>
<servlet>
   <servlet-name>Admin</servlet-name>
   <servlet-class>com.apress.admin.AdminServlet
         </servlet-class>
      <init-param>
         <param-name>email</param-name>
         <param-value>admin@admin.apress.com</param-value>
      </init-param>
      <init-param>
         <param-name>helpURL</param-name>
         <param-value>/admin/help/index.html</param-value>
      </init-param>
</servlet>
</web-app>
```

  For Example registers a **servlet** with name **Admin**, and specifies two initialization **parameters**, **email** and **helpURL**.

- The web container reads this information, and makes it available to the com.apress.admin.AdminServlet via the associated javax.servlet.ServletConfig object.
- If we want to change these parameters, we can do so **without** having to **recompile** the servlet.

  The ServletConfig Interface

  public interface ServletConfig

  Methods:

  - public String getInitParameter(String name)

    Ex: admin@admin.apress.com

- public Enumeration getInitParameterNames()
  Ex: email and helpURL
- public ServletContex getServletContext()
- public String getServletName()

➢ Servlet Exception

javax.servlet package specifies two exception classes:

- javax.servlet.ServletException
- javax.servlet.UnavailableException

public class **ServletExceptio**n extends java.lang.Exception

This is a generic exception, which can be thrown by the init(), service(), doXXX(), and destroy() methods. The class provides the following constructions:

- public ServletException()
- public ServletException(String message)

While creating objects of type ServletException, you can embed any application level exception (called the root-cause).

The containers use the root-cause exception for logging purposes.

For instance, you can embed a java.sql.SQLException in a javax.servlet.ServletException.

There are two additional constructors to support root-cause exceptions:

- public ServletException(Throwable cause)
- public ServletException(String message, Throwable cause)

The getRootCause() returns the root-cause exception:

- public Throwable getRootCause()
- public class **UnavailableExceptio**n extends ServletException

The purpose of this is to indicate to the webcontainer that the servlet is either temporarily or permanently unavailable, then the following constructions:

- public UnavailableException(String message)
- public UnavailableException(String message, int seconds)

4. Explain about cookies and session tracking with an example? (or) Define Session and Describe with an example how to use the HttpSession object to find out the creation time and the last-accessed time for a session using HttpSession interface.

Ans.

Cookies: Cookies are the most commonly used means of tracking client sessions. Cookies were initially introduced by Netscape, and this technology was later standardized in RFC 2109.

Definition: A Cookie is a small piece of textual information sent by the server to the client, stored on the client, and return by the client for all requests to the server.

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

Drawback: This may not be an effective way because many time browsers do not support a cookie, so I would not recommend using this procedure to maintain the sessions.

Session: Session tracking enables you to track a user's progress over multiple servlets or HTML pages, which, by nature, are stateless. A *session* is defined as a series of related browser requests that come from the same client during a certain time period. Session tracking ties together a series of browser requests—think of these requests as pages—that may have some meaning as a whole, such as a shopping cart application.

javax.servlet.HttpSession interface

The HttpSession Object:

Apart from the above mentioned three ways, servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.

You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below:

HttpSession session = request.getSession();

You need to call *request.getSession()* before you send any document content to the client. Here is a summary of the important methods available through HttpSession object:

S.No.	Method & Description
1	public Object getAttribute(String name) This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	public Enumeration getAttributeNames() This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	public long getCreationTime() This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	public String getId() This method returns a string containing the unique identifier assigned to this session.

Session Tracking Example:

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
// Extend HttpServlet class
        public class SessionTrack extends HttpServlet {
            public void doGet(HttpServletRequest request,
                              HttpServletResponse response)
                                  throws ServletException, IOException
        {
            // Create a session object if it is already not created.
            HttpSession session = request.getSession(true);
                // Get session creation time.
            Date createTime = new Date(session.getCreationTime());
                // Get last access time of this web page.
            Date lastAccessTime = new Date(session.getLastAccessedTime());
                String title = "Welcome Back to my website";
                Integer visitCount = new Integer(0);
                String visitCountKey = new String("visitCount");
                String userIDKey = new String("userID");
                String userID = new String("ABCD");
                // Check if this is new comer on your web page.
            if (session.isNew()){
                title = "Welcome to my website";
                session.setAttribute(userIDKey, userID);
            } else {
        visitCount = (Integer)session.getAttribute(visitCountKey);
```

```
                visitCount = visitCount + 1;
                userID = (String)session.getAttribute(userIDKey);
                                }
                session.setAttribute(visitCountKey, visitCount);
                                // Set response content type
                        response.setContentType("text/html");
                        PrintWriter out = response.getWriter();
        out.println("<html>\n" + "<head><title>" + title + "</title></head>\n" +
 "<body bgcolor=\"#f0f0f0\">\n" + "<h1 align=\"center\">" + title + "</h1>\n" + "<h2
align=\"center\">Session Infomation</h2>\n" +"<table border=\"1\" align=\"center\">\n"
+ "<tr bgcolor=\"#949494\">\n" + " <th>Session info</th><th>value</th></tr>\n" +
"<tr>\n" + " <td>id</td>\n" + " <td>" + session.getId() + "</td></tr>\n" + "<tr>\n" +
" <td>Creation Time</td>\n" + " <td>" + createTime + " </td></tr>\n" +
"<tr>\n" + " <td>Time of Last Access</td>\n" +
" <td>" + lastAccessTime + " </td></tr>\n" + "<tr>\n" +
" <td>User ID</td>\n" + " <td>" + userID + " </td></tr>\n" + "<tr>\n" +
" <td>Number of visits</td>\n" + " <td>" + visitCount + "</td></tr>\n" + "</table>\n"
+ "</body></html>");
                } }
```

Compile above servlet **SessionTrack** and create appropriate entry in web.xml file. Now running *http://localhost:8080/SessionTrack* would display the following result when you would run for the first time:

Output:

Welcome to my website

Session Information **Session info**	**Value**
Id	0AE3EC93FF44E3C525B435 1B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	**0**

Now try to run the same servlet for second time, it would display following result.

Output:

Welcome Back to my website

Session Information **info type**	**Value**
Id	0AE3EC93FF44E3C525B435 1B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	**1**

Note: Have a glance on the topics:

ServletRequest, ServletResponse, HttpServletRequest, HttpServletResponse interfaces.

UNIT-V

DATABASE PROGRAMMING WITH JDBC

1. Discuss the process of mapping SQL types to Java?

Ans:

Standard SQL-Java data type mappings

The table opposite shows standard mappings between SQL data types and Java primitive and object types. In practice, many databases provide these standard types plus some subtly non-standard variants. You might find that on your column you are using the subtly non-standard variant!

If you are not sure of the type, you can always retrieve getObject() and then find out what type it is (via getClass()). Luckily, JDBC will generally come to the rescue by *converting* between the type returned by the JDBC driver and the type requested.

Correspondence between SQL and Java types

SQL data type	**Java data type**
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
FLOAT, DOUBLE	Double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
VARCHAR	String
BLOB	java.sql.Blob

Data type conversion

The ResultSet.getXXX() methods are generally slightly "lenient". If you ask for a different type to that returned by the JDBC driver (e.g. if you ask for an int when the actual data type was a short), then JDBC will automatically convert the value to the requested type. This conversion works:

- between all **numeric types** (though you could lose precision in the conversion);
- between most types and String.

2. Explain the process of connection management using JDBC.

Ans:

Connection Management:

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.

23

Methods to obtain connections

The Driver Manager 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.

3. Discuss about Prepared Statements?

Ans:

In database management systems, a **prepared statement** or **parameterized statement** is a feature used to execute the same or similar database statements repeatedly with high efficiency. Typically used with SQL statements such as queries or updates, the prepared statement takes the form of a template into which certain constant values are substituted during each execution.

The typical workflow of using a prepared statement is as follows:

1. **Prepare**: The statement template is created by the application and sent to the database management system (DBMS). Certain values are left unspecified, called*parameters*, *placeholders* or *bind variables* (labelled "?" below):
   - INSERT INTO PRODUCT (name, price) VALUES (?, ?)
2. The DBMS parses, compiles, and performs query optimization on the statement template, and stores the result without executing it.
3. **Execute**: At a later time, the application supplies (or *binds*) values for the parameters, and the DBMS executes the statement (possibly returning a result). The application may execute the statement as many times as it wants with different values. In this example, it might supply 'Bread' for the first parameter and '1.00' for the second parameter.

As compared to executing SQL statements directly, prepared statements offer two main advantages:[1]

- The overhead of compiling and optimizing the statement is incurred only once, although the statement is executed multiple times. Not all optimization can be performed at the time the prepared statement is compiled, for two reasons: the best plan may depend on

the specific values of the parameters, and the best plan may change as tables and indexes change over time.[2]

- Prepared statements are resilient against SQL injection, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

On the other hand, if a query is executed only once, server-side prepared statements can be slower because of the additional round-trip to the server.[3] Implementation limitations may also lead to performance penalties: some versions of MySQL did not cache results of prepared queries,[4] and some DBMSs such as PostgreSQL do not perform additional query optimization during execution.[5][6]

A stored procedure, which is also precompiled and stored on the server for later execution, has similar advantages. Unlike a stored procedure, a prepared statement is not normally written in a procedural language and cannot use or modify variables or use control flow structures, relying instead on the declarative database query language. Due to their simplicity and client-side emulation, prepared statements are more portable across vendors.

This example uses Java and the JDBC API:

```
java.sql.PreparedStatement stmt = connection.prepareStatement(
            "SELECT * FROM users WHERE USERNAME = ? AND ROOM = ?");
stmt.setString(1, username);
stmt.setInt(2, roomNumber);
stmt.executeQuery();
```

Java PreparedStatement provides "setters" (setInt(int), setString(String), setDouble(double), etc.) for all major built-in data types.

4. Explain about Database Drivers

Ans:

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 implement 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.

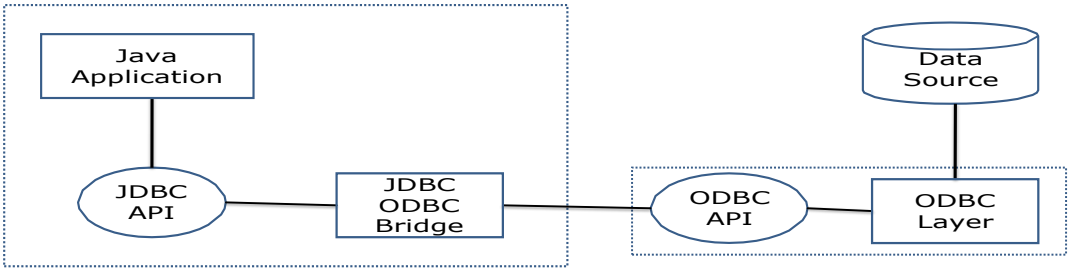


Figure Type 1: JDBC-ODBC Bridge Driver

11

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) diver is an example of a Type 2 driver.

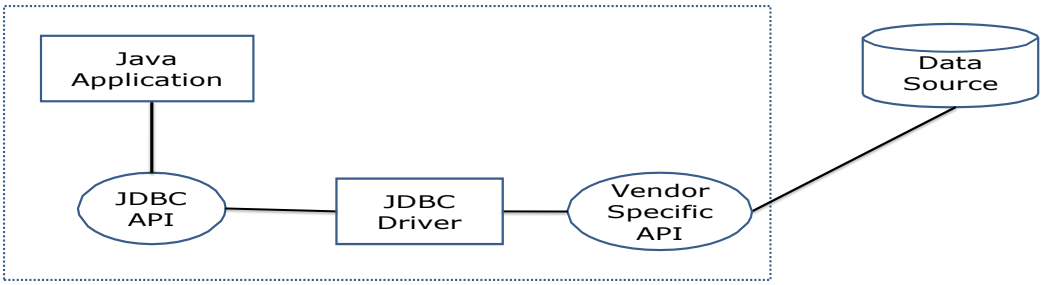


Figure Type 2: JDBC-NativeAPI

14

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.

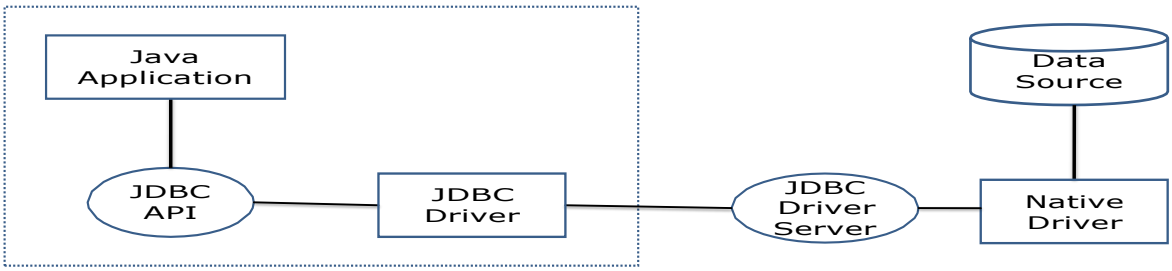
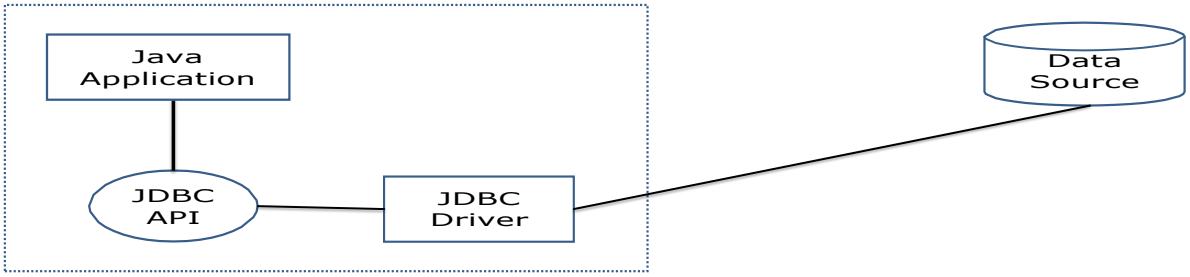


Figure Type 3: JDBC-PartJava

17

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.



FigureType 4: 100% pure Java

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.

5. Explain about Save Points in JDBC

Ans:

SAVEPOINT in JDBC

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off auto-commit and manage your own transactions:

- To increase performance
- To maintain the integrity of business processes
- To use distributed transactions

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to **turn** it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit:

```
conn.setAutoCommit(false);
```

Commit & Rollback:

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows:

```
conn.commit();
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback();
```

The following example illustrates the use of a commit and rollback object:

```
try{
//Assume a valid connection object conn
conn.setAutoCommit(false);
Statementstmt=conn.createStatement();

String SQL ="INSERT INTO Employees "+
"VALUES (106, 20, 'Rita', 'Tez')";
stmt.executeUpdate(SQL);
//Submit a malformed SQL statement that breaks
String SQL ="INSERTED IN Employees "+
"VALUES (107, 22, 'Sita', 'Singh')";
stmt.executeUpdate(SQL);
// If there is no error.
conn.commit();
}catch(SQLException se){
// If there is any error.
conn.rollback();
}
```

In this case none of the above INSERT statement would success and everything would be rolled back.

Using Savepoints:

The new JDBC 3.0 Savepoint interface gives you additional transactional control. Most modern DBMS support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints:

- **setSavepoint(String savepointName):** defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(SavepointsavepointName):** deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (StringsavepointName)** method which rolls back work to the specified savepoint.

The following example illustrates the use of a Save point object:

```
try{
//Assume a valid connection object conn
conn.setAutoCommit(false);
Statementstmt=conn.createStatement();

//set a Savepoint
Savepoint savepoint1 =conn.setSavepoint("Savepoint1");
String SQL ="INSERT INTO Employees "+
"VALUES (106, 20, 'Rita', 'Tez')";
stmt.executeUpdate(SQL);
//Submit a malformed SQL statement that breaks
String SQL ="INSERTED IN Employees "+
"VALUES (107, 22, 'Sita', 'Tez')";
stmt.executeUpdate(SQL);
// If there is no error, commit the changes.
conn.commit();

}catch(SQLException se){
// If there is any error.
conn.rollback(savepoint1);
}
```

In this case none of the above INSERT statement would success and everything would be rolled back.

6. Write a java program for storing student details like name, roll number, branch and mobile number into the database using JDBC.

Ans:

```
// STEP 1. Import required packages
import java.sql.*;
```

```
public class FirstExample
{
   public static void main(String[] args)
  {
    Connection conn = null; //Connection is an interface, create only reference: conn
   Statement stmt = null; //Statement is an interface, create only reference: stmt
   try{
       //STEP 2: Register JDBC driver
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  //Class.forName  calls  indirectly
         DriverManager.registerDriver() and Type-1 driver

      //STEP 3: Open a connection

         conn = DriverManager.getConnection("jdbc:odbc:Data4"); //DSN: Data
         System.out.println("Connected to database...");
          //STEP 4: Execute a query
         System.out.println("Creating statement...");
         stmt = conn.createStatement();

    String sql1;
   sql1="create table emp_data4(empId Number,empName Text, age Number, city Text)";
   stmt.executeUpdate(sql1);
   System.out.println("table created");
   String sql2, sql3;
   sql2="insert into emp_data4 values(9,'xyz',29,'delhi')";
   stmt.executeUpdate(sql2);
   sql3="insert into emp_data4 values(10,'abc',50,'Mumbai')";
   stmt.executeUpdate(sql3);
   System.out.println("2 rows inserted");

   String sql4;
   sql4 = "select empId, empName, age,city from emp_data4";
   ResultSet rs = stmt.executeQuery(sql4);

   //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
```

Write short notes on:

7. 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.

8. 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.

9. 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.

10. Exceptions and Warnings

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

11. 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.

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.

12. Establishing 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()

13. Creating and Executing SQL Statements

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()

Note: Have a glance on the topic: Transaction Support

UNIT -6

INTRODUCTION TO JSP

1. What are the limitations of Servlets? Discuss how JSP overcomes these Problems.

- Servlets of web application requires strong knowledge of Java.
- Servlets are very difficult for Non-Java Programmers to understand and carry out.
- We know that, Servlet is a picture of both Presentation logic(HTML) and Business Logic(Java) . At the time of development of Servlet by using both of the above (Presentation and Business Logic), it may become imbalance, because a Servlet developer can't be good in both Presentation logic and Business logic.
- Servlet never provides separation between or clarity between Presentation Logic and Business Logic.
- So that servlets do not give Parallel Development.
- If we do any changes in a servlet, then we need to do Re-deployment process i.e. Servlets modifications requires redeployment, which is one of the time-consuming process.
- If we develop any web application with servlets, then it is mandatory for the web application developer to configure web-application configuration file.
- Servlets do not offer any implicit object. implicit objects generally provided by containers during the dynamic program execution .
- Servlets do not contain a facility called Custom Tags Development.
- Servlets don't provide Global/Implicit exception handling facility.
- Servlets don't vontain Automatic Page compilation concept.

Advantages of using JSP:-

- JSP pages easily combine static templates, including HTML or XML fragments, with code that generates dynamic content.
- JSP pages are compiled dynamically into servlets when requested, so page authors can easily make updates to presentation code. JSP pages can also be precompiled if desired.
- JSP tags for invoking JavaBeans components manage these components completely, shielding the page author from the complexity of application logic.
- Developers can offer customized JSP tag libraries that page authors access using an XML-like syntax.
- Web authors can change and edit the fixed template portions of pages without affecting the application logic. Similarly, developers can make logic changes at the component level without editing the individual pages that use the logic.

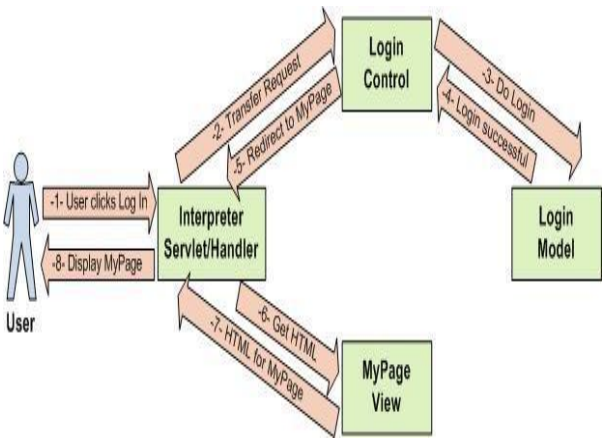
2. What is MVC architecture? Explain with a neat diagram.

The architecture of the portal UI is based on the **Model-View-Control (MVC)** design pattern. The MVC paradigm allows you to separate the code that handles business logic from the code that controls presentation and event handling. Each page in the portal is made up of a combination of at least one Model and View, and one or more Controls.

- **Model** classes store the data for a page or page section. A single page might use one or more Model classes, depending on how much of the page data can be shared by other types of pages. A Model defines how data is accessed and set for a given page, including any functions necessary for security or data validation and modification. Models encapsulate calls to the portal server API and also store UI-specific data. Data that is globally accessed by the UI is available from the ActivitySpaceobject . All other data should be stored in a Model.
- **View** classes contain HTMLElements and HTMLConstructs that describe how the data from the Model should be displayed to the user. In the portal UI design, **DisplayPage** objects are used to aggregate View objects to encapsulate all the information needed to render a particular page. Some Views are common throughout the portal and some are specific to certain pages. For example, the banner that makes up the majority of the portal is a common View that defines the color scheme and where the search section will be displayed. In contrast, the View used to create and modify data within a User Profile is specific to the User Profile function and is seen only on that page.
- **Control** are actions or sets of actions that are executed when a specific event is triggered. Multiple Controls can be defined within a page, each with its own functional specification. For example, one Control might produce a popup window that allows the user to browse for a specific object and places the selection within the View, and another could save the new data to the Model.

The diagram below shows a simplified version of how the classes used for the Login page interact with each other using the Model-View-Control (MVC) pattern.

1. When the user clicks Log In on the login page, a request is sent to the portal. The **Interpreter** Servlet / Handler handles all requests.
2. The Interpreter dispatches the request to the appropriate Control class; in this case, the **Login Control**.
3. The Control performs the action requested by the user on the Model; in this case the **Login Model**.
4. The Model returns a response; in this example, a successful login message.



5. The Control either redirects to another page or returns to the same page with new data since the Model has been updated. In this case, the Control redirects to the MyPage.
6. The Interpreter gets the HTML for the page requested by the Control from the appropriate View, in this case the **MyPage View**.
7. The View returns the HTML for the page.
8. The Interpreter sends the HTML for the page back to the browser.

3. What are the advantages and disadvantages of JSP over Servlets?

ADVANTAGES ARE GIVEN ABOVE:

Disadvantages of using JSP:

1. JSPs are converted into servlets by the JSP container. This makes the page load a bit late (but only for the first time) from then on it loads the servlet from the cache.

2. Java code in JSP: This actually proves to be a nightmare for HTML programmers as java code in JSP has got "lousy looping". The main idea of JSP was to clearly separate the presentation from the business logic or the processing logic. But it doesn't seem to operate that way. The temptation to add java code in JSP is always there. Sometimes the HTML programmers may even "knock off" some java code in the JSP file leading to "chaos" Solution: We can use the combination of custom TagLibs and Bean Tags to overcome this problem, but only to a certain extent. Moreover if we have our architecture right, wherein we use a combination of servlets and JSP we can overcome the above problem. In this architecture only the "View" is given to the JSP. The "Controller" will be a servlet that takes care of the processing logic.
3. Using Templates in JSP: Though in its initial stage, by using templates we can componentize the GUI also. The design pattern says "Anything that changes " should be encapsulated.

4. Discuss about Standard Action and implicit objects?

Standard Actions in JSP:-

- 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.

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.

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 />

Write short notes on:

5. JSP Directives

The three types of directives are:

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" >

Examples

Import java packages

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

Multiple import statements

<%@ page import="java.util.*" %>

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

Including file at *translation time*

<%@ include file="header.html" %>

6. Scripting Elements

- The Scriptlet
- JSP Declarations
- JSP Expression
- JSP Comments
- A Test of Comments

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 %>

Example

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

```
if (name == null)

{ %>

        <h3>Please supply a name</h3>

<% }

else

{ %>

        <h3>Hello <%= name %></h3>

<% } %>
```

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;]+ ... %>

Example

```
%! 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.
- Following is the syntax of JSP Expression:
- **<%= expression %>**
- **Example:**

```
Your name is <%= request.getParameter("name") %>
and your age is <%= request.getParameter("age") %>
The value of pi is <%= Math.PI %> and
The square root of two is <%= Math.sqrt(2.0) %> and
Today's date is <%= new java.util.Date() %>
```

- 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>
```

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 --%>

E**xample**

```
<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>
```

Table 6. Test of Comments

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.

7. 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>
```

8. Write a JSP page to print "hello".

Ans. hello.jsp

```
<html>

    <head>

    <title>Sample page</title></head>
```

```
        <body>
            <%=new String("Hello!") %>
        </body>
    </html>
```

The hello.jsp page is a static HTML page embedded with a JSP command. A JSP command is an XML-like snippet that encapsulates logic that dynamically generates content within the static HTML. JSP commands can include directives, declarations, expressions, actions, and blocks of Java code, all enclosed within angle-brackets, like XML elements.

At compile-time, the JSP is converted into a servlet, which is what tc Runtime instance actually executes at runtime.

The hello.jsp includes the following simple JSP directive:

```
<%=new String("Hello!") %>
```

This JSP directive simply prints out a message to the client (browser): Hello!

NOTE:

Have a glance on the topic:

Processing XML in JSP

UNIT -7

JSP TAG EXTENSIONS

1. Introduction to Javabean

A. **JavaBean** is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes:

- It provides a default, no-argument constructor.
- It should be serializable and implement the Serializable interface.
- It may have a number of properties which can be read or written.
- It may have a number of "getter" and "setter" methods for the properties.

JavaBeans Properties

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

Method	Description
getPropertyName()	For example, if property name is *firstName*, your method name would be getFirstName() to read that property. This method is called accessor.
setPropertyName()	For example, if property name is *firstName*, your method name would be setFirstName() to write that property. This method is called mutator.

A read-only attribute will have only a get**PropertyName**() method, and a write-only attribute will have only a set**PropertyName**() method.

2. Accessing JavaBeans

- The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP.

The full syntax for the useBean tag is as follows:

<jsp:useBean id="bean's name" scope="bean's scope" typeSpec/>

Here values for the scope attribute could be page, request, session or application based on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other useBean declarations in the same JSP.

Following **example** shows its simple usage:

```
<html>
    <head>
        <title>useBean Example</title>
    </head>
    <body>
        <jsp:useBean id="date" class="java.util.Date" />
        <p>The date/time is <%= date %>
    </body>
</html>
```

2.1 Accessing JavaBeans Properties

Along with <jsp:useBean...>, you can use <jsp:getProperty/> action to access get methods and <jsp:setProperty/> action to access set methods. Here is the full syntax:

```
<jsp:useBean id="id" class="bean's class" scope="bean's scope">
<jsp:setProperty name="bean's id" property="property name"
    value="value"/>
<jsp:getProperty name="bean's id" property="property name"/>
  ...........
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the get or set methods that should be invoked.

JavaBeans Example

Consider a student class with few properties:

```
public class StudentsBean implements java.io.Serializable
{
  private String firstName = null;    private String lastName = null;
  private int age = 0;
  public StudentsBean() {}
  public String getFirstName()
      {
            return firstName;
       }
  public String getLastName()
      {
          return lastName;
        }
  public int getAge()
      {
          return age;
      }
  public void setFirstName(String firstName)
      {
          this.firstName = firstName;
       }
  public void setLastName(String lastName)
      {
            this.lastName = lastName;
       }
 public void setAge(Integer age){
  this.age = age;
  }
```

```
}
```

Following is a simple example to access the data using above syntax:

```
<html>
        <head>
                <title>get and set properties Example</title>
        </head>
        <body>
         <jsp:useBean id="students‟  class="/StudentsBean">
         <jsp:setProperty name="students" property="firstName” value="Zara"/>
         <jsp:setProperty name="students" property="lastName‟ value="Ali"/>
         <jsp:setProperty name="students" property="age” value="10"/>
         </jsp:useBean>
         <p>Student First Name:
         <jsp:getProperty name="students" property="firstName"/>
        </p>
        <p>Student Last Name:
        <jsp:getProperty name="students" property="lastName"/>
        </p>
        <p>Student Age:
        <jsp:getProperty name="students" property="age"/>
        </p>
        </body>
</html>
```

3. Jsp Tag Extensions

- JSP 1.1 specification was support for Tag Extensions (custom tags).
- JSP 1.2 Specification, confirming tag extensions as one of the important features of JSP.
- The extension look a lot like standard HTML or XML tags embedded in a JSP page, but they have a special

- Meaning to the JSP engine at translation time, and allow custom functionality to be invoked without having to write Java code within scriptlets.

Attribute	Description
Uri	Specifies the relative or absolute uri of the tag library descriptor
tagPrefix	Specifies the required prefix that distinguishes custom tags from built-in tags. The prefix names **jsp, jspx, java, javax, servelt, sun** and **sunw** are reserved.

Tag Library

- Is a collection of related tags
  - Tag(s) can be packaged in a Tag Library
- Typically packaged as a jar file containing
  - A tag library descriptor (TLD)

    e.g. META-INF/taglib.tld
  - *.class files for the tag handler class(es)
  - Any additional associated resource(s)

Syntax

<%@ taglib prefix="myprefix" uri=”myuri” %>

– prefix: identifies the tag library

– uri: uniquely identifies the tag library descriptor

(TLD) directly or indirectly

Using Custom Tags

Tags are made available within a JSP page *via* the **taglib** directive:

<%@ taglib uri=”*uri*” prefix=”*prefix*” %>

- Directive’s **uri** attribute references the TLD (established via WAR file’s **web.xml**)
- Directive’s **prefix** attribute provides a local namespace for the TLD’s tags

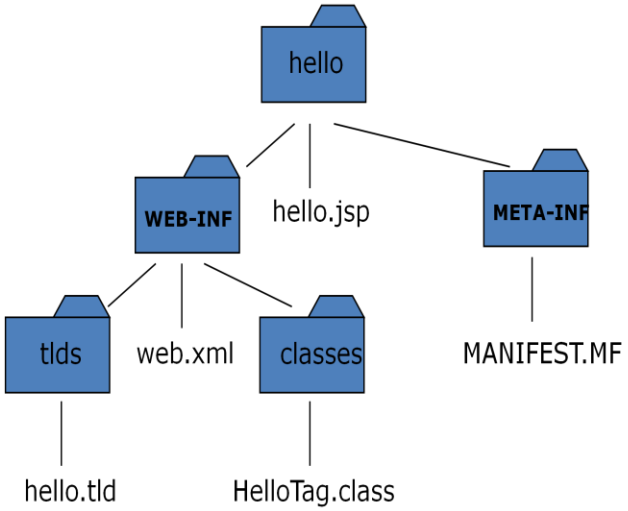


Figure 1. Structure of the war file

A war file is a jar file with special directories and a file named web.xml in the WEB-INF directory

A Simple Tag

Hello.jsp

```
<%@ taglib prefix="ex" uri="/hello"%>
<html>
    <head><title>A sample custom tag</title>
     </head>
<body>
    <ex:Hello>
    //</ex:Hello>
</body>
</html>
```

HelloTag.java

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*; import
java.io.*;
        public class HelloTag extends SimpleTagSupport {
          public void doTag() throws JspException, IOException {
         JspWriter out = getJspContext().getOut();
         out.println("Hello Custom Tag!");
  } }
```

Hello.tld (Tag Library Descriptor>

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>ex</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>HelloTag</tag-class>
    <body-content>empty</body-content></tag> </taglib>
```

web.xml

```
<web-app>
  <taglib>
        <taglib-uri>/hello</taglib-uri>
        <taglib-location>/WEB-INF/tlds/hello.tld</taglib-location>
  </taglib>
   </web-app>
```

4. Anatomy of tag Extensions

- A **Tag handler** is a JavaBean that implements one of three interfaces defined in the javax.servlet.jsp.tagext package:
  - Tag
  - IterationTag
  - BodyTag
- A **Tag Library Descriptor** or TLD which is an XML document that contain information about one or more tag extensions.

Note:

Like the standard actions, custom tags follow XML syntax conventions:

```
<prefix:name attribute="value" attribute="value"/>
        (or)
<prefix:name attribute="value" attribute="value">
    body content

</prefix:name>
```

Tag handlers

- A tag handler class must implement one of the following interfaces:
  - javax.servlet.jsp.tagext.Tag
  - javax.servlet.jsp.tagext.IterationTag
  - javax.servlet.jsp.tagext.BodyTag
- Usually extends utility class
  - javax.servlet.jsp.tagext.TagSupport or
  - javax.servlet.jsp.tagext.BodyTagSupport class
- Located in the same directory as servlet class files
  - /WEB-INF/classes/<package-directory-structure>
- A tag handler can optionally implement javax.servlet.jsp.tagext.TryCatchFinally
- Tag attributes are managed as JavaBeans properties (i.e., via getters and setters)

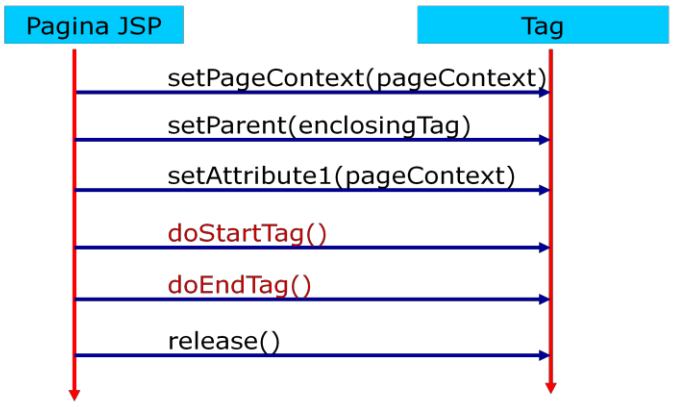


Figure 2. Javax.servlet.jsp.tagext**.Tag interface**

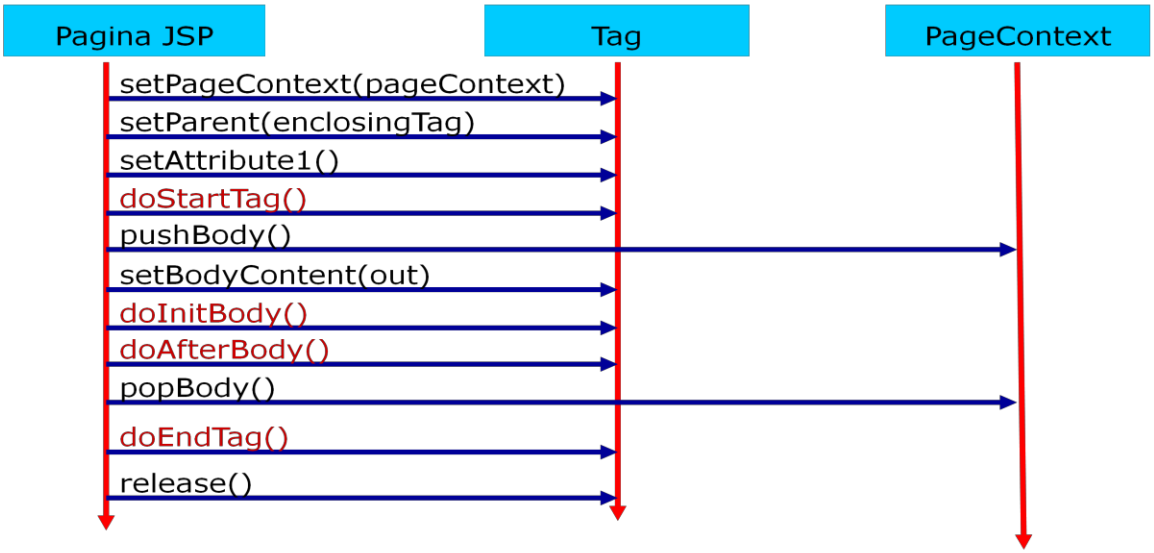


Figure 3. Javax.servlet.jsp.tagext.**BodyTag interface**

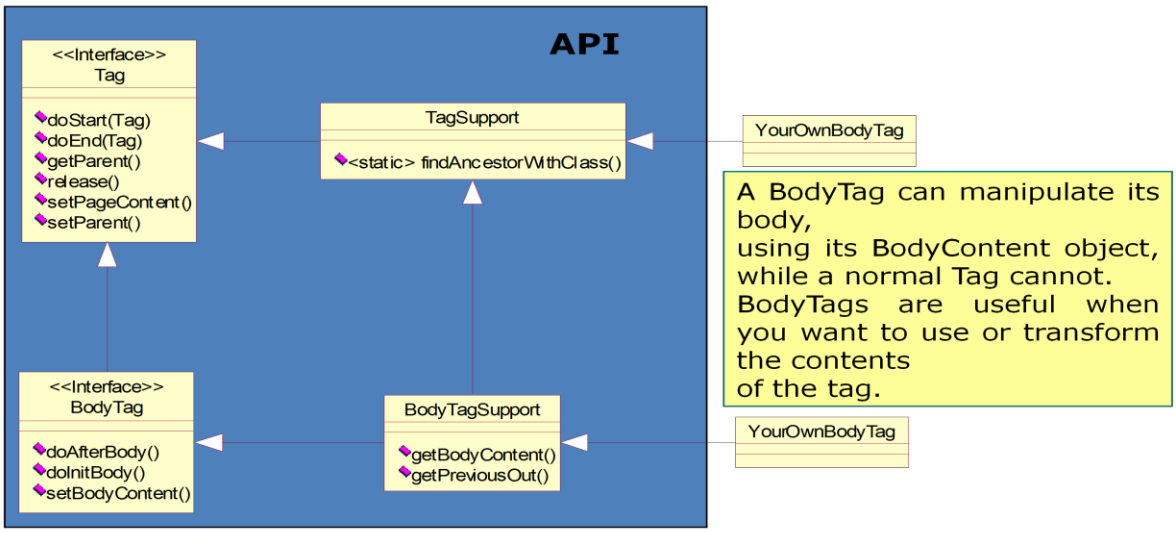


Figure 4. Class Diagram

5. Writing Tag Extensions

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 10.30: customTagWelcome.jsp              -->
6  <!-- JSP that uses a custom tag to output content. -->
7
8  <%-- taglib directive --%>
9  <%@ taglib uri = "example-taglib.tld" prefix = "example" %>
10
11  <html xmlns = "http://www.w3.org/1999/xhtml">
12
13    <head>
14      <title>Simple Custom Tag Example</title>
15    </head>
16
17    <body>
18      <p>The following text demonstrates a custom tag:</p>
19      <h1>
20        <example:welcome />
21      </h1>
22    </body>
23
24  </html>
```

Use `taglib` directive to include use tags in tag library

Use custom tag `welcome` to insert text in the JSP

Figure 5. JSP customTag-Welcome.jsp uses a simple custom tag

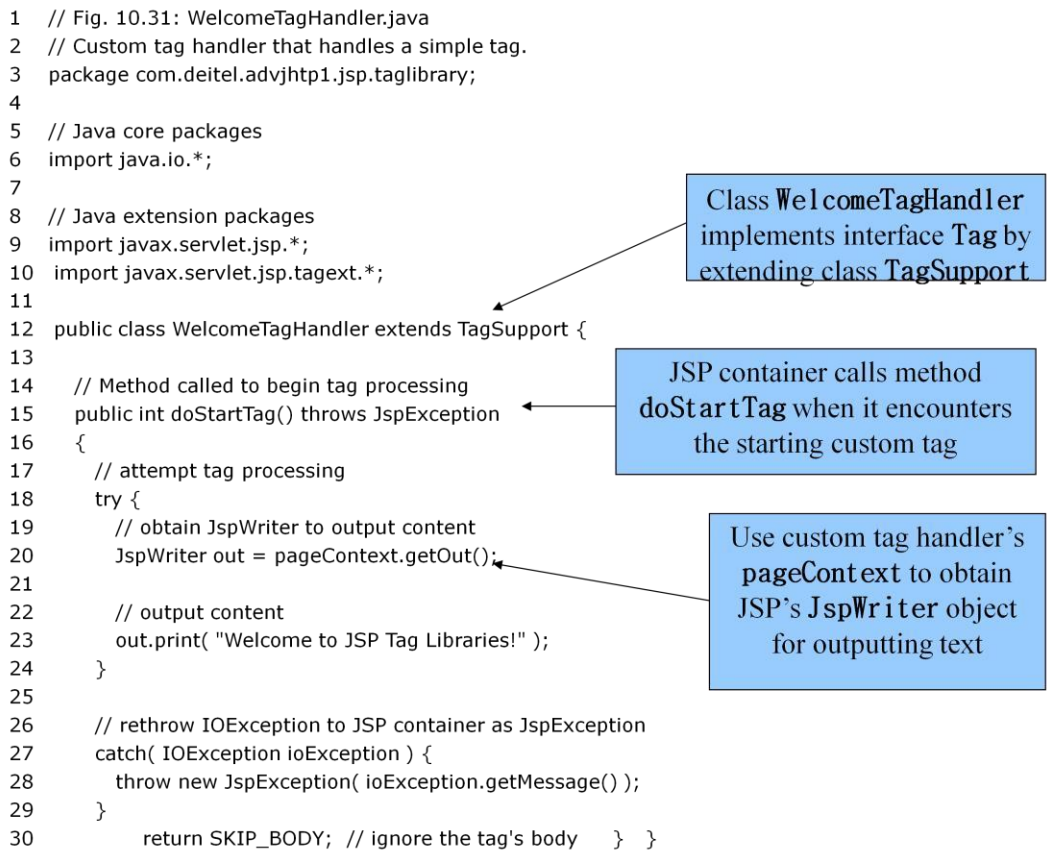


Figure 6. WelcomeTag-Handler custom tag handler.

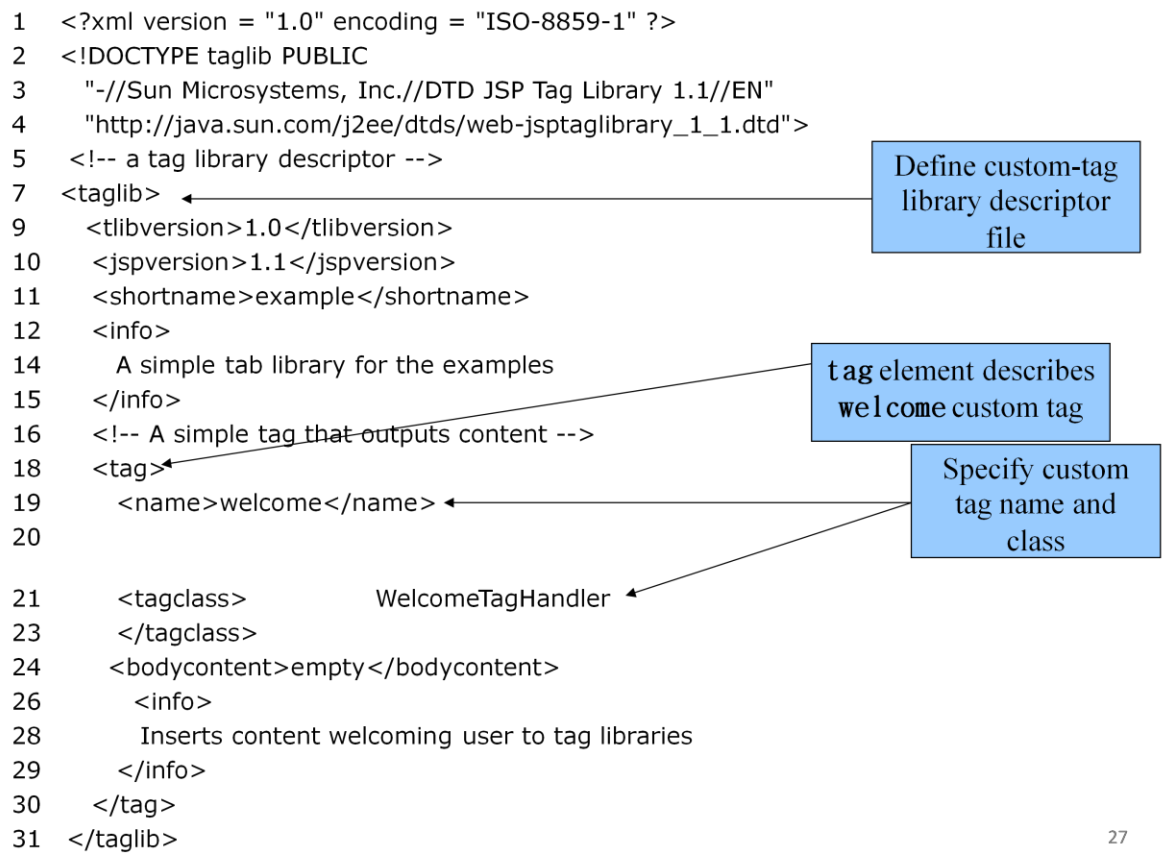


Figure 7. Custom tag library descriptor files example-taglib.tld

NOTE:

Have a glance on the topics:

Introspection, getter and setter methods.

UNIT -8

JSP APPLICATIONS WITH TAG LIBRARIES

1. What is a custom tag library? How is it defined? Give an Example.

Ans. **Custom tag** can be defined as:

- User defined JSP language elements (as opposed to standard tags)
- Encapsulates recurring tasks
- Distributed in a "custom made" tag library, which defines a set of related custom tags and contains the objects that implement the tags.

Why use custom tags?

- Improved separation of presentation and implementation
  - Reduce/eliminate scripting tags
  - Encapsulate common or application-specific page idioms
  - Provide an HTML-friendly layer of abstraction
  - Only three data-oriented JSP actions:

```
<jsp:useBean>
<jsp:setProperty>
<jsp:getProperty>
```

- Custom tags separate presentation from business logic in that page content is created by page designers while the business logic's are captured in the form of custom tags.
- Now custom tags is not just for business logic but it can encapsulate other relatively complex functionality such as complex display, iteration, formatting and so on.

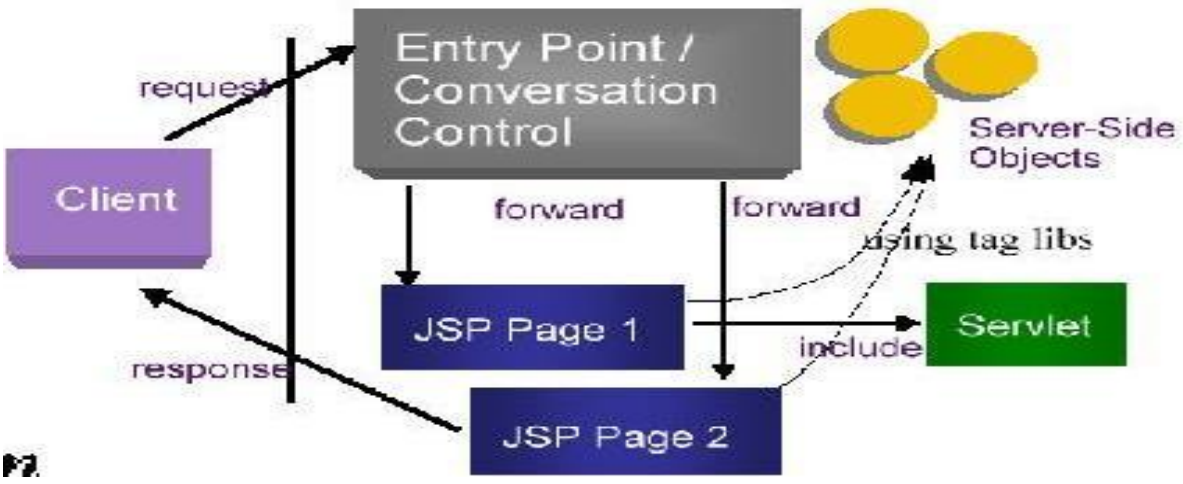


Fig.1. Custom tags fit into the Web application architecture.

- Figure 1 shows how custom tags fit into the Web application architecture.

- Typically HTTP requests coming from the client are handled by the centralized controller, which in turn forwards them to JSP pages. The JSP pages then call server side objects for business logic processing. These server side objects can be either in the form of Java Beans or custom tags.

What Can They Do?

- Generate content for a JSP page
- Access a page"s JavaBeans
- Introduce new JavaBeans
- Introduce new scripting variables
- Any combination of the above
- Plus anything else you can do with Java

Custom tags have a rich set of features. They can

- Be customized via attributes passed from the calling page.
- Pass variables back to the calling page.
- Access all the objects available to JSP pages.
- Communicate with each other. You can create and initialize a JavaBeans component, create a public EL variable that refers to that bean in one tag, and then use the bean in another tag.

- Be nested within one another and communicate via private variables.

Table 1. Custom Tag Attributes

Property	Purpose
Name	The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
Required	This specifies if this attribute is required or optional. It would be false for optional.
Rtexprvalue	Declares if a runtime expression value for a tag attribute is valid
Type	Defines the Java class-type of this attribute. By default it is assumed as **String**
Description	Informational description can be provided.
Fragment	Declares if this attribute value should be treated as a **JspFragment**.

Following is the example to specify properties related to an attribute:

```
.....
    <attribute>
        <name>attribute_name</name>
        <required>false</required>
        <type>java.util.Date</type>
       <fragment>false</fragment>
    </attribute>
.....
```

If you are using two attributes then you can modify your TLD as follows:

```
.....
    <attribute>
       <name>attribute_name1</name>
         <required>false</required>
         <type>java.util.Boolean</type>
         <fragment>false</fragment>
    </attribute>
    <attribute>
         <name>attribute_name2</name>
         <required>true</required>
          <type>java.util.Date</type>
</attribute>
.....
```

2. Introducing the JSP Standard Tag Library (JSPTL)

Sample **Example1:**

Example of forEach tag in jr lib:

```
<%@ taglib uri="http://java.sun.com/jsptl/ea/jr" prefix="jr" %>
<jsp:useBean id="companyPageBean" type="…" />
<jr:forEach var="employee" items="<%=
companyPageBean.getEmployees() %>">
<!--Processing-->
</jr:forEach>
<%@ taglib uri="http//java.sun.com/jsptl/ea/jx" prefix=" jx" %>
<jr:forEach var = "employee" items= "$companyPageBean.employees" >
<!--Processing-->
</jx:forEach>
```

3. Integrating the JSPTL into your JSP page

Example 2

```
<%@ taglib uri=http://java.sun.com/jsptl/ea/jx prefix="jx" %>

<html>

  <head>

        <title>An example JSPTL forEach Tag with Colors>

  </head>

  <body bgcolor="#FFFFFF">

        How does one remember colors of the rainbow?<br><br>

        <table width="300" border="1">

        <tr>

              <td
width="67%"><b><i>Name</i></b></td><td><b><i>RGB

                    Value</i></b></td>

        </tr>

        <jx:forEach var="color" items="$colors">

              <tr bgcolor="#<jx:expr value=„$color.RGBValue"/>">

                    <td><jx:expr value="$color.name"/></td>

                    <td><jx:expr value="$color.RGBValue"/></td>

              </tr>
```

```
            </jx:forEach>
        </body>
</html>
```

4. The JSPTL tags

Table 1 Basic tags

Tag	Attribute	Description
declare		Declares a scripting variable with the specified name and whose value corresponds to the value of a scoped attribute with the same name. This tag provides a bridge, from the explicit form of collaboration of JSPTL tags with their environment (through scoped attributes), to the JSP scripting environment. This tag is available in both the jr and jx tag libraries. Example: <jx:expr id="postcode" type="java.lang.String" />
	Id	As suggested by the JSP 1.1/1.2 Specification, this attribute signifies the name of a scripting variable that will be created by the tag and will be available after the tag within the JSP page.
	Type	The declared type of the scripting variable.

Table 2 Iteration tags

Tag	Attribute	Description
forEach		The main tag for iteration. This tag supports iteration in two main modes: over a list of objects or over a range of values.
		For iteration over a list of objects, a vast number of the standard Java indexed data types are supported, including: arrays and all implementations of java.util.Collection, java.util.Iterator, java.util.Enumeration, java.util.Map, java.sql.ResultSet and Java.util.Hashtable. Also supported is a java.lang.String of comma separated values ("red","green”,"blue").
		We saw the forEach tag in action inthe introduction to the JSPTL section, *integrating* the JSPTL into Your JSP pages: <jx: forEach var=" color " items="$colors" > <TR bgcolor="#<jx:expr value= ' $color .RGBValue„ />”> <TD><jx:expr value="$color.name"/></TD> <TD><jx:expr value="$color.RGBValue" /></TD> </TR> </jx:forEach>
		For iteration over a range, the bodyof the forEach tag is iterated over that number of times. For example, to iterate between 10 and 100, inclusive: <jx:forEach var="n" begin=“10" end="100“> The value is <jx:expr value="$n"/></TD> </jx:forEach>
		It is important to note that the two modes can be combined to itterate over a range of the items in a collection.

	var	The name to assign the scoped attribute that represents the current Item of the collection being iterated over. In the case where the Iteration object is a Java.util.Map, the current item exposed, will be a java.util.Map.Entry object.In the case where the Iteration object is a java.sql.ResultSet the current item exposed will be the java.sql.ResultSet itself, positioned at the next row.
	items	The indexed data type or collection of items to be iterated over.
	status	A status object that represents the current status of the iteration cycle. This is a JSPTL defined type, javax.servlet.jsptl.IteratorTagStatus. This interface is discussed in the next section.
	begin	A range starting value.
	end	A range end value.
	step	Iteration over a range of values will only process values in the range at every 'step' item in the range or collection being iterated over.
forTokens		The forTokens tag is an extension of the forEach tag. It provides the same behaviour, only it iterates over a string of delimited values.
		<jx: forTokens var="color" items="red|green|blue" delims="|"> The color is <jx:expr value="$color" /> </jx: forTokens>
	var	Same behavior as the var attribute of the forEachTag.
	items	A string or an expression-language expression that evaluates to a Java.lang.String containing the delimited values to iterate over.
	status	Same behavior as the status attribute of the forEachTag.

	begin	Same behavior as the begin attribute of the forEachTag.
	end	Same behavior as the end attribute of the forEachTag.
	step	Same behavior as the step attribute of the forEachTag.

Table 3 Iteration Status

Tag	Attribute	Description
current	object	The current item of the list of items iterated over.
index	int	The zero-based index of the current item in the list of items iterated over.
count	int	The one-based position of the current item in the set of items eligible to be iterated over. This value increases by one for each item, regardless of the begin, end, or step attributes of the iteration tag.
		For example, if begin=1, end=10 and step=2 then the set of items eligible for iteration over is 1, 3, 5, 7, and 9. These have position counts of 1, 2, 3, 4, and 5 respectively.
first	boolean	Indicates whether the current item is the first item in the iteration cycle.
last	boolean	Indicates whether the current item is the last item in the iteration cycle.

Table 4 Iteration Tag Extesibility

Tag	Description
interfaceIteratorTag	The main interface for writing custom iteration tags.
interfaceIteratorTag Status	The JSPTL iteration tags provide the status of the iteration cycle within the scope of the tag. Custom iteration tags that confirm to the IteratorTag interface must also provide this information through the IteratorTagStatus object.
abstractclass IteratorTagSupport	Abstract base class for building custom iteration tags. Custom iteration tags just implement the hasNext() and next() methods, providing the stop mechanism of the iteration cycle and the iterated over in sequence.

Table 5 Conditional Tags

Tag	Attribute	Description
if		A simple conditional tag, which only evaluates its body if the supplied condition is *true*. The tag also, optionally, exposes a scoped attribute representing the value of the condition.
		Example: <jx:if var="isMale" test="$employee.sex == „M“”> The test for a employee is a male. </jx:if> <jx:ifvar="isMale"test="$employee.sex == „M“”/> The employee is male was <jx:expr value="$isMale" />
	var	The name to assign the scoped attribute that represents the value of the evaluated condition as a Boolean.

	test	An expression-language expression, if used with the jx tag library or a request-time expression, if used with the jr tag library. The expression must evaluate to a boolean primitive or primitive wrapper Boolean object.
choose		The tag equivalent to the Java switch statement.
		The following example shows its usage: < j x : choose> <jx:when test="$employee.age >= 65"> The employee is a pensioner. </jx:when> <jx:when test="$employee.age >= 40"> The employee is middle-aged. </jx:when> <jx:otherwise> The employee is not middle-aged or a pensioner. </jx:othewise> </jx:choose>
		Only the body of the first when tag whose condition evaluates to true is evaluated. If none of the when conditions is true, the body of an. otherwise tag is evaluated, if there is one.
when		The when tag, only used in context f a parent choose tag.
otherwise		The otherwise tag, only used in context of a parent choose tag.