WEB TECHNOLOGIES

M.Sc. Computer Science

First Year, Semester-II, Paper-V

Lesson Writers

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudramalla
Faculty,
Department of CS&E
Acharya Nagarjuna University

Mrs. Appikatla Pushpa Latha
Faculty,
Department of CS&E
Acharya Nagarjuna University

Dr. U. Surya Kameswari
Assistant Professor
Department ofCS&E
Acharya Nagarjuna University

Editor

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Academic Advisor

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

DIRECTOR, I/c.
Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W., M.Phil., Ph.D.
CENTRE FOR DISTANCE EDUCATION
ACHARYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR 522 510
Ph: 0863-2346222, 2346208
0863- 2346259 (StudyMaterial)
Website www.anucde.info
E-mail:anucdedirector@gmail.com

M.Sc., (Computer Science) : WEB TECHNOLOGIES

First Edition : 2025

No. of Copies :

Publishedby:

Prof. V. VENKATESWARLU
Director, I/c
CentreforDistanceEducation,
AcharyaNagarjunaUniversity

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University.

205CP24: WEB TECHNOLOGIES

SYLLABUS

UNIT-1

HTML common Tags: List, Tables, images, forms, Frames, Cascading Style Sheets; Java Script: Introduction to Java Scripts, Objects in Java Script, Dynamic HTML with Java Script.

UNIT-II

XML-Document type definition, XML Schemas, Document Object model, Presenting XML, Using XML Processors DOM and SAX.

CGI Scripting- What is CGI? - Developing CGI applications - Processing CGI - Returning a Basic HTML page - Introduction to CGI.pm - CGL.pm methods - Creating HTML pages dynamically.

UNIT-III

JDBC: Introduction to JDBC - Connections - Internal Database Connections - Statements - Results Sets - Prepared Statements - Callable Statements.

Network Programming and RMI why networked Java - Basic Network Concepts - looking up Internet Addresses - URLs and URIs - UDP Datagram's and Sockets - Remote Method Invocation.

UNIT-IV

Web Servers, Tomcat web server, Installing the Java Software Development Kit, Tomcat Server & Testing Tomcat, Installing the Java Software Development Kit, Tomcat Server & Testing Tomcat.

Servlets

Introduction to ServletsLifecycle of a Servlet, JSDK, The Servlet API. The javax.servlet Package, Reading Servlet parameters, Reading Initialization parameters. The javax servlet HTTP package, Handling Http Request & Responses, Using Cookies-Session Tracking, Security Issues.

UNIT-V

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

Prescribed Textbooks

1. Web Programming, building internet applications, Chris Bates 2nd edition, WILEYDreamtech (units I, II)
2. Java Programming with JDBC ; Donald Bales, O'Reilly (Unit III)
3. Java Network Programming, elliotte Rusty Harold, 3rd Edition, O'Reilly (Unit III)
4. Java Server Pages - Hans Bergsten, SPD O'Reilly (Unit IV)

Reference Textbooks

1. Robert W. Sebesta, "Programming the World Wide Web", Third Edition, Pearson Education(2007).
2. Anders Moller and Michael schwartzbach, "An Introduction to XML und Web
3. Jeffrey C. Jackson, "Web Technologies - A Computer Science Perspective", Pearson Technologies", Addison Wesley (2006)Education (2008).
4. H.M.Deitel, P.J.Deitel. "Java How to Program", Sixth Edition, Pearson Education (2007).

M.Sc., (Computer Science)
MODEL QUESTION PAPER
205CP24: WEB TECHNOLOGIES

Time: 3 Hours **Max. Marks: 70**

Answer ONE Question from each unit. (5 x 14 = 70M)
All questions carry equal marks.

UNIT – I

1. a) Explain the structure of an HTML document with suitable examples.
   b) Describe various types of lists and tables in HTML with code snippets.

OR

2. a) What is JavaScript? Explain objects in JavaScript with examples.
   b) Discuss the role of Dynamic HTML and demonstrate how JavaScript enhances interactivity.

UNIT – II

3. a) What is XML Schema? Explain its structure and advantages over DTD.
   b) Describe the use of DOM and SAX parsers in XML processing.

OR

4. a) What is CGI? Explain how a CGI application processes form data and returns an HTML page.
   b) Discuss the importance of CGI.pm module and list some of its common methods.

UNIT – III

5. a) Explain the steps for establishing a JDBC connection between Java and a database.
   b) Write a Java program using JDBC to display all rows of a table named *student*.

OR

6. a) Explain UDP datagrams and sockets in Java with an example program.
   b) Discuss the architecture and working of Remote Method Invocation (RMI).

UNIT – IV

7. a) Explain the life cycle of a Servlet with a neat diagram.
   b) Describe how Servlets handle HTTP requests and responses.

OR

8. a) Explain the installation and configuration steps for Tomcat Web Server.
   b) Discuss the concept of session tracking using cookies in Servlets.

UNIT – V

9. a) What are JSP directives? Explain different types of directives with examples.
   b) Describe the role of MVC architecture in JSP application design.

OR

10. a) Explain the anatomy and processing of a JSP page.
    b) Describe the steps for setting up a JSP environment with JDK and Tomcat Server.

CONTENTS

LESSON-1
INTRODUCTION TO HTML AND COMMON TAGS

AIM AND OBJECTIVES:

- Understand the basic structure and purpose of HTML for web page development.
- Identify and apply common HTML tags to create lists, tables, and insert images.
- Design and implement user input forms using appropriate HTML form elements.
- Utilize HTML frames to organize content into multiple sections within a browser window.

Structure:

1.1 HTML

INTRODUCTION

HTML (Hypertext Markup Language) is the standard language used to create web pages or hypertext documents. It is a markup language, meaning it uses a set of instructions known as tags, which are added to text files to format and structure content. Unlike programming languages, HTML is strictly a formatting language - it does not include logic or control structures.

The concept of hypertext allows users to navigate non-linearly through content, enabling easy jumps from one point to another within a document or across multiple documents.

HTML defines the structure and layout of a web page. It is platform-independent, meaning HTML files can be accessed and viewed on any device or operating system with a web browser, regardless of the OS used to create or host the content. As long as there is internet access, users can browse and download HTML files via the World Wide Web (WWW).

In an HTML document, the content is structured using HTML tags, which label different elements of the web page. These tags tell the browser how to display the text, images, and other content. Any content that is not enclosed within tags will be directly displayed on the webpage.

It's important to note that HTML does not define exact page layouts or styles (like font size, headings, etc.) as word processors like Microsoft Word do. Instead, the appearance of HTML elements can vary across different platforms and devices. This is because the formatting is handled by the browser, which adapts the presentation based on the platform's capabilities. By separating content structure from appearance, HTML allows browsers to make layout decisions suitable for their environment.

Web browsers serve as HTML formatters, interpreting HTML code and displaying the content accordingly on screen.

1.1.1 History of HTML

HTML was first created by **Tim Berners-Lee** at **CERN** and gained popularity through the **Mosaic** browser developed at **NCSA**. As the Web grew rapidly in the 1990s, HTML evolved significantly with various enhancements and updates.

To maintain consistency and compatibility among web browsers and developers, standardized versions of HTML were introduced.

- **HTML 2.0** (released in November 1995) standardized the common usage practices of the time.
- **HTML 3.0** (1995) aimed to support more complex web design features but was not widely adopted due to implementation challenges.

Maintaining compatibility across browsers helps reduce development costs and prevents the fragmentation of the Web into proprietary, incompatible systems—ensuring a more unified and accessible experience for all users.

HTML's Future Vision

HTML has always been developed with a vision of universal access. It is designed to support a wide range of devices and platforms—ranging from PCs with high-resolution displays to mobile phones, voice-based devices, and even low-bandwidth systems—ensuring that web content remains accessible and functional across diverse environments.

Advantages of HTML:-

1. **Widely Adopted:** HTML is a universally accepted standard for creating web pages, making it highly reliable and popular.
2. **Cross-Browser Support:** All major web browsers support HTML, ensuring that websites display consistently across platforms.
3. **User-Friendly:** HTML is easy to learn, write, and understand, even for beginners in web development.
4. **No Cost or Special Software:** HTML editing requires only a basic text editor and a web browser, both of which are typically pre-installed on most systems.

Disadvantages of HTML:

1. **Static Content Only:** HTML is limited to creating static pages; it cannot handle dynamic functionality without additional technologies like JavaScript or server-side scripting.
2. **Verbose Coding:** Even simple designs may require writing large amounts of repetitive code.
3. **Weak Security:** HTML lacks built-in security mechanisms, making it dependent on other technologies for secure web applications.
4. **Code Complexity for Larger Pages:** Managing and organizing lengthy HTML code can become complex and error-prone as the size of the webpage increases.

Basic HTML tags:-

(1) **<!doctype>:-** This tag formally starts an HTML document and it also indicates the version of HTML used.

(2) <HTML>:-

Every HTML document starts with a <html> tag and it is always the first tag in a html page and indicates that the document is a HTML document. The end tag <html> is </html>.

Example:

<html> …….. </html>

(3) <head>:-

It contains the head of an html document, which holds about the document such as title. Each property defined html page should have a head which we create with <head>tag. It has header information and it is displayed at the top of the browser. Each tag for <head> is</head>.

<head>..……. </head>

(4) <title>:-

It contains the title of the html document which includes the content that will actually appear in the web browser. The entire content of the web page is placed in the pages <body> tag. The end tag <body> is </body>.

<title>…… </title>

(5) <body>:-

It contains the body of the HTML Document, which includes the content that will actual appear in the web browser. The entire content of the webpage will be placed in the pages <body>tag. The end tag of the <body>tag will be </body>

<body>.……</body>

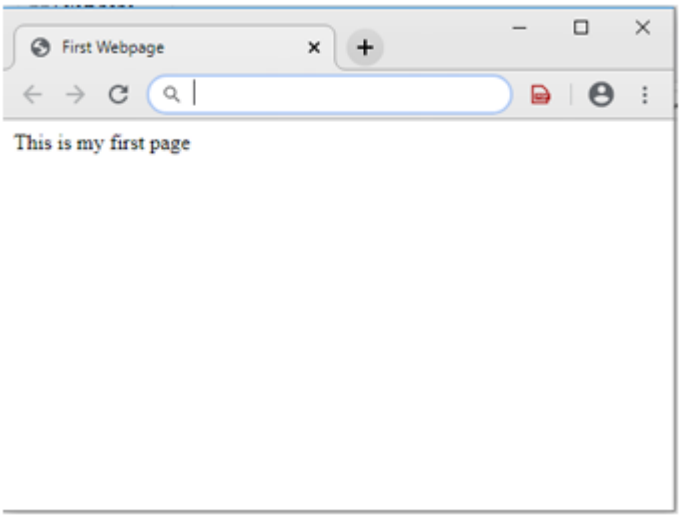
STRUCTURE OF THE HTML PROGRAM:-

The HTML Program is generally divided into two sections i.e head and body. We use <head> and <body> tags to indicate these two sections. <head> section holds the header information of a webpage document indicated by a title that is provided by using <title> tag in the <head>. The title helps us to refer to the webpage. <body> section contains the content which we want to display which the webpage. Anything that is not a tag will be displayed within the webpage.

Example:

```
<html>
    <head>
        <title>First Webpage</title>
    </head>
    <body>
        This is my first page
    </body>
</html>
```

Output:



Attribute:

An Attribute is a Keyword we use in an opening tag to give more information to the web browser. HTML tags tell the web browsers how to format and organize our webpages. But we can customize tags using attributes. The Format of an attribute is:

<tagname Attribute=value>

Attributes of the <body> tag:

(1) **Background:**
The URL or a graphic file to be used in the filling the browser's Background.

(2) **Bgcolor:**
The color of the browser's background.

(3) **Bgproperties:**
It Indicates if the background should scroll when text does. If we set it to "FIXED", the background will not scroll when the text does.

(4) **Bottommargin:**
Specifies the bottom margin,the empty space at the bottom of the document in pixels.

(5) **Id:**
It is a unique alphanumeric identifier for the tag which we can use to refer to it.

(6) **Language:**

Scripting language used for the tag.
(7) **Leftmargin:**
Specifies the left margin, the empty space at the left of the document.
(8) **Marginheight:**
Gives the height of the margin at the top and bottom of the page in pixels.
(9) **MarginWidth:**
Gives the width of the left and right margins of the page in pixels.
(10) **Rightmargin:**
It specifies the right margin, the empty space to the right margin of the document in pixels.
(11) **Scroll:**
It specifies whether a vertical scrollbar appears to the right of the document can be yes (or) no.
(12) **Style:**
Inline style indicating how to render the element.
(13) **Text:**
Color of the in the document.
(14) **Topmargin:**
It specifies the top margin the space at the top of the document in pixels.
(15) **Link:**
It specifies the color of hyperlinks that have not yet been visited.
(16) **Alink:**
It specifies the color of hyperlinks as they are being clicked.
(17) **Vlink:**
It specifies the color of hyperlinks as they have been visited.
(18)**<!-- --> Comment tag:**
Annotates a web page with a comment. In the HTML that we can by looking at the HTML but it will not be displayed in the web browser.
<! ------ This is a comment --------- >

1.2 Common Tags

Common HTML Tags:

1. <html> – Root element that defines the entire HTML document.
2. <head> – Contains metadata and links to external resources like stylesheets and scripts.
3. <title> – Specifies the title of the web page (appears in the browser tab).
4. <body> – Contains the main content of the webpage that is visible to users.
5. <h1> **to** <h6> – Heading tags, where <h1> is the largest and <h6> is the smallest.
6. <p> – Defines a paragraph.
7.
 – Inserts a line break.
8. <hr> – Inserts a horizontal line (thematic break).
9. <a> – Creates a hyperlink. Example: Visit
10. <img> – Embeds an image. Example: <imgsrc="image.jpg" alt="Image">
11. – Creates an unordered (bulleted) list.
12. – Creates an ordered (numbered) list.
13. – List item (used inside <ul> or <ol>).
14. <table> – Defines a table.
15. <tr> – Defines a table row.
16. <td> – Defines a cell in a table row.

17. <th> – Defines a header cell in a table.
18. <form> – Creates a form for user input.
19. <input> – Accepts user input inside a form.
20. <button> – Defines a clickable button.

1.3 Lists:
Lists lets us display information in a compact, right format. There are three kinds of lists:

1. ordered List
2.UnOrdered List
3. Definition List

Type	**Tag**	**Description**
Ordered		Displays list items in a specific order (numbered: 1, 2, 3… or i, ii, iii…).
Unordered		Displays items with bullet points.
Description	<dl>	Used for name/value pairs, like in dictionaries or glossaries.

Ordered List:
Ordered lists use a number system / lettering scheme to indicate that the items are ordered in some ways, ordered lists are created by <ol> tag and the list items are created using <li> tag.
Example:

```
<html>
<head>
<title> Creating Order List </title>
</head>
<body bgcolor=”pink”>
<h1 align=”center”> Creating Order List</h1>
<h1 align=”center”>List of branches in RGMCET</h1>
<ol>
<li>CSE</li>
<li>IT</li>
<li>ECE</li>
<li>EEE</li>
<li>CIVIL</li>
<li>ME</li>
</ol>
</body>
</html>
```

Output

Creating Customized Ordered Lists:-

We can customize the numbering system used in ordered lists by using the TYPE attribute, which we can set to these values:

1. Default numbering system (1, 2, 3, ….)
A. Uppercase Letters (A, B, C, …..)
a. Lowercase Letters (a, b, c, …)
I. Large Roman Numerals (I, II , III, ….)
i. Small Roman Numerals (i, ii, iii, …..)

Example:

```
<ol>
        <li>Wake up</li>
        <li>Brush teeth</li>
        <li>Eat breakfast</li>
</ol>
```

Output:

1. Wake up
2. Brush teeth
3. Eat breakfast

Unordered List:

An Unordered list is a list of items that are marked with burden. The Unordered list is created by using tag are the list items in the list are created by tag and the list items in the list are created by <li> tag.

```
<ul>
<li>List Item 1 </li>
<li>List Item 2 </li>
</ul>
```

Example:

```
<ul>
        <li>Milk</li>
        <li>Eggs</li>
        <li>Bread</li>
</ul>
```

Output:

- Milk
- Eggs
- Bread

Creating Customized Unordered Lists:
We customized unordered lists by setting the "Type" attribute to three different values. DISC (default), SQUARE and CIRCLE which sets the type of bullet that appears before the list item.

Example:

```
<html>
<head>
<title> Creating Unorder List </title>
</head>
<body bgcolor="pink">
<h1 align="center"> Creating Unorder List</h1>
<h1 align="center">List of Colleges in Kurnool</h1>
<ul type="square">
<li>GPREC</li>
<li>RGMCET</li>
<li>GPCET</li>
</ul>
</body>
</html>
```

Output:

Definition List:-
These lists include both definition terms as well as their definition. To create the definition lists we use <dl> tag. For creating definition terms we use <dt> tag and for data definitions we use <dd> tag.

```
<dl>
        <dt>HTML</dt>
        <dd>HyperText Markup Language</dd>
```

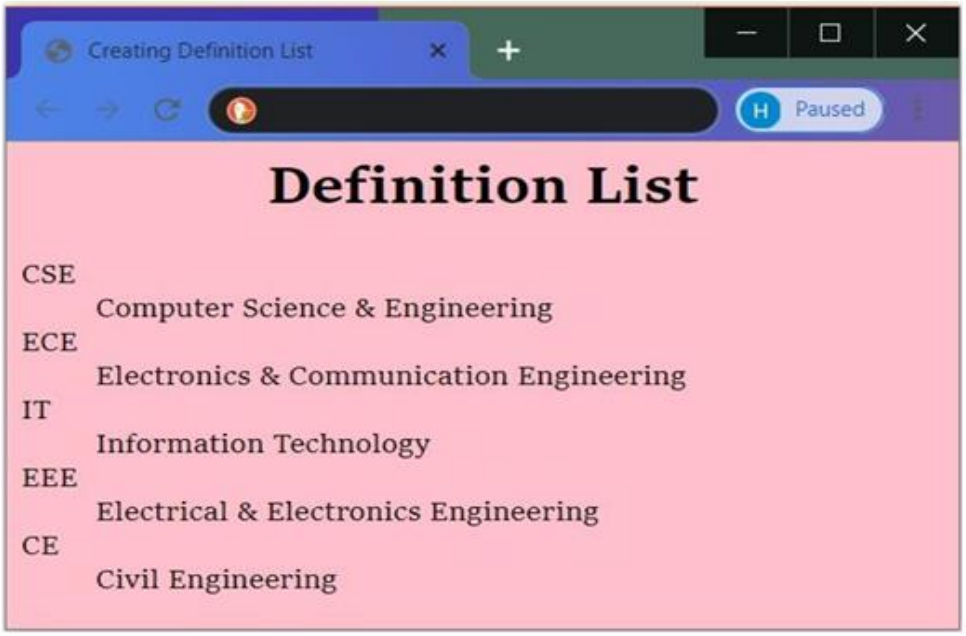
```
        <dt>CSS</dt>
        <dd>Cascading Style Sheets</dd>
</dl>
```

- <dt>: Defines the term (name)
- <dd>: Defines the description (value)

Example:

```
<html>
<head>
<title>Creating Definition List</title>
</head>
<body bgcolo=”pink”>
<h1 align=”center”>Definition List</h1>
<dl>
<dt>CSE<dd>Computer Science & Engineering
<dt>ECE<dd>Electronics & Communication Engineering
<dt>IT<dd>Information Technology
<dt>EEE<dd>Electrical & Electronics Engineering
<dt>CE<dd>Civil Engineering
</dl>
</body>
</html>
```

Output:



Nesting Lists:-

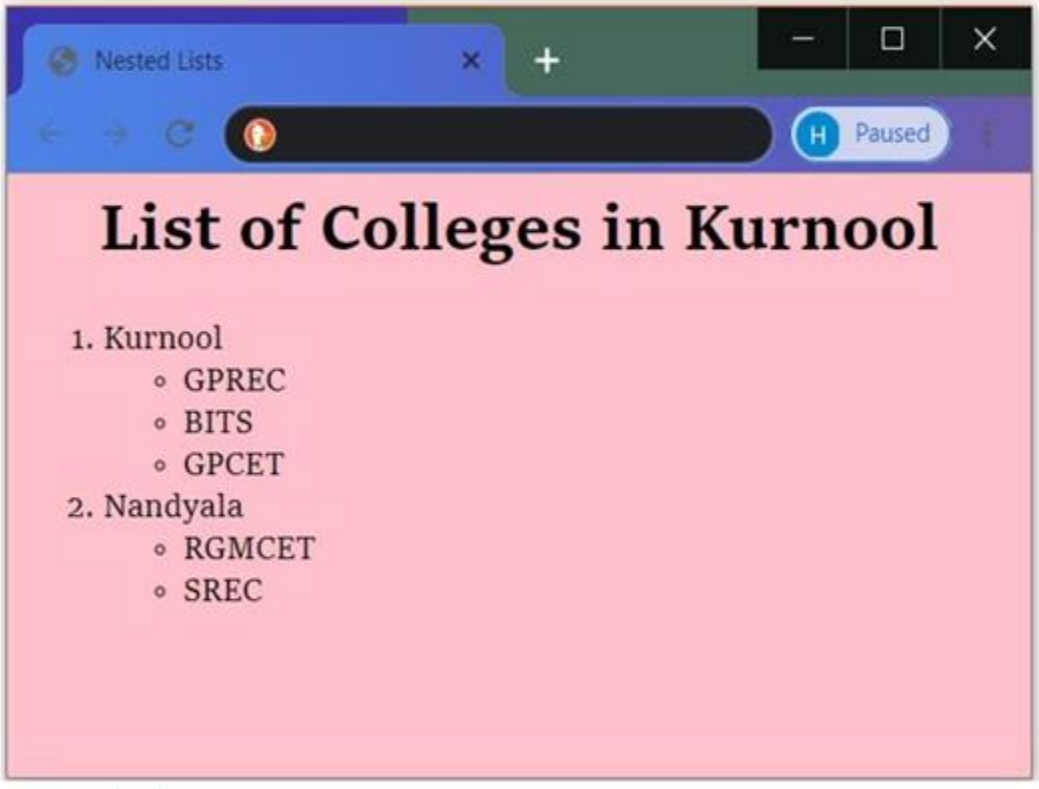
We have the capability of nesting lists inside other lists.

Example:

```
<html>
<head>
<title>Nested Lists</title>
</head>
```

```
<body bgcolor=”pink”>
<h1 align=”center”>List of Colleges in Kurnool</h1>
<ol>
<li>Kurnool</li>
<ul>
<li>GPREC</li>
<li>BITS</li>
<li>GPCET</li>
</ul>
<li>Nandyala</li>
<ul>
<li>RGMCET</li>
<li>SREC</li>
</ul>
</ol>
</body>
</html>
```

Output:

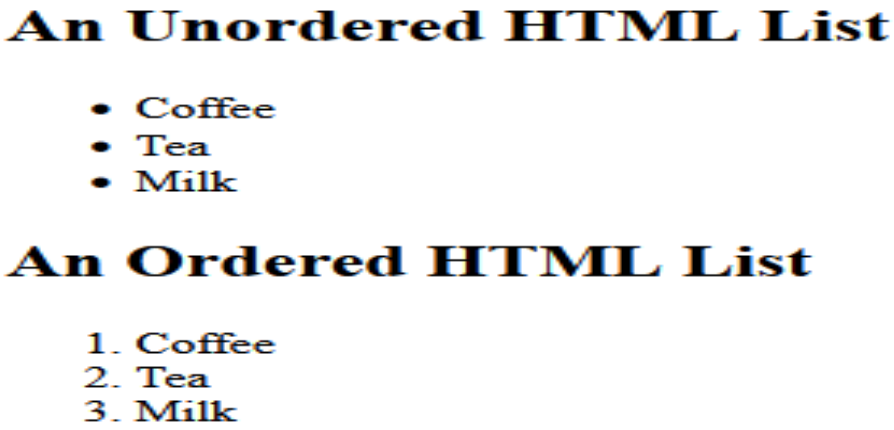


Example program:

```
<!DOCTYPE html>
<html>
<body>
<h2>An Unordered HTML List</h2>
<ul>
                    <li>Coffee</li>
                    <li>Tea</li>
```

```
                    <li>Milk</li>
</ul>
<h2>An Ordered HTML List</h2>
<ol>
                    <li>Coffee</li>
                    <li>Tea</li>
                    <li>Milk</li>
</ol>
</body>
</html>
```

Output:



Creating Hyperlinks:

What makes the web so effective is the ability to define links from one page to another. In web terms, a "hyperlinks" is a reference on the web. Hyperlinks can point to any resources on the web. An anchor is a term used to define a hyperlink destination inside a document. Format of anchor tag is:

 Line Text

The <a> anchor tag has the following attributes.

1. href: It holds the target URL of the hyperlink.
2. Id: A unique alphanumeric identifier for the tag, which we can use to refer to it.
3. name: It specifies an anchor name, the name we want to use when referring to enclose items.
4. Target: This attribute defines where the linked document will be opened.

Example:

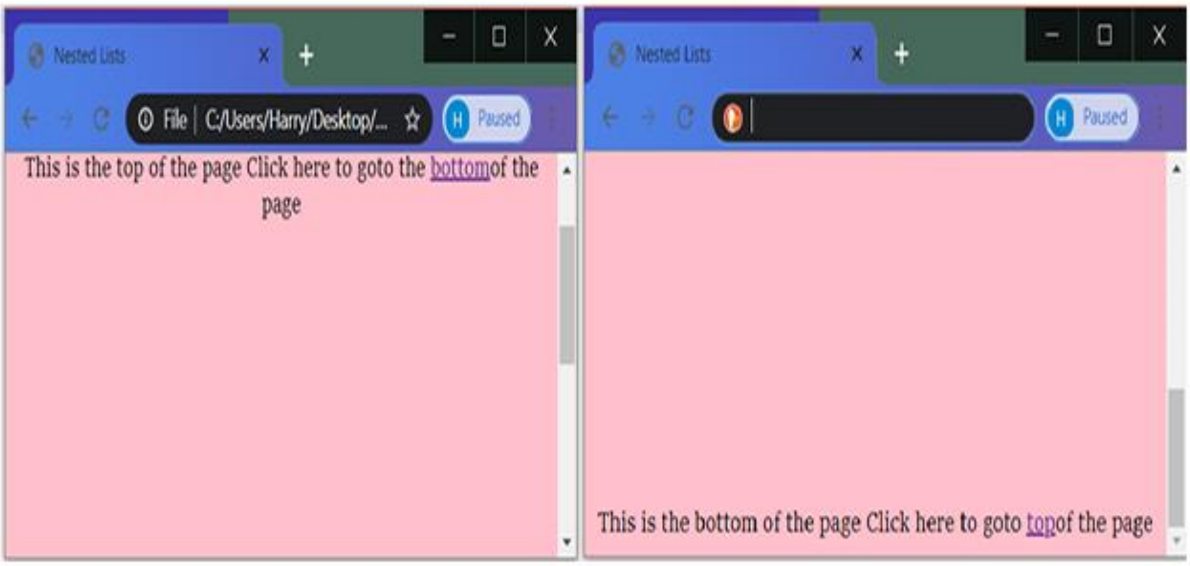
```
<html>
<head>
<title>Creating Hyper Links</title>
</head>
<body bgcolor="pink">
<center><h1>This is page 1</h1>
<a href="page2.html">Click here</a>to goto page2
</center>
</body>
</html>
```

Output:

Providing navigation with in the page:

```
<html>
<head>
<title>Nested Lists</title>
</head>
<body bgcolor="pink">
<center><h1>Linking to a section in a page</h1>
<a name=”top”>This is the top of the page</a>
Click here to goto the <a target=”#bottom”>bottom</a>of the page
<br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br>
<br><br><br>
<a name=”bottom”>This is the bottom of the page</a> Click here to
goto<a target=”#top”>top</a>of the page
</center>
</body>
</html>
```

Output:



1.4 TABLES

Creating HML tables:

A HTML table arranges data/information in terms of rows and columns.Tables are defined in HTML using <table>tag.A table is divided into rows and each row each divided into data cells(columns).The rows of table are created using <tr> tag and data cells are created by <td> tag.

<tr> - Table row
<td> - Table data

Format

```
<table>
<tr>
<td>row1,col1</td>
<td>row1,col2</td>
</tr>
<tr>
<td>row2,col1</td>
<td>row2,col2</td>
</tr>
</table>
```

❖ Heading in a table are defined with <th> tag

Format

```
<table>
<tr>
<th>heading 1</th>
<th>heading 2</th>
</tr>
<tr>
<td>data1</td>
<td>data2</td>
</tr>
</table>
```

Attributes of <table> tag:

- align : specifies the horizontal alignment of the table in the browser window, set to “left,center,right”.
- background : specifies the URL of a background image to be used as background for the table.
- bgcolor : sets the background color of the table cells.
- border : sets the border width.
- bordercolor : sets the external border color of the entire table.
- cellpadding : sets the spacing between cell walls and content.
- cellspacing : sets the spacing between table cells.
- height : sets the height of the whole table.
- width : sets the width of the table.

Attributes of <tr> tag:

- align : specifies the horizontal alignment content in the table cells set to “left,center,right”.
- bgcolor : sets the background color of the table cells.
- bordercolor : sets the external border color of the entire table.
- Valign : sets the vertical alignment of data,set to top,middle,bottom.

Alignment of <td> tag

- align : specifies the horizontal alignment content in the table cells set to "left,center,right".
- bgcolor : sets the background color of the table cells.
- bordercolor : sets the external border color of the entire table.
- colspan : indicates the how many cell columns of the table this cell should span.
- rowspan : indicates the how many cell rows of the table this cell should span.

Example

```
<html>
<head>
<title>Creating Tables</title>
</head>
<body bgcolor="pink">
<center><h1>Creating tables</h1>
<table border="1" cellpadding="3" cellspacing="3">
<tr>
<thcolspa="2">Websites</th>
</tr>
<tr>
<td>Mail sites</td>
<td>Job sites</td>
</tr>
<tr>
<td>Gmail.com</td>
<td>Frushersworld.com</td>
</tr>
<tr>
<td>Yahoo.com</td>
<td>Nauted.com</td>
</tr>
</center>
</table>
</body>
</html>
```

Output:

Advanced Table elements :

- <caption> : the element is an ptional element and it used to provide a string which describes,the contern of the table ,it must follow the table element.
- <thead> : The rows in a table can bcgroudated one are more time we can create a tableby using this <thead>.
- <tbody> : creates a table body when grouping rows.
- <tfoot> : Creates a table foot when groupin rows

Example:

```
<html>
<head>
<title>Advance Table Elements</title>
</head>
<body bgcolor="pink">
<h1 align="center">Contents of Web Technologies</h1>
<center>
<table border="2">
<caption>Subject Description</caption>
<thead>
<tr><td colspan="2">Advance Java Programming</td>
</thead>
<tbody>
<tr>
<td>Units</td>
<td>Contents</td>
</tr>
<tr><td>I</td>
<td>HTML & CSS</td>
</tr>
<tr><td>II</td>
<td>JavaScript</td>
</tr>
<tr><td>III</td>
<td>XML</td>
</tr>
</tbody>
<tfoot align="center">
<tr>
<td colspan="2">The table foot</td>
</tr>
</tfoot>
</table>
</center>
</body>
</html>
```

Output:

Nesting of Tables:

```
<html><head><title>Nesting of Tables</title></head>
<body bgcolor="pink">
<center><h1>Nested tables</h1>
<table border="1" cellpadding="3" cellspacing="3">
<tr><td>
<table border="2">
<tr><th>Mail sites</th>
<th>Job sites</th></tr><tr></tr><tr>
<td>Gmail.com</td>
<td>Frushersworld.com</td>
<td>Yahoo.com</td>
<td>Nauted.com</td>
</tr></table>
</td>
<td>
<table border="2">
<tr><th>Number</th>
<th>Words</th></tr><tr>
<th>1</th>
<th>One</th></tr><tr>
<th>2</th>
<th>Two</th></tr>
</table>
</td>
</tr>
</table>
</center>
</body>
</html>
```

Output:

1.5. IMAGES

Images in HMTL:

In HTML we have the capability of displaying images in a webpage.This images must be in a format that the web browser can handle, such as Graphics Interchange Format (GIF), Joint Photograph Expert Group (JPEG), and for some browser Portable Network Graphics(PNG) formats.
Displaying images in webpage is done by using <img> tag

Format
<imgsrc=”URL of image source”>
Attributes of <img> tag:

- alt: this attribute is used to specify text to be displayed in place of image for browser that cannot handle graphics.
- src: specifies the URL of the image to display.
- border: sets the border for the image.
- height: indicates the height of the image.
- width: indicates the width of the image.
- hspace: sets the horizontal space around the image.
- vspace: sets the vertical space around the image.

Example :

```
<html>
<head>
<title>Images</title>
</head>
<body bgcolor="pink">
<center>
<h1>Images Example</h1>
<h3>Here is an image</h3>
<imgsrc="one.jpg" alt="here is an image" width="300" height="150">
</center>
</body>
</html>
```

Output:

1.6 FORMS

Creating HTML Forms:

Form is a collection of various HTML control files , buttons ,checkboxes , radio buttons ,text fields et., and they use to send the data to the server. There are several form elements.

- **Button :**<input type=”button”>:- are the standard clickable buttons.
- **Checkbox :**<input type=”checkbox”>:- displayed usually as a small box with a check mark in it. The use can toggle the checkbox on or off by checking the checkbox
- **Customizable Buttons :**<button>:- display images one other HMTL inside itself.
- File uploading controls : <input type=”file”>:- allow the user to upload files to the server.
- **Hidden controls:** store data that is not visible to users unless they view the web page source code.
- **Image controls :**<input type=”image”>:- are like submit buttons except that they are images the user can click.
- **Password controls** : <input type=”password”>:- are like text fields , but each typed character displaying by an asterisk or instead any character.
- **Radio buttons :**<input type=”radio”>:- displaying usually as a circle which when selected displayed a dot in the middle. These controls are much like checkboxes except that they work in it mutually exclusive at a time.
- **Reset button :**<input type=”reset”>:- allow the user to clear all the data they has entered.When the user click reset button all controls in the form are removed to that original state displaying the data they had when they first appeared.
- **Selection :** Works much like drop down list boxes also called select controls Format is:

```
<select>
<option>Item1</option>
<option>Item1</option>
<option>Item1</option>
</select>
```

- **Submit button:** when we click the button all the data in the form will be sent to web server for processing.

Text area: are two dimensional text fields allowing user to enter more than one line of text. Format is: <textarea>

Text fields : allow the user to enter one line of text also called a textbox Format is :

<input type=”text”>
In order to create form we use <form> tag Format is:
<form>
|
|
</form>

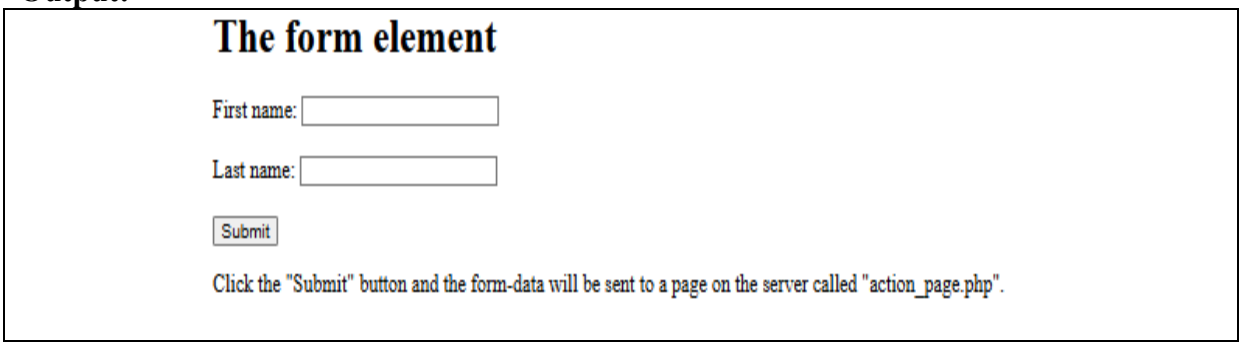
Attributes of <form> tag:

- **name:** gives the name of the form so that we can return it in code. Set to an alphanumeric string.
- **target:** indicates a named frame for browser to display the form results.
- **method:** indicates a method or protocol for sending data to the target action URL.
- **action:** gives the URL that that will handle the form data.

Example program:

```
<!DOCTYPE html>
<html>
<body>
<h1>The form element</h1>
<form action="/action_page.php">
        <label for="fname">First name:</label>
                <input type="text" id="fname" name="fname"><br><br>
<label for="lname">Last name:</label>
                <input type="text" id="lname" name="lname"><br><br>
                <input type="submit" value="Submit">
</form>
<p>Click the "Submit" button and the form-data will be sent to a page on the server called "action_page.php".</p>
</body>
</html>
```

Output:



1.7 FRAMES

Frames in HTML are used to divide the web browser window into multiple sections, where each section can load a separate HTML document. This allows for the display of multiple web pages within a single browser window. Frames were commonly used to create navigation menus that stay static while content changes in another section.

Key Concepts of Frames

1. **Frameset**:
   - Replaces the <body> tag in a frames-based page.
   - Defines how the screen is split — vertically (cols) or horizontally (rows).

Syntax:

```
<frameset cols="30%, 70%">
<frame src="menu.html">
<frame src="content.html">
</frameset>
```

2. **Frame**:
   - Defines each individual frame within a frameset.
   - Uses attributes like src, name, scrolling, and frameborder.

Example:

```
<frame name="menu" src="menu.html" scrolling="yes" frameborder="0">
```

3. **NoFrames**:
   - Provides alternative content for browsers that do not support frames.

Syntax:

```
<noframes>
  Your browser does not support frames.
</noframes>
```

Attributes of <frame> Tag

Attribute	Description
src	Specifies the HTML file to display in the frame.
name	Gives the frame a name to be targeted by links.
scrolling	Sets scrollbar visibility: yes, no, or auto.
frameborder	Controls border visibility: 0 (no border), 1 (with border).

Advantages of Using Frames

- Allows independent scrolling in different parts of the screen.
- Useful for persistent navigation menus.
- Reduces the need to reload the entire page.

Disadvantages of Using Frames

- **Deprecated in HTML5** — modern web development discourages frames.
- Not SEO-friendly; search engines may not index all content properly.
- Bookmarking specific frames is difficult.
- Navigation issues (e.g., back button behavior) may confuse users.

Note: While frames were once popular for building complex web layouts, they have been largely replaced by modern techniques using **CSS**, **JavaScript**, and responsive design principles. Frames are no longer supported in HTML5, so developers are encouraged to use <iframe> or layout techniques like **Flexbox** and **Grid**.

Example:

Let's put above example as follows, here we replaced rows attribute by cols and changed their width. This will create all the three frames vertically:

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

```
<head>
<title>HTML Frames</title></head>
<frameset cols="25%,50%,25%">
<frame name="left" src="/html/top_frame.htm" />
<frame name="center" src="/html/main_frame.htm" />
<frame name="right" src="/html/bottom_frame.htm" />
<noframes>
<body>
Your browser does not support frames. </body> </noframes></frameset>
                    </html>
```

This will produce following result:

Output:

1.8 SUMMARY

HTML is the standard language used to create and structure web pages using predefined tags. Originally developed by Tim Berners-Lee, it evolved from SGML and gained popularity with the Mosaic browser. HTML versions like 2.0 and 3.0 were introduced to improve browser compatibility and standardize web content.HTML's design emphasizes universal access, making it suitable for a wide range of devices, including those with limited bandwidth. It is popular due to its simplicity, cross-platform support, and no cost of development. However, it is mainly used for static content, lacks built-in security, and can become complex for large web pages.An HTML document follows a basic structure comprising <html>, <head>, and <body> tags. Key elements include headings (<h1> to <h6>), paragraphs (<p>), links (<a>), images (<img>), and forms. HTML also supports different types of lists—ordered (<ol>), unordered (<ul>), and description (<dl>), as well as tables using tags like <table>, <tr>, <td>, and <th>. The <img> tag is used for images, with src and alt attributes being essential for proper display and accessibility.HTML promotes semantic structure and accessibility by encouraging the use of meaningful tags, ensuring content is both human- and machine-readable.

1.9 KEY TERMS

HTML (HyperText Markup Language),SGML (Standard Generalized Markup Language),XML (eXtensible Markup Language),Tags, Paragraph, Anchor/hyperlink, Image, Lists, Table, Form, Frame.

1.10 Self-Assessment Questions

1. What is the purpose of a <form> tag in HTML?
2. What does the action attribute in a form specify?

3. How does the method="get" work in a form?
4. When should you use the method="post"?
5. What tag is used to take user input in a form?
6. What is the difference between <input type="text"> and <textarea>?
7. What attribute is used to group form controls?
8. What is the use of the name attribute in an input field?
9. How can you prefill values in form elements?

1.11 Further Readings

1. Web Programming, building internet applications, Chris Bates 2nd edition, WILEY Dreamtech
2. Java Programming with JDBC; Donald Bales, O'Reilly
3. Java Network Programming, elliotte Rusty Harold, 3rd Edition, O'Reilly
4. Java Server Pages - Hans Bergsten, SPD O'Reilly
5. Robert W. Sebesta, "Programming the World Wide Web", Third Edition, Pearson Education(2007).
6. Anders Moller and Michael schwartzbach, "An Introduction to XML und Web
7. Jeffrey C. Jackson, "Web Technologies - A Computer Science Perspective", PearsonTechnologies", Addison Wesley (2006) Education (2008).
8. H.M.Deitel, P.J.Deitel. "Java How to Program", Sixth Edition, Pearson Education (2007).

Dr. Kampa Lavanya

LESSON-2

CASCADING STYLE SHEETS

OBJECTIVES:

- To enhance the visual appearance of web pages.
- To separate structure (HTML) from presentation (CSS).
- To allow consistent styling across multiple pages.
- To make web development and maintenance easier and faster.

Structure:

2.1 CASCADING STYLE SHEETS:

Style sheets represent the World Wide Web consortium's effort to improve on the tag and attribute based style of formatting. Style sheets provide a way of customizing whole pages all at once and in much richer detail than the simple use of tags and attributes. The format of style sheet will be:

```
<style type="text/css">
selector{property:value;property:value;}
selector{property:value;property:value;}
</style>
```

Every line in <style> tag is called as a „Rule" and a style rule has two parts:

a. Selector.

b. Set of declarations.

A selector is used to create a link between the rule and the HTML tag. The declaration has two parts again:

a. Property.

b. Value.

A property specifies additional information and value specifies property value. For example:

```
<style type="text/css">
```

```
body {background-color: #d0e4fe;}
h1 {
color: orange;
text-align: center;
}
p {
font-family: "Times New Roman";
font-size: 20px;
}
</style>
```

If we add above code in the <head> element of web page, entire web page will be displayed in various styles given in style element.

Style sheets are implemented with cascading style sheets specification. Conventionally styles are cascaded i.e., we don't have to use just a single set of styles inside a document, but we can import as many styles as we like. There are three mechanisms by which we can apply styles to our HTML documents:

1. Inline Style sheets.
2. Embedded Style sheets.
3. External Style sheets.

2.1.1 Inline Style Sheets:

Inline style sheets mix content with presentation. To use inline styles we use style attribute in the relevant tag.

- Applied directly within an HTML tag using the style attribute.
- Affects only the specific element.

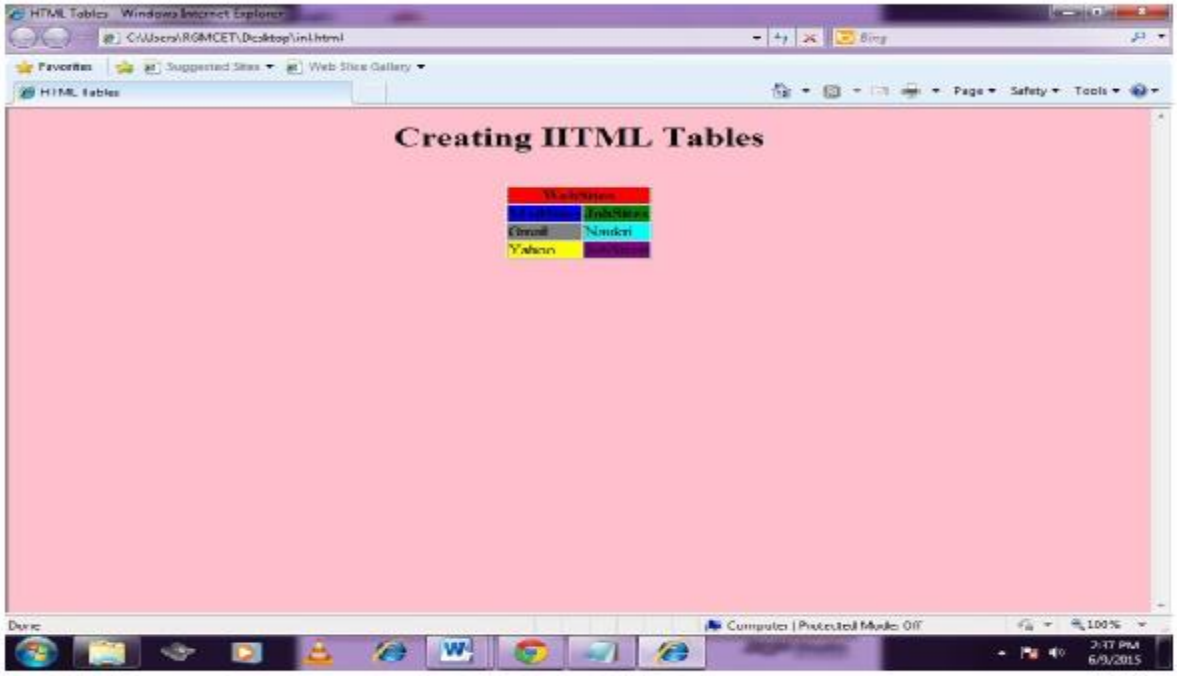
Ex: This is inline styled text.

Example:

```
<html>
<head>
<title>HTML Tables</table>
</head>
<body bgcolor="‘pink‘">
<center>
<h1>Creating HTML Tables</h1><br>
<table border=”2” cellpadding=”4” cellspacing=”4”>
<tr>
<thcolspan=”2” style=”background-color:red”>WebSites</th>
</tr>
<tr>
<th style=”background-color:blue”>MailSites</th>
<th style=”background-color:green”>JobSites</th>
</tr>
<tr>
<td style=”background-color:grey”>Gmail</td>
<td style=”background-color:aqua”>Naukri</td>
</tr>
<tr>
<td style=”background-color:yellow”>Yahoo</td>
```

```
<td style=”background-color:purple”>JobStreet</td>
</tr>
</table>
</center>
</body>
</html>
```

Output:



2.1.2 Internal (or) Embedded Style sheets:

An embedded style sheet is used when a single document has a unique style. We define internal styles in the head section of a HTML page by using „<style>“ tag. The styles defined using embedded style sheets are applied throughout the page and we put the styles into one place.

- Defined within a <style> tag in the <head> section of the HTML document.
- Affects all elements on that page.

Example-1:

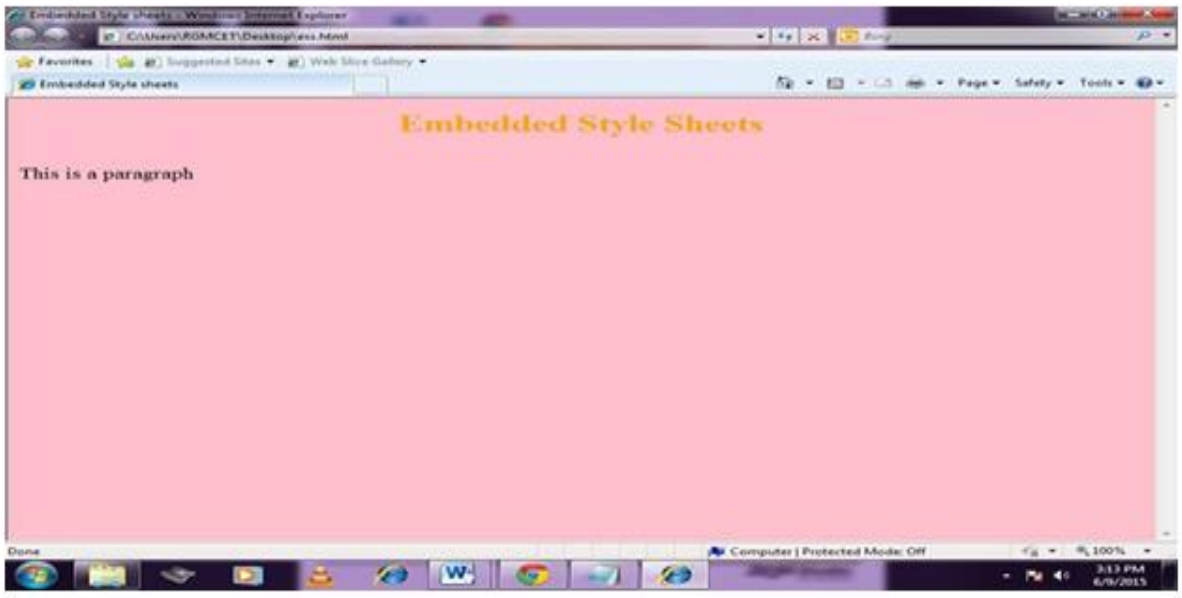
```
<head>
        <style>
                h1
                {
color: green;
text-align: center;
        }
</style>
</head>
```

Example-2:

```
<html>
```

```
<head>
<title>Embedded Style sheets</title>
<style type=”text/css”>
body{background-color:pink;}
h1 {
color:orange;
text-align:
center;
}
p {
font-family: "Times New
Roman";
font-size: 20px;
}
</style>
</head>
<body>
<h1>Embedded Style Sheets</h1><br>
<p>This is a paragraph
</body>
</html>
```

Output:



2.1.3 External Style Sheets:

External style sheets are just that the style sheets are stored separately from our web page. These are useful especially if we are setting the styles for an entire website. When we change the styles in external style sheet we change the styles of all pages. We use „<link>“ element to access the style sheet file defined into our web page. The format of <link> element is:
<link rel=”stylesheet” type=”text/css” href=”extstylesheet.css”>

- Written in a separate .css file and linked to HTML using the <link> tag.
- Can be used across multiple HTML pages.

Example (in HTML):

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

Example (style.css):

```
body {
background-color: #f0f0f0;
font-family: Arial;
}
```

Example program:

extern.css:

```
body {background-color: #d0e4fe;}
h1 {
color: orange; text-align: center;
}
p {
font-family: "Times New Roman"; font-size: 20px;
}
```

extern.html:

```
<html>
<head>
<title>External Style Sheets</title>
<link rel=”stylesheet” type=”text/css” href=”extern.css”>
</head>
<body>
<h1>External Style Sheets</h1><br>
<p>This is a paragraph
</body>
</html>
```

Output:

Cascading Order (Priority)
When multiple styles apply to the same element, CSS follows a specific order of precedence:

1. **Inline CSS** (Highest priority)
2. **Internal CSS**
3. **External CSS**
4. **Browser default styles** (Lowest priority)

The term *"cascading"* refers to this order of priority.
CSS Syntax:

selector {
property: value;
property: value;

}
Selector: The HTML element you want to style.
Property: The style attribute you want to change.
Value: The value you want to assign to that property.

Example:
p {
color: red;
font-size: 14px;
}

2.1.4 Common CSS Properties

Property	Description	Example
color	Text color	color: blue;
background-color	Background color	background-color: yellow;
font-size	Size of the text	font-size: 18px;
font-family	Font type	font-family: Arial;
text-align	Alignment of text	text-align: center;
margin	Space outside the element	margin: 20px;
padding	Space inside the element	padding: 10px;
border	Border style	border: 1px solid black;
width/height	Size of an element	width: 100px;

2.2 CSS SELECTORS

CSS selectors are used to "select" the HTML elements you want to style.

1. Universal Selector:

```
* {
margin: 0;
}
```

2. Element Selector:

```
h1 {
color: blue;
}
```

3. Class Selector:

```
.myclass {
font-size: 18px;
}
```

Used in HTML like: Text

4. ID Selector:

```
#myid {
color: green;
}
```

Used in HTML like: Box

5. Group Selector:

```
h1, h2, p {
color: red;
}
```

6. Descendant Selector:

```
div p {
color: orange;
}
```

2.3 BOX MODEL IN CSS

Every HTML element is treated as a box with four components:

1. **Content** – The actual text or image.
2. **Padding** – Space between content and border.
3. **Border** – Surrounds the padding (if any) and content.
4. **Margin** – Space outside the border.

Box Model Example:

```
div {
padding: 10px;
border: 1px solid black;
margin: 20px;
}
```

Advanced CSS Features

Pseudo-classes (e.g., :hover, :first-child)

```
a:hover {
color: red;
}
```

Pseudo-elements (e.g.,::before, ::after)

```
p::first-letter {
```

```
font-size: 30px;
}
```
Media Queries (for responsive design)
```
@media screen and (max-width: 600px) {
body {
background-color: lightblue;
  }
}
```
Animations and Transitions
```
div {
transition: all 0.3s ease-in-out;
}
```
Flexbox and Grid Layouts (modern layout techniques)

Advantages of CSS

- Enhances presentation and user experience.
- Reduces repetition and saves time.
- Makes maintenance easier.
- Supports responsive web design.
- Allows separation of concerns (structure vs. presentation).

Disadvantages of CSS

- Browser compatibility issues may occur.
- Complex projects may require a CSS preprocessor (like SCSS/SASS).
- Overriding styles can be difficult in large projects.

Note: CSS is an essential part of modern web development. It provides the tools needed to make web pages visually appealing, consistent, and responsive. With advanced features like Flex box, Grid, animations, and media queries, CSS goes far beyond just changing colors and fonts—it shapes the entire user experience.

Example:
```
<html>
<head>
<title>My Web Page</title>
<style type="text/css">
h1{font-family:mssanserif;font-size:30;font-style:italic;fontweight:
bold;color:red;background-color:blue;border:thin groove}
.m{border-width:thick;border-color:red;border-style:dashed}
.mid{font-family:BankGothicLtBT;text-decoration:link;texttransformation:uppercase;text-
indentation:60%}
</style>
</head>
<body class="m">
<h1> ANUCS</h1>
<p class="mid">Acharya  Nagarjuna  University Guntur</p>
</div>
</body>
</html>
```

Output

2.4 SUMMARY

Cascading Style Sheets (CSS) is a styling language developed by the World Wide Web Consortium (W3C) to enhance the visual presentation of web pages. It allows developers to separate content (HTML) from design, making web development more efficient and maintainable. CSS supports a concept called "cascading," where multiple styles can apply to the same element with a specific order of precedence: inline styles have the highest priority, followed by internal styles, external styles, and finally the browser's default styles. There are three main ways to apply CSS: inline (directly within HTML tags), internal or embedded (within a <style> tag in the <head> section), and external (linked via the <link> tag to a separate CSS file).

CSS uses a variety of selectors to target HTML elements for styling. These include the universal selector (*), element selectors (h1, p), class selectors (.classname), ID selectors (#idname), group selectors (h1, p), and descendant selectors (div p). One of the key concepts in CSS is the box model, which treats each HTML element as a box composed of four parts: content, padding (space between content and border), border (surrounds the padding), and margin (space outside the border). This model provides developers with fine-grained control over the layout, spacing, and design of web elements, making CSS an essential tool for creating visually appealing and user-friendly websites.

2.5 KEY TERMS

CSS (Cascading Style Sheets),W3C (World Wide Web Consortium), Style Separation, Cascading Priority, Inline Style, Internal/Embedded Style, External Style, <style> Tag, <link> Tag, CSS Selectors**,** Box Model, Browser Default Styles.

2.6 Self-Assessment Questions

1. What is CSS? Explain its importance in web development.
2. What are the different ways to apply CSS to an HTML document? Give examples.
3. What is the cascading order of CSS, and how does it affect styling?
4. Differentiate between inline, internal, and external style sheets with examples.
5. What are CSS selectors? Explain different types of selectors with examples.
6. Write a CSS rule to set the font color of all <h1> and <p> tags to blue.
7. What is the box model in CSS? Explain its components with a neat diagram.
8. Write the syntax for linking an external CSS file to an HTML page.
9. Describe the difference between padding and margin in the CSS box model.
10. What happens when multiple CSS rules apply to the same element?

11. How do browser default styles influence the appearance of a web page?
12. Design a simple HTML page using an external CSS file that styles a heading and a paragraph.

2.7 FURTHER READINGS

1. Web Programming, building internet applications, Chris Bates 2nd edition, WILEY Dreamtech
2. Java Programming with JDBC; Donald Bales, O'Reilly
3. Java Network Programming, elliotte Rusty Harold, 3rd Edition, O'Reilly
4. Java Server Pages - Hans Bergsten, SPD O'Reilly
5. Robert W. Sebesta, "Programming the World Wide Web", Third Edition, Pearson Education(2007).
6. Anders Moller and Michael schwartzbach, "An Introduction to XML und Web
7. Jeffrey C. Jackson, "Web Technologies - A Computer Science Perspective", PearsonTechnologies", Addison Wesley (2006) Education (2008).
8. H.M.Deitel, P.J.Deitel. "Java How to Program", Sixth Edition, Pearson Education (2007).

Dr. Kampa Lavanya

LESSON-3
INTRODUCTION TO JAVASCRIPT AND CLIENT-SIDE SCRIPTING

OBJECTIVES:

- Understand the basic syntax rules of JavaScript and how to declare and use variables effectively.
- Learn to define and call functions to organize reusable blocks of code.
- Explore how to handle user interactions through JavaScript events like clicks and form submissions.
- Embed JavaScript into HTML documents using <script> tags in various ways (inline, internal, external).
- Implement basic client-side form validation using JavaScript to ensure correct user input before submission.

STRUCTURE:

3.10 SELF-ASSESSMENT QUESTIONS

3.11 FURTHER READINGS

3.1 JAVASCRIPT

1. Introduction

JavaScript is a high-level, interpreted programming language used primarily for creating interactive and dynamic content on web pages. It allows developers to implement features like image sliders, form validation, dropdown menus, modal windows, and more.

2. History

- **Created by**: Brendan Eich in 1995 at Netscape.
- **Initially called**: Mocha, then LiveScript, and finally renamed JavaScript.
- **Standardized as**: ECMAScript (by ECMA International) in 1997.

3. Features of JavaScript

- JavaScript was designed to add interactivity to HTML pages
- **Lightweight and interpreted**: No need for compilation.
- **Client-side scripting**: Runs in the user's browser.
- **Dynamic typing**: Variables do not require type declaration.
- **Prototype-based OOP**: Supports object-oriented programming.
- **Event-driven**: Reacts to user actions like clicks and input.
- **Cross-platform**: Works on all modern browsers and devices.
- **Asynchronous programming**: Supports callbacks, promises, and async/await.

What Can JavaScript Do?

JavaScript provides HTML designers with a powerful programming tool. While HTML authors may not always be trained programmers, JavaScript is a lightweight scripting language with simple and readable syntax, making it accessible to beginners. With just small code snippets, JavaScript can greatly enhance a webpage's functionality.

3.1.1Key Capabilities of JavaScript

a) **Add Dynamic Content to Web Pages**
   - JavaScript can insert dynamic text into HTML using variables.
   - **Example:**
   - document.write("<h1>" + name + "</h1>");

This writes the value of the variable name directly into the HTML page.

b) **React to User Events**
   - JavaScript can execute specific code in response to events such as:
     - Page loading
     - Mouse clicks
     - Keyboard inputs
   - This interactivity improves the user experience.

c) **Read and Modify HTML Elements**
- JavaScript can access and manipulate the content of HTML elements using the **DOM (Document Object Model)**.

d) It can change text, attributes, styles, and structure of the webpage dynamically.

e) **Validate Form Data**
- JavaScript can check user inputs before the form is submitted to the server.
- This reduces unnecessary server load and improves performance and user feedback.
- Example checks: Empty fields, valid email format, password strength.

f) **Detect Browser Information**
- JavaScript can detect the user's browser and operating system.
- This is useful for customizing content or redirecting users to a browser-compatible version of the site.

g) **Create and Manage Cookies**
- JavaScript can store, retrieve, and delete cookies on the user's computer.
- Cookies are useful for saving user preferences, login sessions, and other personalized data.

4. JavaScript Syntax Basics

```
// Variable declaration
let name = "Alice"; // ES6 syntax
    var age = 25;       // ES5 syntax
const pi = 3.14;    // Constant

// Function
function greet() {
alert("Hello, " + name);
}

// Conditional
if (age > 18) {
console.log("Adult");
} else {
console.log("Minor");
}
```

3.2 VARIABLES IN JAVASCRIPT

What is a Variable?

A **variable** in JavaScript is a named container used to store data that can be referenced and manipulated in a program. Variables allow developers to reuse values and manage dynamic data.

Declaring Variables

JavaScript provides three keywords to declare variables:

Keyword	Scope	Reassignment	Hoisting	Block Scoped
var	Function	Yes	Yes	No
let	Block	Yes	No	Yes
const	Block	No	No	Yes

Syntax

```
var x = 10;
let name = "Alice";
const PI = 3.14;
```

- var is function-scoped and can be redeclared.
- let is block-scoped and cannot be redeclared in the same scope.
- const is block-scoped and must be initialized during declaration. Its value cannot be changed (immutable binding).

Variable Naming Rules

- Names must start with a letter, underscore _, or dollar sign $.
- Cannot start with a digit.
- Are case-sensitive (Name and name are different).
- Should not use reserved JavaScript keywords (like for, if, etc.)

Valid Names:

```
letuserName;
let $price;
let _totalAmount;
```

Invalid Names:

```
let 2value;     // Invalid: starts with a number
let if;         // Invalid: reserved keyword
```

Data Types Stored in Variables

Variables can hold different types of data:

```
let age = 25;              // Number
let name = "John";         // String
letisActive = true;        // Boolean
let person = {name:"Ava"}; // Object
let items = [1, 2, 3];     // Array
let result = null;         // Null
let score;                 // Undefined
```

Variable Scope

1. **Global Scope** – Declared outside any function or block.
2. **Function Scope** – Declared inside a function using var.
3. **Block Scope** – Declared using let or const inside {} like loops, if statements.

```
functiontestScope() {
var a = 10; // function scoped
let b = 20; // block scoped
if (true) {
let c = 30;
   console.log(c); // 30
  }
  // console.log(c); // Error: c is not defined
}
```

Variable Hosting

- var declarations are hoisted to the top of their scope but not initialized.
- let and const are hoisted but stay in the temporal dead zone (TDZ) until declared.

```
console.log(x); // undefined (due to hoisting)
var x = 5;
```

```
// console.log(y); // Error: Cannot access 'y' before initialization
let y = 10;
```

Best Practices

- Always prefer let or const over var.
- Use const when the value should not change.
- Declare variables at the top of their scope.
- Use meaningful names (e.g., userAge instead of x).

Example Program

```
<!DOCTYPE html>
<html>
        <body>
<h2>JavaScript Variable Example</h2>
<p id="demo"></p>
<script>
        constfirstName = "John";
        let age = 30;
        var message = firstName + " is " + age + " years old.";
        document.getElementById("demo").innerHTML = message;
</script>

        </body>
</html>
```

Output:

John is 30 years old.

3.3 FUNCTIONS

What is a Function?

A **function** is a block of reusable code designed to perform a specific task. Functions help organize code, reduce repetition, and improve maintainability.

Why Use Functions?

- To reuse code.
- To break down complex problems into smaller, manageable parts.
- To improve readability and maintainability.
- To execute code only when invokedorcalled.

Function Declaration (Function Statement)

```
function greet() {
console.log("Hello, World!");
}
greet(); // Output: Hello, World!
```

Function Parameters and Arguments

Functions can accept inputs called **parameters**, and these values passed are called **arguments**.

```
functiongreetUser(name) {
console.log("Hello, " + name + "!");
}
greetUser("Alice"); // Output: Hello, Alice!
```

Return Statement

Functions can return a value using the return keyword.

```
function add(a, b) {
return a + b;
}
let result = add(5, 3); // result is 8
```

Function Expressions

A function can also be defined as an **expression** and stored in a variable.

```
const multiply = function(x, y) {
return x * y;
};
console.log(multiply(4, 5)); // Output: 20
```

Arrow Functions (ES6+)

Arrow functions provide a shorter syntax for writing functions.

```
const square = (n) => {
return n * n;
};
console.log(square(6)); // Output: 36
```

Simplified Version (one-liner):

```
const square = n => n * n;
```

Anonymous Functions

Functions without a name are called anonymous functions. Often used in event handlers or passed as arguments.

```
setTimeout(function() {
console.log("Executed after 2 seconds");
}, 2000);
```

Immediately Invoked Function Expression (IIFE)

A function that runs immediately after it is defined.

```
(function() {
console.log("IIFE executed!");
})();
```

Nested Functions

Functions can be defined inside other functions and have access to variables in the parent function.

```
function outer() {
letouterVar = "I'm outer!";

function inner() {
console.log(outerVar); // Can access outer variable
  }

inner();
}
outer();
```

Function Scope

- Variables declared inside a function are local to that function.
- Functions have access to global variables and to variables in their parent functions (closures).

Rest Parameters

Allows a function to accept an indefinite number of arguments as an array.

```
function sum(...numbers) {
```

```
returnnumbers.reduce((total, num) => total + num);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

Default Parameters

Set default values for parameters.

```
function greet(name = "Guest") {
console.log("Hello, " + name);
}
greet();       // Output: Hello, Guest
greet("Ravi"); // Output: Hello, Ravi
```

Example: Function in HTML

```
<!DOCTYPE html>
<html>
        <body>
<h2>JavaScript Function Example</h2>
<p>Click the button to see a message.</p>
<button onclick="showMessage()">Click Me</button>
        <script>
functionshowMessage() {
alert("Hello! You clicked the button.");
  }
        </script>
        </body>
</html>
```

Best Practices

- Use descriptive function names.
- Keep functions small and focused on a single task.
- Avoid global variables inside functions.
- Use arrow functions for shorter syntax when appropriate.
- Use const or let when assigning functions to variables.

Common Use Cases

- Input validation
- Performing calculations
- Event handling (e.g., button clicks)
- API calls
- Data manipulation

3.4 EVENTS IN JAVASCRIPT

3.4.1. Introduction to Events in JavaScript

Events in JavaScript are actions or occurrences that happen in the browser, which JavaScript can respond to. Examples of events include:

- A user clicking a button

- A web page loading
- A form being submitted
- A key being pressed

JavaScript allows developers to create dynamic and interactive web applications by reacting to these events.

3.4.2. Common Types of JavaScript Events

Event	Description
click	Triggered when an element is clicked
dblclick	Triggered when an element is double-clicked
mouseover	Triggered when the mouse pointer moves over an element
mouseout	Triggered when the mouse pointer moves out of an element
keydown	Triggered when a key is pressed
keyup	Triggered when a key is released
load	Triggered when a page or image is fully loaded
submit	Triggered when a form is submitted
change	Triggered when the value of a form element changes
focus	Triggered when an element gains focus
blur	Triggered when an element loses focus

3.4.3. Adding Event Handlers in JavaScript

JavaScript provides multiple ways to attach event handlers to elements:
a) Inline HTML Event Handling
<button onclick="alert('Button clicked!')">Click Me</button>
b) Using DOM Properties
letbtn = document.getElementById("myBtn");
btn.onclick = function () {
alert("Button clicked!");
};
c) Using addEventListener()
letbtn = document.getElementById("myBtn");

```
btn.addEventListener("click", function () {
alert("Button clicked using addEventListener!");
});
```

addEventListener() is the preferred method as it allows attaching multiple handlers and supports event bubbling and capturing.

3.4.4. Event Object

When an event occurs, an event object is automatically passed to the event handler. It contains useful information like:

- type – the type of the event
- target – the element on which the event occurred
- preventDefault() – prevents the default action
- stopPropagation() – stops the event from bubbling up

```
document.getElementById("myForm").addEventListener("submit", function (event) {
event.preventDefault(); // Prevents form from submitting
alert("Form submission prevented!");
});
```

3.4.5. Event Propagation

JavaScript events follow a three-phase propagation model:

1. **Capturing Phase** (trickle down)
2. **Target Phase** (event reaches the target)
3. **Bubbling Phase** (bubble up to ancestors)

```
element.addEventListener("click", handler, true); // Capturing
element.addEventListener("click", handler, false); // Bubbling (default)
```

3.4.6. Event Delegation

Instead of adding event listeners to individual elements, event delegation allows attaching a single event listener to a parent element that handles events for its child elements.

```
document.getElementById("parent").addEventListener("click", function (e) {
if (e.target&&e.target.matches("button.classname")) {
alert("Button inside parent clicked");
  }
});
```

3.4.7. Removing Event Listeners

You can remove an event listener using removeEventListener().

```
function greet() {
alert("Hello");
}
btn.addEventListener("click", greet);

// To remove
btn.removeEventListener("click", greet);
```

Note: The function reference must be the same to remove it.

3.4.8. Keyboard and Mouse Events

Keyboard Events: keydown, keypress, keyup

```
document.addEventListener("keydown", function (e) {
console.log("Key pressed:", e.key);
});
```

Mouse Events: click, dblclick, mousedown, mouseup, mousemove, mouseenter, mouseleave

```
document.addEventListener("mousemove", function (e) {
console.log("Mouse X:", e.clientX, "Mouse Y:", e.clientY);
});
```

3.4.9. Form Events

Form controls support events like:

- submit
- change
- focus
- blur

```
document.getElementById("name").addEventListener("change", function () {
alert("Name changed!");
});
```

3.4.10. Best Practices for Using Events

- Prefer addEventListener() over inline handlers
- Use event delegation for dynamic or multiple elements
- Always remove unused event listeners to avoid memory leaks
- Avoid anonymous functions if the event needs to be removed
- Use preventDefault() and stopPropagation() wisely

3.4.11. Example: Simple Event Handler

```
<!DOCTYPE html>
<html>
<head>
<title>Event Example</title>
</head>
<body>
<button id="btn">Click Me</button>

<script>
document.getElementById("btn").addEventListener("click", function () {
alert("Button was clicked!");
   });
</script>
</body>
</html>
```

Conclusion

Events in JavaScript are crucial for creating interactive and dynamic web applications. Mastery of events and their proper handling ensures a responsive and user-friendly experience. Understanding concepts like event propagation, delegation, and proper use of listeners is essential for efficient JavaScript programming.

3.5 EMBEDDING JAVASCRIPT IN HTML

JavaScript can be embedded in an HTML document to add dynamic functionality to web pages. This is done using the <script> tag. JavaScript can be embedded in several ways: directly within the HTML file (inline or internal), or through an external JavaScript file.

3.5.1. Using the <script> Tag

The <script> tag is used to embed JavaScript code in HTML. It can be placed in the <head> or <body> section of an HTML document.

```
<script>
  // JavaScript code goes here
</script>
```

3.5.2. Types of Embedding JavaScript

There are **three main methods** of embedding JavaScript into HTML:

a) Inline JavaScript

JavaScript code can be placed directly within an HTML element's attribute, such as the onclick attribute of a button.

Example:

```
<button onclick="alert('Hello, World!')">Click Me</button>
```

Use Case: Best for simple tasks and demonstrations. Not recommended for complex logic or production code.

b) Internal JavaScript (Embedded in HTML)

You can write JavaScript within a <script> tag inside the HTML file, typically in the <head> or at the end of the <body>.

Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Internal JavaScript</title>
<script>
functiongreetUser() {
alert("Welcome to my website!");
   }
</script>
</head>
<body>
<button onclick="greetUser()">Say Hello</button>
</body>
</html>
```

c) External JavaScript

JavaScript can be written in a separate .js file and linked to the HTML document using the src attribute of the <script> tag.

Example:

1. HTML File (index.html)

```
<!DOCTYPE html>
<html>
        <head>
                <title>External JavaScript</title>
                <script src="script.js"></script>
        </head>
        <body>
                <button onclick="greetUser()">Click Me</button>
        </body>
</html>
```

2. JavaScript File (script.js)

```
functiongreetUser() {
alert("Hello from external JavaScript!");
}
```

Benefits:

- Cleaner HTML code
- Reusability of scripts across pages
- Easier maintenance and debugging

Script Placement in HTML

Placement	Description
<head>	Executes before the content loads. May delay rendering.
End of <body>	Preferred for performance. Loads content first, then script.
defer	Attribute used to defer script execution until after HTML parsing.
async	Attribute used to download and execute script asynchronously.

Example with defer:

```
<script src="script.js" defer></script>
```

Security Note

Avoid embedding user-generated content directly into <script> tags to prevent Cross-Site Scripting (XSS) attacks.

Table

Method	Where Used	Use Case	Example
Inline	HTML attributes	Quick tests, simple interactions	<button onclick="alert('Hi')">
Internal	<script> tag	Page-specific logic	<script>function(){...}</script>
External	Linked .js file	Reusable and maintainable code	<script src="script.js"></script>

Best Practices

- Always place scripts just before </body> unless required in <head>.
- Use external JavaScript for modularity and reuse.
- Avoid inline JavaScript in production.
- Use defer or async for performance optimization.

3.6 FORM VALIDATION BASICS

Form validation is the process of checking the data entered into a form to ensure it meets certain rules before it is submitted to a server. It helps in maintaining data quality, enhancing user experience, and improving security.

Types of Form Validation

1. **Client-Side Validation**
   - Happens in the user's browser (before data is sent to the server).
   - Uses HTML5 attributesorJavaScript**.**
   - Provides immediate feedback.
2. **Server-Side Validation**
   - Happens on the server after the form is submitted.
   - Essential for security (users can bypass client-side validation).
   - Typically done using languages like PHP, Python, Java, etc.

Common Validation Rules

- **Required fields**: Ensures the user has entered something.
- **Data type checks**: Ensures values are numbers, emails, URLs, etc.
- **Length checks**: Limits input to a specific number of characters.
- **Pattern matching**: Validates format using regular expressions (e.g., phone number, postal code).
- **Matching fields**: Confirms that values in two fields are the same (e.g., password and confirm password).

Client-Side Validation Techniques

1. HTML5 Attributes

- required: Field must be filled.
- type="email": Must be a valid email format.
- min, max, maxlength: Numeric or text length limits.
- pattern: Regular expression validation.

```
<form>
<input type="text" name="name" required>
<input type="email" name="email" required>
<input type="password" name="pwd" minlength="6" required>
<input type="submit">
</form>
```

Example

```
<!DOCTYPE html>
<html>
<head>
<script>
functionvalidateForm() {
let x = document.forms["myForm"]["fname"].value;
if (x == "") {
```

```
alert("Name must be filled out");
return false;
  }
}
</script>
</head>
<body>
<h2>JavaScript Validation</h2>
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Output:

Common Use Cases

- Form validation
- Image sliders
- Interactive maps
- Popups/modals
- Real-time updates (chat, notifications)
- Games
- AJAX requests (load data without refreshing page)

Advantages

- Enhances user experience
- Reduces server load
- Fast client-side processing
- Wide community and support

Limitations

- Runs in the browser – can't access server files directly
- Code visibility (can be viewed and modified by users)
- May behave differently across browsers if not standardized

Modern JavaScript Tools and Frameworks

- **Libraries**: jQuery, Axios
- **Frameworks**: React, Angular, Vue.js
- **Runtimes**: Node.js (for server-side JS)
- **Build tools**: Webpack, Babel, ESLint

Note:

JavaScript is an essential language for web development. It bridges the gap between static HTML/CSS and dynamic user interaction, making websites more functional, responsive, and user-friendly.

Example: Change Text on Button Click

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Example</title>
</head>
<body>
<p id="myParagraph">This is the initial text.</p>
<button onclick="changeText()">Click Me</button>
<script>
functionchangeText() {
document.getElementById("myParagraph").innerHTML = "This text has been changed!";
   }
</script>
</body>
</html>
```

Explanation of Code

1. **HTML Structure**:
   - A paragraph element is defined with the ID myParagraph.
   - A button element is provided to trigger the JavaScript function.
2. **JavaScript Functionality**:
   - The changeText() function is defined inside a <script> tag.
   - It uses document.getElementById("myParagraph") to access the paragraph.
   - The innerHTML property is used to change the paragraph's content.
3. **Event Handling**:
   - The onclick attribute of the button is used to call the changeText() function when the button is clicked.
4. **Output**:
   - Initially, the paragraph displays: **"This is the initial text."**
   - After clicking the button, the paragraph updates to: **"This text has been changed!"**

Key JavaScript Concepts Demonstrated

- **HTML Structure**: Defines the elements to be manipulated.
- **JavaScript Behavior**: Provides interactivity and control logic.
- **Event Handling**: The onclick event responds to user actions.
- **DOM Manipulation**: document.getElementById() and innerHTML are used to modify webpage content.
- **Functions**: Encapsulate reusable code logic that executes when triggered.

3.7DATA TYPES

- **Primitive**: String, Number, Boolean, Null, Undefined, Symbol, BigInt.
- **Non-Primitive**: Object, Array, Function.

Strings - are a series of letters and numbers enclosed in quotation marks. JavaScript uses the string literally; it doesn't process it. You'll use strings for text you want displayed or values you want passed along.

Numbers - are values that can be processed and calculated. You don't enclose them in quotation marks. The numbers can be either positive or negative.

Boolean (true/false) - lets you evaluate whether a condition meets or does not meet specified criteria.

Null - is an empty value. null is not the same as 0 -- 0 is a real, calculable number, whereas null is the absence of any value.

Data Types

TYPE	EXAMPLE
Numbers	Any number, such as 17, 21, or 54e7
Strings	"Greetings!" or "Fun"
Boolean	Either true or false
Null	A special keyword for exactly that – the null value (that is, nothing)

3.8SUMMARY

JavaScript is a high-level, interpreted scripting language designed to make web pages interactive and dynamic. Developed by Brendan Eich in 1995 and standardized as ECMAScript in 1997, it supports features such as dynamic typing, prototype-based object-oriented programming, event-driven design, and asynchronous operations. With JavaScript, developers can manipulate HTML elements, validate form data, detect browsers, and manage cookies, making it a key component of front-end web development.

Variables in JavaScript can be declared using var, let, or const, each offering different scope and behavior. The language supports multiple data types and enforces specific naming rules for identifiers. Functions—whether defined through declarations, expressions, or arrow syntax—enable code reuse and modularity. Events are central to JavaScript, allowing responses to user interactions like mouse clicks, keypresses, or page loads. Event handling can be implemented inline, through DOM properties, or using the addEventListener**()** method, providing flexibility in interaction management.

JavaScript can be embedded into HTML using the <script> tag, either inline, internally within the HTML document, or externally via linked script files. External scripts encourage cleaner, modular code. Additionally, the defer and async attributes optimize how scripts load and execute, improving performance. Through its event-driven model, versatile syntax, and seamless integration with HTML and CSS, JavaScript remains an essential tool for creating modern, responsive, and interactive web applications.

3.9KEY TERMS

JavaScript,ECMAScript,Variable (var, let, const),DataTypes,Function,Arrow Function,EventHandling,DOM (Document Object Model),FormValidation,addEventListener(),EventDelegation,EventPropagation,Script Tag (**<script>**),Inline/Internal/External JavaScript,Hoisting.

3.10 Self-Assessment Questions

1. What is JavaScript and where is it primarily used?
2. What is the difference between var, let, and const in JavaScript?
3. List some of the basic data types in JavaScript.
4. What is the purpose of functions in JavaScript?
5. How do arrow functions differ from regular functions in JavaScript?
6. What is the DOM in JavaScript?
7. How can JavaScript be added to an HTML document?
8. What is form validation and why is it important in JavaScript?
9. Explain the use of addEventListener() in event handling.
10. What is the difference between inline, internal, and external JavaScript?

3.11Further Readings

1. JavaScript: The Definitive Guide, Seventh Editionby David Flanagan. O'Reilly Media.
2. Eloquent JavaScript: A Modern Introduction to Programming, Third EditionbyMarijnHaverbeke. No Starch Press.
3. Beginning JavaScript, Fifth EditionbyJeremy McPeak. Wrox (Wiley).
4. Learning JavaScript Design Patterns, First EditionbyAddyOsmani. O'Reilly Media.
5. The complete Reference Java 2 Fifth Edition by Patrick Naughton and Herbert Schildt. TMH

Dr. Kampa Lavanya

LESSON-4
OBJECTS IN JAVA SCRIPTAND DYNAMIC HTML (DHTML)

AIM AND OBJECTIVES:

- Understand the concepts and structure of JavaScript objects, including their properties and methods.
- Learn how to create, access, modify, and delete object properties using different approaches.
- Explore how to use JavaScript to dynamically change HTML content, attributes, and styles.
- Apply the `this` keyword and object methods to build real-world functionalities.
- Demonstrate the use of JavaScript to manipulate DOM elements for creating interactive web pages.

STRUCTURE:

4.1 OBJECTS IN JAVASCRIPT

In JavaScript, objects are collections of related data and functionality. These can include built-in objects provided by the browser (like window, document, etc.) or user-defined objects. Objects contain properties (values) and methods (functions).

1. Window Object

The windowobject is the top-level object in the browser's JavaScript environment. It represents the browser window or tab that displays the HTML document. Every global

variable, function, or object created in JavaScript is automatically a member of the window object.

Key Features:

- It is automatically created by the browser when a web page loads.
- Acts as the global object in browsers.
- Contains properties like location, history, navigator, etc.
- Provides methods like alert(), prompt(), confirm(), setTimeout(), and setInterval().

Note:

- The window object is global, so you usually don't need to type window.explicitly.
- alert("Hello!"); // Equivalent to window.alert("Hello!");
- console.log(window.innerHeight); // Displays window height

Common Properties:

Property	Description
window.innerWidth	Width of the content area of the window
window.innerHeight	Height of the content area
window.location	Provides URL and navigation functions
window.document	Refers to the current page's document
window.navigator	Gives browser information
window.history	Controls the session history

Common Methods:

Method	Description
alert(message)	Displays a popup alert box
confirm(message)	Displays a popup with OK and Cancel
prompt(message)	Displays a popup with a text input field
setTimeout(fn, time)	Executes a function after a delay

Method	Description
setInterval(fn, time)	Repeats function execution at intervals
open(url)	Opens a new browser window
close()	Closes the current window

Example Usage:

```
// Display an alert box
window.alert("Welcome to JavaScript!");
// Set a timeout
window.setTimeout(function() {
alert("This alert appears after 3 seconds.");
}, 3000);
// Log the window dimensions
console.log("Width: " + window.innerWidth);
console.log("Height: " + window.innerHeight);
// Redirect to another URL
window.location.href = "https://www.example.com";
```

2. Navigator Object

The **Navigator Object** is a built-in object in JavaScript that contains information about the web browser being used by the client (user). It is a property of the window object, and it provides useful metadata about the browser's name, version, platform, and capabilities.

The navigator object is often used for browser detection and for controlling or accessing certain browser features.

Syntax:

```
window.navigator
// or simply
navigator
```

Key Properties of Navigator Object:

Property	Description
appName	Returns the name of the browser (mostly returns "Netscape" for compatibility reasons)

Property	Description
appCodeName	Returns the code name of the browser (usually "Mozilla")
appVersion	Returns the version information of the browser
userAgent	Returns the user-agent header sent by the browser to the server
platform	Returns the platform on which the browser is running (e.g., "Win32", "Linux x86_64")
language or languages	Returns the language preference of the browser
cookieEnabled	Returns true if cookies are enabled in the browser
onLine	Returns true if the browser is online
javaEnabled()	Returns true if Java is enabled in the browser
plugins	Returns a list of all plugins installed in the browser
mimeTypes	Returns a list of all MIME types supported by the browser
hardwareConcurrency	Returns the number of logical processors available to run threads
maxTouchPoints	Returns the maximum number of simultaneous touch points supported
webdriver	Returns true if the browser is controlled by automation (e.g., Selenium)

Example: Using Navigator Object

```
<!DOCTYPE html>
<html>
<head>
<title>Navigator Object Example</title>
</head>
<body>
<h2>Browser Information</h2>
<script>
document.write("<b>Browser Name:</b> " + navigator.appName + "<br>");
document.write("<b>Browser Code Name:</b> " + navigator.appCodeName + "<br>");
document.write("<b>Browser Version:</b> " + navigator.appVersion + "<br>");
```

```
document.write("<b>Platform:</b> " + navigator.platform + "<br>");
document.write("<b>User Agent:</b> " + navigator.userAgent + "<br>");
document.write("<b>Cookies Enabled:</b> " + navigator.cookieEnabled + "<br>");
document.write("<b>Online Status:</b> " + navigator.onLine + "<br>");
</script>
</body>
</html>
```

Common Uses of Navigator Object

1. **Browser Detection:**
   To perform actions depending on which browser is being used (though feature detection is preferred over browser detection).
2. **Language Preferences:**
   To customize content based on the user's preferred language.
3. **Online/Offline Detection:**
   To check whether the user is connected to the internet.
4. **Feature Availability:**
   Determine if Java, cookies, or other plugins are enabled or available.

Note:

Avoid relying solely on browser detection using the navigator object for functionality. Instead, use feature detection (via libraries like Modernizer) to ensure consistent behavior across different environments.

Table

Feature	Description
Object Name	navigator
Purpose	Provides information about the browser and system
Important Properties	userAgent, platform, language, cookieEnabled, onLine
Common Use	Browser info, feature support, user customization

Key Properties:

- navigator.appName: Browser name
- navigator.appVersion: Browser version
- navigator.platform: OS platform
- navigator.language: Language setting
- navigator.onLine: Checks if the browser is online

Example:

```
console.log("Browser Name: " + navigator.appName);

console.log("Online: " + navigator.onLine);
```

3. Document Object

The documentobject is a part of the Browser Object Model (BOM**)** and more specifically, it is the core of the DOM (Document Object Model)**.** It represents the webpage loaded in the browser and acts as an entry point to access and manipulate HTML content, structure, and styles through JavaScript.

Syntax:

```
window.document
// or simply
document
```

Purpose of Document Object

- To access elements in the HTML document.
- To modify the structure, style, and content of the webpage.
- To handle events**.**
- To dynamically create or delete HTML elements.

Important Properties of Document Object

Property	Description
document.title	Gets or sets the title of the document
document.URL	Returns the complete URL of the document
document.domain	Gets the domain name of the server
document.body	Represents the <body> element of the document
document.head	Represents the <head> element of the document
document.forms	Returns a collection of all <form> elements
document.images	Returns a collection of all <img> elements
document.links	Returns a collection of all <a> elements with an href

Property	Description
document.cookie	Gets or sets cookies for the current page
document.readyState	Returns the loading state of the document (e.g., "loading", "complete")
document.documentElement	Returns the root <html> element

Commonly Used Methods of Document Object

Method	Description
getElementById(id)	Returns the element with the specified ID
getElementsByTagName(tag)	Returns a collection of elements with the given tag name
getElementsByClassName(class)	Returns a collection of elements with the specified class name
querySelector(selector)	Returns the first element that matches the CSS selector
querySelectorAll(selector)	Returns all elements that match the CSS selector
createElement(tag)	Creates a new HTML element
createTextNode(text)	Creates a new text node
appendChild(node)	Adds a node to the end of a list of children of a specified parent
removeChild(node)	Removes a child node from the document
write(text)	Writes HTML expressions or JavaScript code to the document
open()	Opens a document for writing
close()	Closes a document stream opened with document.open()

Example: Access and Modify Elements

```
<!DOCTYPE html>
<html>
<head>
<title>Document Object Example</title>
</head>
<body>
<h1 id="heading">Hello World</h1>
<p class="message">This is a paragraph.</p>
<script>
      // Access and modify content using document object
document.getElementById("heading").innerHTML = "Welcome to JavaScript!";
document.querySelector(".message").style.color = "blue";
alert("Page Title: " + document.title);
</script>
</body>
</html>
```

Use Cases of the Document Object

- Reading or changing the page content dynamically.
- Adding or removing HTML elements using JavaScript.
- Validating form input.
- Responding to user interactions like clicks and key presses.
- Manipulating CSS styles and classes.

Important Notes

- Changes made using the document object reflect immediately on the webpage.
- document.write() should be avoided after the page has loaded as it can overwrite the entire document.

Table

Feature	Description
Object Name	document
Part of	DOM (Document Object Model)
Purpose	Interact with and modify the HTML structure
Common Methods	getElementById(), querySelector(), createElement(), write()

Feature	Description
Common Properties	title, URL, body, head, forms, images
Usage	Dynamic HTML manipulation, event handling, content modification

Common Methods:

- getElementById()
- getElementsByTagName()
- querySelector()
- createElement(), appendChild()

Example:

```
document.getElementById("demo").innerHTML = "Text changed!";
```

4. Form Object

Form Object in JavaScript

The Formobject in JavaScript represents an HTML <form> element. It allows JavaScript to access**,** manipulate**,** validate, and submit form data dynamically.

Each form in an HTML document becomes a part of the document.forms collection. You can access a form either by its index or name.

Syntax:

```
// Access form by index
document.forms[0]
// Access form by name (name attribute in HTML)
document.forms["formName"]
```

Purpose of the Form Object

- To retrieve form input values.
- To set or modify form fields.
- To validate form inputs before submission.
- To handle form submission via JavaScript.

Important Properties of the Form Object

Property	Description
elements	Collection of all form elements (inputs, selects, buttons, etc.)
length	Number of elements in the form
name	The name of the form (from the name attribute)
action	URL where the form data is sent (from the action attribute)
method	HTTP method used when submitting the form (GET or POST)
target	Specifies where to display the response (like _blank, _self)
enctype	Encoding type for submitted form data (e.g., application/x-www-form-urlencoded)
acceptCharset	Character encodings the server can handle
autocomplete	Indicates whether form input fields can be auto-completed (on or off)
noValidate	If present, form will not be validated on submit

Common Methods of the Form Object

Method	Description
submit()	Submits the form programmatically
reset()	Resets all form fields to their default values
checkValidity()	Returns true if the form is valid
reportValidity()	Reports validity and displays error messages if invalid

Example: Accessing and Using Form Object

```
<!DOCTYPE html>
<html>
<head>
<title>Form Object Example</title>
</head>
```

```
<body>
<form name="myForm" action="/submit" method="post">
   Name: <input type="text" name="username"><br>
   Email: <input type="email" name="email"><br>
<input type="button" value="Submit" onclick="submitForm()">
</form>
<script>
functionsubmitForm() {
var form = document.forms["myForm"];
var name = form["username"].value;
var email = form["email"].value;
if (name === "" || email === "") {
alert("All fields are required!");
      } else {
alert("Form submitted with:\nName: " + name + "\nEmail: " + email);
form.submit(); // optional: submit programmatically
      }
   }
</script>
</body>
</html>
```

Form Validation Example

```
functionvalidateForm() {
var form = document.forms["myForm"];
if (!form.checkValidity()) {
alert("Form is invalid!");
   } else {
alert("Form is valid and ready to submit.");
   }
}
```

Accessing Form Elements

You can access individual form fields using:

```
document.forms["myForm"]["username"].value;
```

Or loop through all form elements:

```
var form = document.forms[0];
for (vari = 0; i<form.length; i++) {
console.log(form.elements[i].name + ": " + form.elements[i].value);
```

}

Table

Feature	Details
Object	Form
Part of	DOM (document.forms)
Used for	Accessing and manipulating form fields
Properties	elements, action, method, target, length, name
Methods	submit(), reset(), checkValidity(), reportValidity()
Use Cases	Input validation, programmatic form submission, dynamic data handling

5. Date Object

Date Object in JavaScript

The Dateobject is a built-in JavaScript object used to work with dates and times. It allows you to create, retrieve, and manipulate date and time values such as the current date, day, month, year, hour, minute, second, and millisecond.

Syntax:

```
let date = new Date();           // Current date and time
let date = new Date(milliseconds); // Date based on milliseconds since Jan 1, 1970
let date = new Date(dateString);   // Date from a string (e.g., "2025-06-11")
let date = new Date(year, month, day, hours, minutes, seconds, ms);
```

Note: In JavaScript, months are zero-indexed (January = 0, December = 11).

Creating Date Objects

1. Current Date and Time

```
let now = new Date();
```

2. From String

let d = new Date("2025-06-11");

3. From Components

let d = new Date(2025, 5, 11, 10, 30, 0); // June 11, 2025 10:30:00

4. From Milliseconds

let d = new Date(0); // January 1, 1970 (Unix Epoch Time)

Important Date Methods

Getter Methods (to retrieve parts of the date)

Method	Description
getFullYear()	Returns the 4-digit year (e.g., 2025)
getMonth()	Returns the month (0–11)
getDate()	Returns the day of the month (1–31)
getDay()	Returns the day of the week (0–6, where 0 = Sunday)
getHours()	Returns the hour (0–23)
getMinutes()	Returns the minutes (0–59)
getSeconds()	Returns the seconds (0–59)
getMilliseconds()	Returns the milliseconds (0–999)
getTime()	Returns milliseconds since Jan 1, 1970
getTimezoneOffset()	Returns difference from UTC in minutes

Setter Methods (to set parts of the date)

Method	Description
setFullYear(year)	Sets the full year

Method	Description
setMonth(month)	Sets the month (0–11)
setDate(day)	Sets the day of the month
setHours(hour)	Sets the hour
setMinutes(min)	Sets the minutes
setSeconds(sec)	Sets the seconds
setMilliseconds(ms)	Sets the milliseconds
setTime(ms)	Sets the date based on milliseconds since 1970

Conversion Methods

Method	Description
toDateString()	Returns date as a readable string (e.g., "Wed Jun 11 2025")
toTimeString()	Returns time as a readable string (e.g., "10:30:00 GMT+0530")
toISOString()	Returns ISO format string (e.g., "2025-06-11T05:00:00.000Z")
toLocaleDateString()	Returns date as a localized string
toLocaleTimeString()	Returns time as a localized string
toUTCString()	Returns UTC date string

Examples

Example 1: Get Current Date and Time

```
let now = new Date();
console.log(now.toString());
```

Example 2: Extract Date Components

```
let today = new Date();
console.log("Year: " + today.getFullYear());
console.log("Month: " + today.getMonth()); // 0 = January
```

```
console.log("Date: " + today.getDate());
console.log("Day: " + today.getDay());     // 0 = Sunday
```

Example 3: Set Custom Date

```
letcustomDate = new Date();
customDate.setFullYear(2026);
customDate.setMonth(11); // December
customDate.setDate(25);
console.log(customDate.toDateString());  // "Fri Dec 25 2026"
```

Example 4: Compare Two Dates

```
let d1 = new Date("2025-06-01");
let d2 = new Date("2025-06-11");
if (d1 < d2) {
console.log("d1 is earlier than d2");
}
```

Use Cases of Date Object

- Displaying current date and time
- Calculating date differences
- Creating countdowns or clocks
- Validating date input in forms
- Generating timestamps for logs

Important Notes

- Months are 0-indexed: January is 0, December is 11.
- Always consider time zone differences when working with global users.
- Prefer using toISOString() or toLocaleString**()** for consistent formatting.

Table

Feature	Description
Object	Date
Purpose	Work with dates and times
Common Getters	getFullYear(), getMonth(), getDate(), getHours()
Common Setters	setFullYear(), setMonth(), setDate()

Feature	Description
Formats	toString(), toISOString(), toLocaleString()
Use Cases	Clocks, timestamps, reminders, form validations

6. String Object

The Stringobject in JavaScript is a built-in object that allows you to create, manipulate, andwork with text (string values)**.** Strings are sequences of characters used for storing and manipulating text.

What is a String?

A **string** is a sequence of characters enclosed in:

- single quotes (')
- double quotes ("")
- backticks (``) — for template literals

```
let str1 = 'Hello';
let str2 = "World";
let str3 = `Hello, ${str2}`; // Template literal
```

Creating String Objects

1. String Literal

```
let name = "JavaScript";
```

2. String Object Using Constructor

```
letnameObj = new String("JavaScript");
```

Note: Using string objects (new String(...)) is not recommended for regular usage. Use string literals for simplicity and performance.

Common String Properties

Property	Description
length	Returns the number of characters in the string

```
let text = "Hello World";
console.log(text.length);  // Output: 11
```

Common String Methods

String Inspection Methods

Method	Description
charAt(index)	Returns the character at a specified index
charCodeAt(index)	Returns the Unicode of the character
includes(substring)	Checks if the string contains a substring
startsWith(substring)	Checks if the string starts with a substring
endsWith(substring)	Checks if the string ends with a substring
indexOf(substring)	Returns the index of the first occurrence
lastIndexOf(substring)	Returns the last index of the substring

String Manipulation Methods

Method	Description
concat(string2)	Combines two or more strings
replace(search, replace)	Replaces substring with another
replaceAll(search, replace)	Replaces all occurrences
slice(start, end)	Extracts part of a string
substring(start, end)	Similar to slice but does not accept negative indices
substr(start, length)	Deprecated; use slice instead
toLowerCase()	Converts string to lowercase
toUpperCase()	Converts string to uppercase

Method	Description
trim()	Removes whitespace from both ends
trimStart() / trimEnd()	Trims start or end spaces

String Splitting and Matching

Method	Description
split(separator)	Splits a string into an array
match(regex)	Matches string against a regular expression
matchAll(regex)	Returns all matches as an iterator
search(regex)	Searches for a match using a regular expression
includes(text)	Checks if text exists in the string

String Conversion Methods

Method	Description
toString()	Returns string representation
valueOf()	Returns the primitive string value

Examples

Example 1: Basic Usage

```
letmsg = "Hello JavaScript";
console.log(msg.length);        // 17
console.log(msg.charAt(0));     // "H"
console.log(msg.toUpperCase()); // "HELLO JAVASCRIPT"
```

Example 2: Searching and Replacing

```
let txt = "The rain in Spain";
console.log(txt.includes("rain"));        // true
```

```
console.log(txt.indexOf("rain"));        // 4
console.log(txt.replace("rain", "sun"));  // "The sun in Spain"
```

Example 3: Extracting Parts of a String

```
letstr = "Hello, world!";
console.log(str.slice(0, 5));    // "Hello"
console.log(str.substring(7));  // "world!"
```

Example 4: Using Template Literals

```
let name = "Alice";
let greeting = `Hello, ${name}!`;
console.log(greeting); // "Hello, Alice!"
```

Example 5: Splitting a String

```
letcsv = "apple,banana,orange";
let fruits = csv.split(",");
console.log(fruits); // ["apple", "banana", "orange"]
```

Use Cases of String Object

- Displaying and formatting messages
- Validating form inputs
- Searching and filtering text
- Parsing and formatting data
- Creating dynamic content (e.g., templates)

Important Notes

- Strings are immutable: operations like replace() or concat() return a new string, not modify the original.
- Prefer using string literals instead of new String().
- For multi-line strings or embedded variables, use template literals (`) with ${}.

Table

Feature	Description
Object	String
Type	Wrapper object for text

Feature	Description
Common Property	Length
Common Methods	charAt(), slice(), toUpperCase(), replace(), split()
Use Cases	Text display, manipulation, validation, and search
Mutability	Immutable – returns new string after changes

7. Array Object

The Arrayobject in JavaScript is a built-in object used to store multiple values in a singlevariable. Arrays are versatile data structures that allow storing items like numbers, strings, objects, and even other arrays.

What is an Array?

An array is a collection of items stored in a single variable. Each item has a numeric index, starting from 0.

let fruits = ["Apple", "Banana", "Cherry"];

Creating Arrays

1. Array Literal (Recommended)

let colors = ["Red", "Green", "Blue"];

2. Using new Array() Constructor

let numbers = new Array(1, 2, 3, 4);

3. Creating an Empty Array

letemptyArr = [];

Avoid using new Array(size) unless you know what you are doing. It creates an empty array of a specific length but without elements.

Array Properties

Property	Description
length	Returns the number of elements in the array

letarr = [10, 20, 30];
console.log(arr.length); // Output: 3

Common Array Methods

Adding & Removing Elements

Method	Description
push(item)	Adds item to the end
pop()	Removes and returns the last item
unshift(item)	Adds item to the beginning
shift()	Removes and returns the first item
splice(start, deleteCount, items...)	Adds/removes elements at a specific index
slice(start, end)	Returns a shallow copy of a portion of the array

Searching & Testing

Method	Description
indexOf(item)	Returns the first index of the item
lastIndexOf(item)	Returns the last index of the item
includes(item)	Checks if the array contains the item
find(callback)	Returns the first element that satisfies the condition
findIndex(callback)	Returns the index of the first element that satisfies the condition
some(callback)	Returns true if at least one element passes the test
every(callback)	Returns true if all elements pass the test

Iteration and Transformation

Method	Description
forEach(callback)	Executes a function on each element
map(callback)	Creates a new array with transformed values
filter(callback)	Creates a new array with elements that pass the test
reduce(callback, initialValue)	Reduces array to a single value
flat(depth)	Flattens nested arrays
flatMap(callback)	Maps and flattens results into a new array

Sorting & Reversing

Method	Description
sort(compareFunction)	Sorts the array elements in-place
reverse()	Reverses the order of the elements

Joining & Converting

Method	Description
join(separator)	Joins all elements into a string
toString()	Converts array to a string (comma-separated)
Array.isArray()	Checks whether a variable is an array

Examples

Example 1: Creating and Accessing Elements

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Output: "Apple"
console.log(fruits.length); // Output: 3
```

Example 2: Adding and Removing

```
fruits.push("Mango");
console.log(fruits); // ["Apple", "Banana", "Cherry", "Mango"]
fruits.pop();
console.log(fruits); // ["Apple", "Banana", "Cherry"]
```

Example 3: Iterating Over an Array

```
fruits.forEach(function(item, index) {
console.log(index + ": " + item);
});
```

Example 4: Transforming with map()

```
let numbers = [1, 2, 3];
let squared = numbers.map(n => n * n);
console.log(squared); // [1, 4, 9]
```

Example 5: Filtering with filter()

```
let scores = [45, 67, 89, 30];
let passed = scores.filter(score => score >= 50);
console.log(passed); // [67, 89]
```

Example 6: Reducing with reduce()

```
let total = [10, 20, 30].reduce((acc, curr) =>acc + curr, 0);
console.log(total); // 60
```

Use Cases of Array Object

- Storing lists of items (names, numbers, etc.)
- Iterating and transforming data
- Filtering and sorting information
- Storing results from APIs or form data
- Creating dynamic UI components

Important Notes

- JavaScript arrays are dynamic — they grow/shrink in size.
- Arrays can store mixeddata types, but it's best to use uniform types.
- Use Array.isArray**()** to check if a variable is truly an array (as typeof returns "object").

Table

Feature	Description
Object	Array
Purpose	Store and manipulate ordered collections of data
Indexing	Starts from 0
Property	Length
Common Methods	push(), pop(), map(), filter(), reduce(), sort()
Use Cases	Lists, filtering, mapping, sorting, iteration, UI building
Type Check	Array.isArray(var)

Overview Table

Object	Purpose	Example Method / Property
window	Global browser context	alert(), setTimeout()
navigator	Browser information	navigator.appName
document	HTML document interaction	getElementById()
form	HTML form access and data	document.forms
Date	Dates and time manipulation	getDate(), getFullYear()
String	Text processing	toUpperCase(), replace()
Array	Collection of values	push(), length, sort()

4.2 DYNAMIC HTML WITH JAVA SCRIPT

What is Dynamic HTML (DHTML)?

Dynamic HTML (DHTML**)** is not a language itself, but a collection of technologies used together to create interactive and dynamic websites. DHTML allows web pages to changeafter they are loaded, without requiring a full page reload.

DHTML Combines:

- **HTML** – to define structure/content
- **CSS** – to style and position elements
- **JavaScript** – to add interactivity and control behavior
- **DOM (Document Object Model)** – to access and manipulate HTML elements dynamically

Purpose of DHTML

- Enhance user experience with interactive and responsive interfaces
- Create animations, dropdown menus, form validation, etc.
- Update content dynamically without reloading the entire page

Core Components of DHTML

Component	Role
HTML	Provides the basic structure and elements of the page
CSS	Styles elements dynamically (e.g., color, font, layout)
JavaScript	Adds functionality and manipulates page content
DOM	Allows access and modification of elements on the fly

4.2.1 How JavaScript Enables DHTML

JavaScript is the engine behind DHTML. Using JavaScript, you can:

- Access and modify HTML elements (via DOM)
- Change CSS styles dynamically
- Respond to user events (click, hover, input, etc.)
- Insert, remove, or replace content on the page

Example: Changing Content Dynamically

```
<!DOCTYPE html>
<html>
<head>
<title>DHTML Example</title>
<script>
functionchangeContent() {
document.getElementById("demo").innerHTML = "Content changed using DHTML!";
   }
</script>
</head>
<body>
<h2 id="demo">Original Content</h2>
<button onclick="changeContent()">Click Me</button>
</body>
</html>
```

Explanation:

- JavaScript accesses the element using getElementById
- The content is updated dynamically without reloading the page

DHTML Features Enabled by JavaScript

Feature	Description
Event Handling	Responds to user actions like clicks, keypresses, hovers
DOM Manipulation	Modifies HTML structure in real-time
CSS Style Control	Changes the look of elements dynamically
Animations	Moves, fades, or transforms elements smoothly
Form Validation	Validates user input before submission
Real-Time Content Updates	Updates parts of the page based on conditions or input

4.2.2 DOM Methods in DHTML

JavaScript Method	Purpose
getElementById()	Access a specific element
getElementsByClassName()	Access multiple elements by class
innerHTML	Set or get the content inside an element
style.property	Change element style (e.g., style.color)
createElement()	Create new HTML elements
appendChild()	Add new elements to the DOM
removeChild()	Remove elements from the DOM

Dynamic Style Example

```
<script>
functionchangeColor() {
document.getElementById("text").style.color = "red";
```

```
  }
</script>
<p id="text">Hello, world!</p>
<button onclick="changeColor()">Change Color</button>
```

Event Handling Example

```
<script>
functionshowMessage() {
alert("You clicked the button!");
  }
</script>
<button onclick="showMessage()">Click Here</button>
```

Advantages of DHTML

- No page reload required for updates
- Better user interactivity and engagement
- Fast and responsive UI
- Enables advanced features like sliders, popups, drag-drop, etc.

Disadvantages of DHTML

- Complex for beginners to debug and maintain
- Not all older browsers may support advanced DHTML features
- Too much DHTML can affect page performance

Real-Life Use Cases

- Dynamic menus and tooltips
- Live form validation (email/password check)
- Content sliders and carousels
- Expandable/collapsible sections
- Interactive games and visualizations
- AJAX-powered web applications (e.g., Gmail, Facebook)

Table

Feature	Description
Full Form	Dynamic HTML
Key Technologies	HTML, CSS, JavaScript, DOM

Feature	Description
Driven By	JavaScript
Main Purpose	Make web pages dynamic and interactive
Core Capabilities	Content change, style update, event handling
Use Cases	Menus, animations, form validation, pop-ups

Note:DHTML with JavaScript is a powerful combination that allows developers to build interactive, responsive, and dynamic web applications**.** It forms the foundation of modern web interactivity, and understanding how JavaScript works with HTML and CSS is key to mastering front-end development.

4.3 SUMMARY

JavaScript provides powerful built-in objects like Window**,** Navigator**,** Document**,** Form**,** Date**,** String**,** and Array that form the core of dynamic web development. The Window object serves as the global object and includes properties and methods like alert(), location, and setTimeout(). The Navigator object offers browser-specific details such as appName, userAgent, and onLine status. The Document object allows access and manipulation of HTML elements using methods like getElementById() and querySelector().

The Form object is used to access and validate form inputs dynamically. The Date object enables working with dates and times through methods like getFullYear() and setDate(). The String object allows text manipulation with methods such as toUpperCase(), replace(), and split(). Arrays, represented by the Array object, can hold multiple values and support powerful methods like map(), filter(), and reduce(). These objects enable developers to create responsive, data-driven interfaces.

In addition, Dynamic HTML (DHTML) integrates HTML, CSS, JavaScript, and the DOM to create interactive and visually dynamic web pages. JavaScript is essential to DHTML, enabling real-time content updates, event handling, and style changes without reloading the page.

4.4 KEY TERMS

Window Object,Navigator Object**,** Document Object,Form Object,Date Object,String Object,Array Object,Dynamic HTML (DHTML),Document Object Model (DOM),Event Handling**.**

4.5 SELF-ASSESSMENT QUESTIONS

1. What is the role of the Window object in JavaScript?
2. How can the Navigator object be used to detect browser information?
3. How can you access and validate form data using the Form object in JavaScript?
4. What are some useful methods of the Date object to retrieve or set date values?
5. How do String methods like replace() and split() help in text manipulation?
6. What is Dynamic HTML (DHTML), and how does JavaScript contribute to it?

4. 6 FURTHER READINGS

1. JavaScript: The Definitive Guide, Seventh Edition by David Flanagan. O'Reilly Media.
2. Eloquent JavaScript: A Modern Introduction to Programming, Third Edition by MarijnHaverbeke. No Starch Press.
3. Beginning JavaScript, Fifth Edition by Jeremy McPeak. Wrox (Wiley).
4. The complete Reference Java 2 Fifth Edition by Patrick Naughton and Herbert Schildt. TMH.

Dr. Kampa Lavanya

LESSON-5
XML BASICS AND DATA PROCESSING IN WEB APPLICATIONS

AIM AND OBJECTIVES:

- Understand the purpose and syntax of Document Type Definition (DTD) for defining the structure of XML documents.
- Learn how to use XML Schemas (XSD) to enforce data types and validation rules in XML files.
- Explore the Document Object Model (DOM) to access, modify, and traverse XML documents programmatically.
- Gain knowledge of presenting XML data using technologies like XSLT for transformation and display.
- Compare and utilize XML processors such as DOMandSAX for parsing and handling XML data efficiently.

STRUCTURE:

5.1INTRODUCTION TO XML

XML (eXtensible Markup Language) is a simplified subset of SGML (Standard Generalized Markup Language), a powerful markup language adopted as a standard by the International Organization for Standardization (ISO). SGML was originally developed to add structure and formatting to data in a way that could be used across various applications.

Unlike other languages that focus on how data is displayed, XML is designed to describe whatthe data is. It emphasizes data structure rather than presentation. XML was introduced as

a recommendation by the World Wide Web Consortium (W3C**)** to facilitate data sharing and transportation across systems in a platform-independent manner.

Markup and Tags

In XML, markup refers to the set of instructions (called tags) used to define elements within a document. These tags help structure the data but do not specify how it should be displayed. XML syntax closely resembles HTML, but its purpose is different: while HTML formats data for display, XML organizes data for storage and transport.

5.1.1 Basic XML Structure

Below is an example of a simple XML document:

```
<?xml version="1.0"?>
<college>
<studdetail>
<regno>05j0a1260</regno>
<name>
<firstname>karthik</firstname>
<lastname>btech</lastname>
</name>
<country name="india"/>
<branch>csit</branch>
</studdetail>
</college>
```

- The first line is a processing instruction that declares the file as an XML document and specifies the version.
- <college> is the root element, and all other elements are nested within it.

Well-Formed vs Valid XML

- A well-formed XML document adheres strictly to XML syntax rules:
  - All tags must be properly opened and closed.
  - Tags must not overlap.
  - Empty tags must be self-closed (e.g., <tag/>).
  - The document must include an XML declaration at the top.
- A valid XML document is not only well-formed but also follows a defined DTD **(**Document Type Definition**)** or XMLSchema that specifies its structure and allowed elements. XML parsers can be used to check both well-formedness and validity.

XML Elements and Rules

XML documents are composed of:

- **Elements** (the core content)
- **Control information** (like comments and declarations)
- **Entities** (reusable data)

Key characteristics of XML elements include:

- **Nesting Tags:** XML requires proper nesting. If one tag is opened inside another, it must be closed before the outer tag is closed.
- <parent>
- <child>Content</child>
- </parent>

- **Case Sensitivity:** XML is case-sensitive. <Name> and <name> are treated as different tags. It's a best practice to use lowercase for all tags.
- **Empty Tags:** Tags without content must be self-closed using a forward slash:
- <country name="india"/>
- **Attributes:** Elements can include **attributes** to hold additional information:
- <country name="india"/>

Attributes should not replace elements when the data is complex or needs further structure.

Control Information in XML

In XML, control information refers to special components that provide instructions and structure to the document. There are three main types of control information:

1. **Comments**

Comments in XML are used to include notes or explanations within the document that are ignored by the parser.

- Syntax: <!-- This is a comment -->
- XML comments are similar to those in HTML.

2. **Processing Instructions (PIs)**

These are special instructions intended for applications that process the XML file.

- Example: <?xml version="1.0"?>
- This declaration tells the XML processor which version of XML is being used.

3. **Document Type Declarations (DOCTYPE)**

A Document Type Declaration links an XML document to a DTD (Document Type Definition), which defines its structure and the rules for validation.

- Syntax: <!DOCTYPE element SYSTEM "filename.dtd">
- Example: <!DOCTYPE cust SYSTEM "customer.dtd">
- The DTD can be either internal (within the XML) or external (in a separate file).

Entities in XML

Entities in XML are reusable content placeholders used to store small pieces of data that may be repeated throughout the document. These help in maintaining consistency and manageability.

For example, an XML document using control information and entities might look like this:

```
<?xml version="1.0"?>
<!DOCTYPE stud SYSTEM "student.dtd">
<college>
<studdetail>
<regno>mc20001</regno>
<name>
<firstname>feroz</firstname>
<lastname>pg</lastname>
</name>
<country name="india"/>
<branch>cse</branch>
</studdetail>
</college>
```

In this example:

- The <?xml version="1.0"?> is a processing instruction.
- The <!DOCTYPE stud SYSTEM "student.dtd"> links the XML file to an external DTD.

- Elements like <college> and <studdetail> are structured using rules defined in the DTD.

5.2 DOCUMENT TYPE DEFINITION

1. Introduction to DTD

Document Type Definition (DTD) defines the structure and the legal elements and attributes of an XML document. It acts as a blueprint or grammar that XML documents must follow to be considered valid.
While XML provides flexibility in data representation, DTD ensures data consistency, validity, and structure conformity across documents.
DTD can be written as part of the XML document (internal DTD) or in a separate file (external DTD**).**

2. Purpose of DTD

- To validate the structure and content of XML documents.
- To ensure data integrity and uniformity across systems.
- To define rules for:
  - Elements and their hierarchy
  - Attributes of elements
  - Entities
  - Notations

3. Types of DTD

a. Internal DTD

Defined within the XML document itself, using the <!DOCTYPE> declaration.
Syntax:

```
<?xml version="1.0"?>
<!DOCTYPE root-element [
<!ELEMENT element-name (child-elements)>
<!ATTLIST element-name attribute-name attribute-type #default>
]>
<root-element>
  ...
</root-element>
```

Example:

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
<to>Alice</to>
<from>Bob</from>
```

<heading>Reminder</heading>
<body>Meeting at 10 AM</body>
</note>

b. External DTD
Stored in a separate .dtd file and referenced from the XML document.
XML Document:
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
<to>Alice</to>
<from>Bob</from>
<heading>Reminder</heading>
<body>Meeting at 10 AM</body>
</note>

note.dtd (External DTD File):
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>

4. DTD Components

a. Elements
Defines the allowable content of an element.
Syntax:
<!ELEMENT element-name (child-elements | #PCDATA | ANY | EMPTY)>

Examples:
<!ELEMENT title (#PCDATA)><!-- Text-only -->
<!ELEMENT book (title, author)><!-- Nested structure -->
<!ELEMENT page ANY><!-- Any content -->
<!ELEMENTimg EMPTY><!-- Empty element -->
b. Attributes
Defines attributes for elements and their data types.

Syntax:
<!ATTLIST element-name attribute-name attribute-type default-value>
Attribute Types:

- CDATA: Character data
- ID: Unique identifier
- IDREF: Reference to another ID
- IDREFS: Multiple references
- NMTOKEN: Name token
- ENUMERATION: List of allowed values

Default Values:

- #REQUIRED
- #IMPLIED
- #FIXED "value"
- "default value"

Example:

<!ATTLIST book isbn CDATA #REQUIRED>
<!ATTLIST book category (fiction | nonfiction | reference) "fiction">

c. Entities

Used to define constants or placeholders that can be reused.

Syntax:

<!ENTITY entity-name "replacement-text">

Example:

<!ENTITY author "ABC”

Usage in XML:

<creator>&author;</creator>

d. Comments in DTD

<!-- This is a comment -->

5. DTD Element Content Types

Type	Description
#PCDATA	Parsed Character Data (text)
EMPTY	Element has no content
ANY	Element can contain any content
Child list	Defines specific child elements and their order

Modifiers:

- ? – zero or one
- – zero or more
- + – one or more
- | – choice
- , – sequence

Example:

<!ELEMENT name (first, middle?, last)>
<!ELEMENT phone (home | mobile)>
<!ELEMENT address (street, city, state, zip)>

6. Advantages of DTD

- Simplicity and ease of use
- Useful for simple validation tasks
- Wide support by XML parsers
- Promotes consistency and reusability

7. Limitations of DTD

- No support for data types beyond text (e.g., integer, date)
- Limited namespace support
- Cannot enforce constraints like min/max values or string patterns
- Written in a different syntax (not XML-based)

Alternative: XML Schema (XSD) overcomes these limitations and is written in XML itself.

8. Validating XML with DTD
To validate an XML document:

- Use an XML parser (e.g., Xerces, DOM, SAX)
- Ensure the DOCTYPE declaration correctly references the DTD
- Check the document structure, elements, and attributes against the DTD rules

9. Table

Feature	Description
Full form	Document Type Definition
Purpose	Validates XML structure and content
Defined in	Internally in XML or externally as a .dtd file
Key components	Elements, attributes, entities, content models
Limitation	No support for data types or namespaces
Replacement	XML Schema (XSD) for advanced validation

Library XML with DTD:
library.xml

```
<?xml version="1.0"?>
<!DOCTYPE library SYSTEM "library.dtd">
<library>
<book isbn="12345">
<title>XML Basics</title>
<author>John Smith</author>
</book>
</library>
```

library.dtd

```
<!ELEMENT library (book+)>
<!ELEMENT book (title, author)>
<!ATTLIST book isbn CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

5.3 XML SCHEMAS

1. Introduction to XML Schema
XML Schema Definition (XSD) is a powerful way to define the structure, content, and semantics of XML documents. It is a recommendation by the World Wide Web Consortium (W3C) and is considered more powerful and expressive than Document Type Definition (DTD).

Purpose of XML Schema

- To define the structure of an XML document.
- To define the data types of elements and attributes.
- To validate whether an XML document adheres to a specific format.
- To support namespace and extensibility features.

2. Advantages of XML Schema over DTD

Feature	DTD	XML Schema (XSD)
Syntax	SGML-based	XML-based
Data types	Not supported	Strongly supported (e.g., string, int)
Namespaces	Not supported	Fully supported
Custom types	Not available	Available
Reuse of components	Limited	Extensive (via complex types, imports)

3. XML Schema Syntax

XML Schema documents are XML files that define:

- **Elements**
- **Attributes**
- **Data types**
- **Element relationships**

Basic Structure of XML Schema

<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Definitions go here -->
</xs:schema>

4. Defining Elements and Attributes

Defining a Simple Element

<xs:element name="studentName" type="xs:string"/>

Defining an Attribute

<xs:attribute name="id" type="xs:integer"/>

5. Complex Types

Used to define elements that contain:

- Other elements
- Attributes

Example: Complex Type with Child Elements

<xs:element name="student">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
</xs:sequence>
</xs:complexType>
</xs:element>

Example: Complex Type with Attributes

<xs:element name="book">
<xs:complexType>
<xs:attribute name="isbn" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

6. Data Types in XML Schema

XML Schema provides many built-in data types, categorized as:

Primitive Types

- xs:string
- xs:integer
- xs:boolean
- xs:decimal
- xs:date
- xs:time

Derived Types

- xs:positiveInteger
- xs:nonNegativeInteger
- xs:token
- xs:ID

7. Occurrence Constraints

To control the number of times an element can occur:

- **minOccurs** – Minimum number of occurrences
- **maxOccurs** – Maximum number of occurrences

Example

<xs:element name="phone" type="xs:string" minOccurs="0" maxOccurs="3"/>

8. Restriction and Facets

To place constraints on data values using facets:

Example: Restricting String Length

<xs:simpleType name="usernameType">
<xs:restriction base="xs:string">
<xs:minLength value="5"/>
<xs:maxLength value="12"/>
</xs:restriction>
</xs:simpleType>

9. Reusing Schema Components

- **Named Types**: Reuse custom types across the schema
- **Include / Import**: Modularize large schemas

Include Another Schema

<xs:includeschemaLocation="commonTypes.xsd"/>

10. Namespaces in XML Schema

Namespaces prevent element name conflicts.

Example

<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.com/student"
xmlns="http://www.example.com/student"
elementFormDefault="qualified">

11. Validating XML with Schema

XML files can be validated against an XSD to ensure structural and data correctness using tools like:

- XML parsers (e.g., Xerces, XMLSpy)
- IDEs (e.g., Eclipse, IntelliJ)
- Java (via JAXP)

12. Example: XML and XSD
Sample XML Document
<student>
<name>John</name>
<age>21</age>
</student>

Corresponding XSD
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="student">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

13. Tools Supporting XML Schema
- **Editors**: Oxygen XML Editor, XMLSpy
- **Parsers**: Xerces, SAXON
- **Programming APIs**:
  - Java: JAXP**,** JAXB
  - .NET: XmlSchemaSet**,** XDocument

Note:
XML Schema is a robust and feature-rich way to define the structure and constraints of XML documents. It supports a wide range of data types, complex content modeling, namespaces, and extensibility, making it ideal for applications requiring strict data validation and integration across systems.

5.4 DOCUMENT OBJECT MODEL

1. Introduction to DOM
The Document Object Model (DOM) is a W3C standard that defines a platform- and language-neutral interface to access and manipulate the content, structure, and style of XML or HTML documents.
- It represents a document as a tree structure.
- Each part of the document (elements, attributes, text) is a node in the tree.
- DOM allows programs and scripts to dynamically access and update the document's content, structure, and style.

2. Key Features of DOM
- **Tree Structure**: Represents documents as a hierarchy of nodes.
- **Language-Independent**: DOM can be used in Java, JavaScript, Python, etc.
- **Dynamic**: Allows dynamic modification of documents.
- **Standardized**: Defined by the W3C DOM Specification**.**

3. DOM Tree Structure

DOM views an XML/HTML document as a tree of nodes.

Types of Nodes

Node Type	Description
Document Node	Root of the document tree
Element Node	Represents XML/HTML elements
Attribute Node	Represents attributes of elements
Text Node	Represents text content
Comment Node	Represents comments
Processing Instruction	Represents special instructions

Example XML

```
<student>
<name>John</name>
<age>21</age>
</student>
```

Corresponding DOM Tree

```
Document
└── Element: student
    ├── Element: name
    │   └── Text: John
    └── Element: age
        └── Text: 21
```

4. DOM Levels

DOM has been standardized in multiple levels:

- **DOM Level 1**: Core functionalities – tree structure, basic node access.
- **DOM Level 2**: Adds events, style, and support for namespaces.
- **DOM Level 3**: Adds support for loading/saving documents, validation.

5. Accessing DOM with Java (Using JAXP)

Java provides JAXP (Java API for XML Processing) to work with DOM.

Steps to Use DOM in Java

1. Create DocumentBuilderFactory
2. Create DocumentBuilder
3. Parse the XML file to get Document
4. Traverse or modify the DOM tree

Java Example: Reading XML using DOM

```
importjavax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class DOMReadExample {
public static void main(String[] args) throws Exception {
      File inputFile = new File("student.xml");
DocumentBuilderFactorydbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilderdBuilder = dbFactory.newDocumentBuilder();
```

```
        Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();

System.out.println("Root element: " + doc.getDocumentElement().getNodeName());

NodeListnodeList = doc.getElementsByTagName("student");
for (inti = 0; i<nodeList.getLength(); i++) {
            Node node = nodeList.item(i);
if (node.getNodeType() == Node.ELEMENT_NODE) {
                Element element = (Element) node;
System.out.println("Name:                                                    "                                    +
element.getElementsByTagName("name").item(0).getTextContent());
System.out.println("Age:                                                     "                                    +
element.getElementsByTagName("age").item(0).getTextContent());
            }
        }
    }
}
```

6. Common DOM Interfaces (Java - org.w3c.dom)

Interface	Description
Node	Base interface for all nodes
Element	Represents an element
Attr	Represents an attribute
Text	Represents text within elements
Document	Represents the entire XML/HTML document
NodeList	A list of nodes
NamedNodeMap	Map for attributes

7. DOM Operations

a. Traversing Nodes

- getFirstChild(), getLastChild()
- getNextSibling(), getPreviousSibling()
- getParentNode(), getChildNodes()

b. Modifying Document

- createElement(), createTextNode()
- appendChild(), removeChild()
- setAttribute(), removeAttribute()

c. Reading Data

- getNodeName(), getNodeValue(), getNodeType()
- getTextContent(), getElementsByTagName()

8. Advantages of DOM

- **Random Access**: Any node can be accessed anytime.
- **Modifiable**: Nodes can be added, updated, or deleted.
- **Standard Interface**: Supported across multiple languages.
- **Rich Functionality**: Allows full document manipulation.

9. Disadvantages of DOM

- **Memory Intensive**: Loads the entire document into memory.
- **Slower for Large Files**: Not suitable for very large XML documents.

- **More Complex**: DOM API can be verbose and complex for beginners.

10. DOM vs SAX

Feature	**DOM**	**SAX**
Parsing Mode	Loads entire document in memory	Event-based (reads sequentially)
Access	Random access to any part	Sequential access only
Modification	Supports document modification	Read-only
Performance	Slower for large files	Faster and memory-efficient

11. DOM in Web Browsers (JavaScript)
DOM is also widely used in **web browsers** via JavaScript to manipulate HTML documents dynamically.
Example (HTML + JavaScript)

```
<p id="demo">Hello</p>
<script>
document.getElementById("demo").innerHTML = "Hello, DOM!";
</script>
```

Note:
The Document Object Model (DOM) is a critical concept for working with both XML and HTML documents. It offers a structured, object-oriented view of a document, enabling dynamic access and manipulation. While powerful, it should be used carefully for large documents due to memory and performance considerations.

5.5 PRESENTING XML

1. Introduction
Presenting XML refers to the methods and technologies used to display or render the data stored in XML (eXtensible Markup Language) documents in a human-readable and visually appealing format. By itself, XML is only a data representation format; it doesnot define how the data should appear on screen or on paper**.**
To make XML content presentable, we need to use associated technologies that can transform or style XML data into HTML, PDF, plain text, or other output formats.
2. Need for Presenting XML

- XML is self-descriptive but not inherently visual.
- For users to understand or interact with XML content, it must be transformed into a user-friendly format**.**
- Enables data sharing across different platforms and presentation customization**.**

3. Techniques for Presenting XML
There are several technologies and methods for presenting XML data effectively:
A. Using XSLT (Extensible Stylesheet Language Transformations)

- XSLT is a W3C standard used to transform XML data into other formats such as HTML, text, or another XML structure.
- It works by applying templates and rules to match elements in the XML file and transform them accordingly.

Example
XML File: students.xml

```
<students>
```

```
<student>
<name>John</name>
<age>21</age>
</student>
</students>
```

XSLT File: students.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>Student Information</h2>
<table border="1">
<tr><th>Name</th><th>Age</th></tr>
<xsl:for-each select="students/student">
<tr>
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="age"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Linking XML with XSLT

```
<?xml-stylesheet type="text/xsl" href="students.xsl"?>
```

B. Using CSS with XML

- **CSS (Cascading Style Sheets)** can be used to apply basic formatting to XML documents, just like in HTML.
- XML must be well-structured and follow a specific format for CSS to work effectively.
- Best suited for simple styling (fonts, colors, borders).

Example

XML

```
<?xml-stylesheet type="text/css" href="style.css"?>
<note>
<to>Tina</to>
<from>John</from>
<body>Hello, how are you?</body>
</note>
```

CSS (style.css)

```
note {
display: block;
background-color: lightyellow;
padding: 10px;
font-family: Arial;
}

to, from, body {
```

```
display: block;
margin: 5px 0;
}
```

C. Converting XML to HTML via Programming

- You can use programming languages like Java, Python, PHP, or JavaScript to read XML data and present it as HTML.
- Commonly used in **web** applications and dynamic content generation**.**

Example: Using JavaScript

```
<script>
fetch('students.xml')
.then(response =>response.text())
.then(data => {
const parser = new DOMParser();
constxmlDoc = parser.parseFromString(data, "text/xml");
const name = xmlDoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
document.body.innerHTML = "<h1>Student: " + name + "</h1>";
  });
</script>
```

D. Converting XML to PDF or Other Formats

- XML can be converted to PDF using tools like Apache FOP (Formatting Objects Processor)andXSL-FO (XSL Formatting Objects)**.**
- Common in business applications where XML-based data must be printed or archived.

Workflow

1. XML + XSL-FO → Apache FOP → PDF

4. Technologies Involved in XML Presentation

Technology	**Purpose**
XSLT	Transforms XML into HTML, text, or other XML
XSL-FO	Converts XML into print formats like PDF
CSS	Styles XML elements
JavaScript	Dynamically reads and displays XML in web pages
Apache FOP	Converts XSL-FO documents into PDF
Programming APIs	Java (JAXP), Python (lxml), etc., for customized rendering

5. Presentation Best Practices

- Always validate XML before presenting.
- Use XSLT for complex formatting and CSS for simple visual enhancements.
- Ensure cross-browser compatibility when using XML on the web.
- For large documents, consider server-sidetransformation for better performance.
- Use responsive and accessible design when converting to HTML.

6. Applications of Presenting XML

- Web content management systems
- Online reporting systems
- E-commerce catalogs
- Invoice and billing systems
- Educational content presentation
- News and media feeds (RSS/Atom)

Presenting XML is essential to make XML data usable and understandable for human users. Whether through XSLT**,** CSS**,** orprogramming techniques, transforming XML into a readable format bridges the gap between raw structured data and practical user applications.
By leveraging the right tools and technologies, XML data can be presented effectively across web, print, and mobile platforms.

5.6 XML PROCESSOR

1. Introduction

An **XML Processor** is a software component or engine that reads, interprets, and processes XML documents according to defined standards (like XML 1.0 by W3C). It ensures that an XML document is well-formed, and optionally, valid according to a DTD or XML Schema.
The XML processor is also commonly referred to as an XML parser.

2. Types of XML Processors

XML processors are categorized into two main types based on how they access and process the document:

A. Validating Processor

- Checks both well-formedness and validity.
- Validates the document against a DTD or XML Schema.
- Reports errors if the document doesn't conform to the structure.

B. Non-Validating Processor

- Only checks for well-formedness.
- Does not validate against any DTD or Schema.
- Faster and simpler, suitable when validation isn't required.

3. Functions of an XML Processor

Function	Description
Parsing	Reads XML text and constructs a tree or events
Validation	Checks XML against a DTD or Schema (optional)
Reporting Errors	Identifies and reports syntax or structure errors
Providing Interfaces	Supplies access through APIs like DOM or SAX
Data Extraction	Enables retrieval of specific information from XML documents

4. Common XML Processing Models

There are two primary programming models for using XML processors:

A. DOM (Document Object Model)

- Tree-based processing.
- Loads the entire XML document into memory as a tree structure.
- Allows random access, traversal, and modification.

Pros:

- Easy to use and understand.
- Supports both reading and writing.

Cons:

- High memory usage for large documents.

B. SAX (Simple API for XML)

- Event-based processing.
- Parses the document sequentially and generates events (startElement, endElement, etc.).
- No tree is built; ideal for read-only, forward-only access**.**

Pros:

- Fast and memory-efficient.

- Good for large files.

Cons:

- More complex to code.
- No backward access or modification.

5. Popular XML Processors

Processor	Language	Type	Description
Xerces	Java, C++	Validating	Apache XML processor supporting DOM, SAX, Schema
MSXML	C++, COM, VB	Validating	Microsoft XML Parser for Windows platforms
libxml2	C	Validating	Open-source XML parser from the GNOME project
Expat	C	Non-validating	Fast, lightweight stream-oriented parser
JAXP	Java	Both	Java API for XML Processing (uses DOM, SAX, StAX)
lxml	Python	Validating	Powerful Python binding for libxml2 and libxslt

6. Using XML Processors in Java (JAXP)

Java provides JAXP (Java API for XML Processing) which supports DOM, SAX, and StAX models.

Example: Using DOM Parser

```
importjavax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class DOMParserExample {
public static void main(String[] args) throws Exception {
      File inputFile = new File("students.xml");
DocumentBuilderFactorydbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilderdBuilder = dbFactory.newDocumentBuilder();
      Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();

System.out.println("Root element: " + doc.getDocumentElement().getNodeName());
   }
}
```

Example: Using SAX Parser

```
importorg.xml.sax.*;
importorg.xml.sax.helpers.DefaultHandler;
importjavax.xml.parsers.*;

public class SAXParserExample {
public static void main(String[] args) throws Exception {
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
SAXParsersaxParser = factory.newSAXParser();

DefaultHandler handler = new DefaultHandler() {
public void startElement(String uri, String localName, String qName, Attributes attributes) {
System.out.println("Start Element: " + qName);
        }
    };

saxParser.parse("students.xml", handler);
  }
}
```

7. Well-Formedness vs. Validity

Criterion	Well-Formed XML	Valid XML
Syntax Rules	Must follow basic syntax	Must follow syntax and structure
DTD or Schema	Not required	Required for validation
Checked by	All processors	Only validating processors

8. Error Handling in XML Processors

- **Well-formedness errors**: Missing tags, improper nesting, etc.
- **Validation errors**: Mismatch with DTD/Schema definitions.
- Most processors provide mechanisms to report**,** log, and sometimes recover from errors.

9. Performance Considerations

Model	Memory Use	Speed	Use Case
DOM	High	Medium	Small to medium documents
SAX	Low	High	Large, read-only documents
StAX	Medium	High	Event-driven, pull-based parsing

10. Applications of XML Processors

- Web services (SOAP, REST)
- Configuration files (Spring, Maven)
- Data exchange between systems
- Digital publishing
- Document storage and retrieval
- Enterprise systems integration

Note: An XML processor is an essential tool for working with XML documents. It validates, parses, and provides programmatic access to the data, enabling applications to read**,** transform**,** and present XML content.

Choosing the right processor (DOM, SAX, or StAX) depends on the application requirements, such as performance, memory constraints, and the need for validation.

5.7 DOM AND SAX

DOM and SAX in XML Processing

1. Introduction

When working with XML documents programmatically, two primary APIs are commonly used for parsing and processing:

- DOM (Document Object Model)
- SAX (Simple API for XML)

Both provide ways to access the data and structure of XML documents, but they do so using different approaches suited for different use cases.

2. What is DOM (Document Object Model)?

Definition:

DOM is a tree-based parsing method. It represents the entire XML document as a hierarchical tree of nodes in memory. This allows developers to access, navigate, and manipulate any part of the XML document at any time.

How It Works:

- Loads the entire XML document into memory.
- Constructs a tree where each element, attribute, or text is a node.
- Provides random access to any node.

Key Features:

- Tree structure (parent-child relationships)
- Allows read and write access
- Supports navigation and modification

DOM Parser Workflow:

1. Parse the XML document.
2. Create a tree (DOM tree) in memory.
3. Access or modify nodes using methods.

DOM Example in Java:

```
importjavax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;
public class DOMExample {
public static void main(String[] args) throws Exception {
      File file = new File("students.xml");
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
      Document doc = builder.parse(file);
NodeList list = doc.getElementsByTagName("student");
for (inti = 0; i<list.getLength(); i++) {
          Element student = (Element) list.item(i);
System.out.println("Name:                                    "                                    +
student.getElementsByTagName("name").item(0).getTextContent());
      }
}
}
```

3. What is SAX (Simple API for XML)?

Definition:

SAX is an event-based parsing method. Instead of building a tree, it reads the XML document sequentially and fires events (start element, end element, characters) when it encounters different components of the document.

How It Works:

- Reads the XML document line by line**.**

- Generates events such as:
  - startElement()
  - characters()
  - endElement()
- The application must handle these events**.**

Key Features:

- Does not load the entire document into memory.
- Suitable for largeXML documents.
- Fast and memory-efficient**.**

SAX Parser Workflow:

1. Set up a handler to listen for XML events.
2. Parse the document.
3. React to events like element starts and ends.

SAX Example in Java:

```
importorg.xml.sax.*;
importorg.xml.sax.helpers.DefaultHandler;
importjavax.xml.parsers.*;

public class SAXExample {
public static void main(String[] args) throws Exception {

SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
DefaultHandler handler = new DefaultHandler() {
public void startElement(String uri, String localName, String qName, Attributes attributes) {
if (qName.equalsIgnoreCase("name")) {
System.out.print("Name: ");
            }
        }
public void characters(char ch[], int start, int length) {
System.out.println(new String(ch, start, length));
        }
    };

parser.parse("students.xml", handler);
  }
}
```

1. Comparison Between DOM and SAX

Feature	**DOM**	**SAX**
Model	Tree-based	Event-based
Memory Usage	High (loads whole XML in memory)	Low (sequential reading)
Speed	Slower for large files	Faster for large files
Access Type	Random access to any part	Sequential access only
Modification	Supports modification	Read-only

Feature	DOM	SAX
Complexity	Easier to implement	More complex due to event handling
Use Case	Small to medium-sized documents	Large documents, stream processing
Navigation	Built-in methods (e.g., getChild)	No navigation—developer handles flow

5. When to Use DOM

- You need to modify or update XML data.
- Random access to nodes is necessary.
- The XML document is small or medium-sized**.**
- You want a simpler programming model**.**

6. When to Use SAX

- The XML document is very large**.**
- You only need to read data (not modify).
- You require faster processing.
- Memory usage must be minimal.

7. Hybrid Approach

Some applications use a combination of DOM and SAX, known as StAX (Streaming API for XML), which gives the developer more control (pull-based parsing) over the event stream.

Note:BothDOM and SAX provide powerful ways to work with XML data, but their usage depends on the requirements:

- Use DOM when you need fullaccess, modification, and tree-based structure.
- Use SAX when you want efficient, stream-based, and lightweight processing of large XML documents.

Understanding the difference helps in choosing the right parser for your application's performance, memory, and complexity needs.

5.8 SUMMARY

XML (eXtensible Markup Language) is a standardized language used to structure, store, and transport data. A Document Type Definition (DTD) defines the legal building blocks of an XML document by specifying its structure with a list of allowed elements and attributes. In contrast, XML Schemas are more powerful than DTDs, as they allow data type definitions, namespaces, and detailed validation rules using XML syntax itself.

The Document Object Model (DOM) is a tree-based representation of an XML document in memory, allowing developers to access, modify, and navigate the document structure dynamically. Presenting XML involves transforming XML content into human-readable formats using tools like CSS for styling and XSLT (Extensible Stylesheet Language Transformations) for converting XML into HTML or other formats. XML data is not typically presented directly; it requires formatting or transformation to be user-friendly.

To work with XML programmatically, XML Processors (or parsers) are used. These processors validate the document’s structure and convert it into usable data formats for applications. There are two main types of processors: DOM and SAX. The DOM parser reads the entire XML into memory and builds a node tree, suitable for editing and random access. The SAX parser, on the other hand, is event-driven, reading the document sequentially and firing events during parsing—making it ideal for large, read-only documents. DOM is easier for manipulation but memory-intensive, while SAX is faster and more efficient for large-

scale XML processing. Understanding these technologies is essential for developers working with XML in real-world applications.

5. 9 KEY TERMS

XML, DTD, XML Schema (XSD), Well-formed XML, Valid XML, DOM, SAX, XML Processor, XSLT, Namespace.

5. 10 SELF-ASSESSMENT QUESTIONS

1. What is the purpose of a Document Type Definition (DTD) in XML?
2. How does an XML Schema differ from a DTD?
3. What is meant by a well-formed XML document?
4. What is the role of the Document Object Model (DOM) in XML processing?
5. How does SAX process XML documents differently from DOM?
6. What is an XML Processor, and what are its main types?
7. How can XML data be presented to users in a readable format?

5.11FURTHER READINGS

1. Beginning XML, Fifth Edition by David Hunter, Jeff Rafter, and Joe Fawcett. Wiley Publishing.
2. Learning XML, Second Edition by Erik T. Ray. O'Reilly Media.
3. XML in a Nutshell, Third Edition by Elliotte Rusty Harold and W. Scott Means. O'Reilly Media.
4. Internet and World Wide Web: How to Program, Third Edition by Paul Deitel and Harvey Deitel. Pearson Education.

Dr. VasanthaRudramalla

LESSON-6

CGI SCRIPTING

AIM AND OBJECTIVES:

1. Understand the fundamentals of CGI (Common Gateway Interface) and its role in web development.
2. Learn how to develop and deploy CGI applications that interact with web servers.
3. Gain knowledge of processing client requests and handling form data using CGI scripts.
4. Explore the use of CGI.pm and its methods for simplifying CGI programming in Perl.
5. Learn to create dynamic HTML pages that respond to user input and display customized content.

STRUCTURE:

1. INTRODUCTION TO CGI

CGI (Common Gateway Interface) is a standard protocol used to enable web servers to execute external programs, typically scripts, and generate dynamic content. CGI allows web applications to interact with users, process forms, and display customized web pages. CGI scripts can be written in Perl, Python, Bash, or other languages.

Key Points:

- CGI programs are executed by the web server and return HTML to the client browser.
- They process input from HTML forms, URL parameters, or cookies.
- CGI is language-independent; Perl is one of the most popular languages for CGI scripting.

2. Developing CGI Applications

To develop CGI applications, you need:

1. A web server (like Apache) configured to run CGI scripts.
2. A scripting language (Perl, Python, etc.) installed on the server.
3. Proper permissions for executing the script in the server's cgi-bin directory.

Steps:

- Write the script with proper shebang line (e.g., #!/usr/bin/perl).
- Make the script executable (chmod +x script.pl).
- Place the script in the cgi-bin directory of the web server.

3. Processing CGI Input

CGI scripts often handle user input via HTML forms. The input is sent using either:

- **GET method:** Parameters are sent via URL query string.
- **POST method:** Parameters are sent in the HTTP request body (more secure for sensitive data).

In Perl, the CGI.pm module is commonly used to process form data.

4. Returning a Basic HTML Page

A CGI script must output HTTP headers followed by HTML content.

Example: Basic HTML Page using Perl CGI

```
#!/usr/bin/perl
use CGI qw(:standard);

print header();  # HTTP header
printstart_html("Welcome Page");  # HTML start
print h1("Hello, Welcome to CGI Scripting!");  # Header tag
print p("This page is generated using a CGI script.");  # Paragraph
printend_html();  # HTML end
```

Output in Browser:

Hello, Welcome to CGI Scripting!
This page is generated using a CGI script.

5. Introduction to CGI.pm

CGI.pm is a Perl module that simplifies writing CGI scripts.
Common features:

- Handling GET/POST requests
- Generating HTML elements dynamically
- Managing cookies and headers
- Processing file uploads

Loading CGI.pm:

```
use CGI;
my $query = CGI->new();
```

6. CGI.pm Methods

Some useful CGI.pm methods:

- param("name") – fetch value of a form parameter
- header() – send HTTP headers
- start_html("Title") – begin HTML document
- end_html() – end HTML document
- textfield("username") – generate input field
- submit("Submit") – generate submit button

7. Creating HTML Pages Dynamically

CGI scripts can generate HTML dynamically based on user input.

Example: Processing Form Input

HTML Form (form.html):

```
<html>
<body>
<form action="/cgi-bin/greet.pl" method="post">
   Name: <input type="text" name="username">
<input type="submit" value="Greet Me">
</form>
</body>
</html>
```

CGI Script (greet.pl):

```
#!/usr/bin/perl
use CGI qw(:standard);

my $query = CGI->new();
my $name = $query->param("username");

print header();
printstart_html("Greeting Page");
if($name){
print h1("Hello, $name!");
} else {
print h1("Hello, Guest!");
}
printend_html();
```

Input:

- User enters Alice in the form.

Output in Browser:

Hello, Alice!

8. Notes and Best Practices

- Always print header() before HTML content.
- Use POST method for sensitive data to avoid exposing it in the URL.
- Make scripts executable (chmod +x script.pl) and ensure correct permissions in cgi-bin.
- Consider modern alternatives like PHP, Python Flask/Django, or Node.js for new projects, as CGI is older and slower due to creating a new process for each request.

6.2 WHAT IS CGI

CGI (Common Gateway Interface) is a standard protocol that allows web servers to execute external programs, often scripts, and generate dynamic content for web pages. It enables communication between a web server and a program that can process user requests, such as form submissions, database queries, or other server-side logic.

Key Points:

- CGI scripts can be written in Perl, Python, Bash, C, or other languages.
- CGI programs generate HTML or other content and send it back to the browser.
- CGI is executed on the server-side; the user interacts via the browser.
- Every time a CGI script runs, the server starts a new process, which may impact performance for high-traffic websites.

2. Features of CGI

- **Language-independent:** Can be written in any programming language.
- **Handles User Input:** Processes GET and POST requests from HTML forms.
- **Dynamic Content:** Generates dynamic web pages based on user input.
- **Interactivity:** Can interact with databases, files, and other server resources.

3. Developing CGI Programs

Requirements:

1. A web server configured to run CGI (like Apache).
2. A scripting language installed (Perl is common).
3. Scripts stored in the server's cgi-bin directory.
4. Proper execution permissions (chmod +x script.pl).

6.3 DEVELOPING CGI APPLICATIONS

1. Introduction

Developing CGI (Common Gateway Interface) applications involves creating server-side scripts that can process user requests, interact with databases or files, and return dynamic content (usually HTML) to the browser. CGI scripts can be written in Perl, Python, PHP, Bash, or other languages**.**

Purpose:

- Handle user input from HTML forms.
- Generate dynamic web pages based on user requests.

- Integrate server-side logic with client-side interface.

Key Components of a CGI Application:

1. Web server capable of running CGI scripts (e.g., Apache).
2. CGI script stored in the cgi-bin directory.
3. HTML pages/forms to accept user input.
4. Backend logic (optional) like file operations, database access, or calculations.

2. Steps to Develop a CGI Application

1. **Write an HTML form** to collect user input.
2. **Create a CGI script** to process the input.
3. **Set script permissions** so the server can execute it (chmod +x script.pl).
4. **Place the script** in the server's cgi-bin directory.
5. **Test the application** in a web browser.

3. Example CGI Application

Example 1: Greeting Application (Perl)

HTML Form (index.html):

```
<html>
<body>
<h2>Welcome to the Greeting App</h2>
<form action="/cgi-bin/greet.pl" method="post">
   Enter your name: <input type="text" name="username">
<input type="submit" value="Greet Me">
</form>
</body>
</html>
```

CGI Script (greet.pl):

```
#!/usr/bin/perl
use CGI qw(:standard);

# Create a new CGI object
my $query = CGI->new();

# Retrieve parameter from the form
my $name = $query->param("username");

# Send HTTP header and start HTML
```

```
print header();
printstart_html("Greeting Page");

# Generate dynamic output
if ($name) {
print "<h1>Hello, $name! Welcome to CGI applications.</h1>";
} else {
print "<h1>Hello, Guest!</h1>";
}

printend_html();
```

How to Run:

1. Place index.html in your web directory.
2. Place greet.pl in cgi-bin.
3. Make greet.pl executable: chmod +x greet.pl.
4. Access index.html in a browser: http://localhost/index.html.

Input:

- User enters Alice in the form.

Output (Browser):

Hello, Alice! Welcome to CGI applications.

Example 2: Simple Calculator (Addition)

HTML Form (calculator.html):

```
<html>
<body>
<h2>Simple Addition Calculator</h2>
<form action="/cgi-bin/add.pl" method="post">
   Number 1: <input type="text" name="num1"><br>
   Number 2: <input type="text" name="num2"><br>
<input type="submit" value="Add Numbers">
</form>
</body>
</html>
```

CGI Script (add.pl):

```
#!/usr/bin/perl
```

```
use CGI qw(:standard);

my $query = CGI->new();
my $num1 = $query->param("num1");
my $num2 = $query->param("num2");

print header();
printstart_html("Addition Result");

if ($num1 =~ /^\d+$/ && $num2 =~ /^\d+$/) {
my $sum = $num1 + $num2;
print "<h1>The sum of $num1 and $num2 is $sum.</h1>";
} else {
print "<h1>Please enter valid numbers.</h1>";
}

printend_html();
```

Input:

- Number 1: 10
- Number 2: 25

Output (Browser):

The sum of 10 and 25 is 35.

4. Notes on Developing CGI Applications

- Use POST for sensitive data (passwords, personal info) and GET for simple data.
- CGI scripts can interact with databases (MySQL, PostgreSQL) for dynamic content.
- Use CGI.pm methods (param(), header(), start_html(), end_html()) to simplify development.
- CGI applications are simple and effective but can become slow under heavy load since each request creates a new process.
- Modern alternatives: FastCGI, PHP, Node.js, Python Flask/Django for better performance.

Developing CGI applications allows creating interactive web apps with dynamic content using server-side scripts. By combining HTML forms and CGI scripts, developers can process user input and generate custom output efficiently.

6.4 Processing CGI

1. Introduction

Processing CGI (Common Gateway Interface) involves handling the input sent from a client (usually through an HTML form), performing server-side operations (like calculations, file handling, or database interactions), and sending a dynamic response back to the browser.

Key Concepts:

- CGI scripts run on the server and can be written in Perl, Python, or other languages.
- Data from the client is sent via GETorPOST methods.
- The CGI script parses this data, processes it, and generates output (usually HTML).

2. Steps to Process CGI Data

1. **Create an HTML form** to accept user input.
2. **Receive input** in the CGI script using query parameters.
3. **Validate input** to avoid errors or security risks.
4. **Process the input** (e.g., calculations, database queries, text processing).
5. **Return dynamic output** to the browser.

3. Example: Processing User Input in CGI (Perl)

Example 1: Simple Name Echo

HTML Form (nameform.html):

```
<html>
<body>
<h2>Enter Your Name</h2>
<form action="/cgi-bin/process_name.pl" method="post">
   Name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

CGI Script (process_name.pl):

```
#!/usr/bin/perl
use CGI qw(:standard);

# Create CGI object
my $query = CGI->new();
```

```
# Retrieve input from form
my $name = $query->param("username");

# Generate HTML output
print header();
printstart_html("Processing CGI Example");

if ($name) {
print "<h1>Hello, $name! Your input has been processed successfully.</h1>";
} else {
print "<h1>No input received!</h1>";
}

printend_html();
```

How it works:

1. User enters a name and clicks Submit.
2. Data is sent to process_name.pl.
3. CGI script retrieves the data using param().
4. Script generates HTML dynamically and returns it to the browser.

Input:

- Name: Alice

Output (Browser):

Hello, Alice! Your input has been processed successfully.

Example 2: Simple Addition Processor

HTML Form (addform.html):

```
<html>
<body>
<h2>Add Two Numbers</h2>
<form action="/cgi-bin/process_add.pl" method="post">
   Number 1: <input type="text" name="num1"><br>
   Number 2: <input type="text" name="num2"><br>
<input type="submit" value="Add">
</form>
</body>
</html>
```

CGI Script (process_add.pl):

```
#!/usr/bin/perl
use CGI qw(:standard);

my $query = CGI->new();

# Get input values
my $num1 = $query->param("num1");
my $num2 = $query->param("num2");

print header();
printstart_html("Addition Result");

# Validate numeric input
if ($num1 =~ /^\d+$/ && $num2 =~ /^\d+$/) {
my $sum = $num1 + $num2;
print "<h1>The sum of $num1 and $num2 is $sum.</h1>";
} else {
print "<h1>Please enter valid numbers.</h1>";
}

printend_html();
```

Input:

- Number 1: 12
- Number 2: 8

Output (Browser):

The sum of 12 and 8 is 20.

4. Notes on Processing CGI

- **GET vs POST:**
  - **GET:** Data is sent in the URL. Visible and limited in length.
  - **POST:** Data is sent in the request body. Safer for sensitive information.
- **Input Validation:** Always validate input to prevent errors and security issues (e.g., SQL injection, script injection).
- **Dynamic Output:** CGI scripts can generate full HTML pages dynamically based on input, including tables, forms, or reports.
- **Use of CGI.pm:** Simplifies processing with methods like:
  - param() – Retrieve input values

  - header() – Send HTTP header
  - start_html() / end_html() – Generate HTML page structure
- **Execution:** CGI scripts must have execute permissions (chmod +x script.pl) and be placed in cgi-bin.

6.5 RETURNING A BASIC HTML PAGE

1. Introduction

Returning a basic HTML page from a CGI script means that the script generates an HTML document dynamically and sends it as a response to the browser. This is the simplest form of server-side processing, often used as the first step in learning CGI.

Key Points:

- The CGI script generates HTTP headers followed by HTML content.
- The browser interprets the HTML and displays it.
- This can be done in Perl, Python, or other languages supporting CGI.

2. Steps to Return a Basic HTML Page

1. **Create a CGI script** in the server's cgi-bin directory.
2. **Send the HTTP header** to indicate content type (usually text/html).
3. **Generate HTML content** using print statements.
4. **Make the script executable** (chmod +x script.pl).
5. **Access the script** through a browser to see the HTML page.

3. Example Program in Perl (Returning a Basic HTML Page)

CGI Script (hello.pl):

```
#!/usr/bin/perl
use strict;
use warnings;

# Send HTTP header
print "Content-type: text/html\n\n";

# Generate HTML content
print "<html>\n";
print "<head><title>Basic HTML Page</title></head>\n";
print "<body>\n";
print "<h1>Welcome to My First CGI Page!</h1>\n";
print "<p>This HTML page is returned by a CGI script.</p>\n";
print "</body>\n";
```

```
print "</html>\n";
```

Steps to Run:

1. Save the script as hello.pl in the cgi-bin directory of your web server.
2. Make it executable: chmod +x hello.pl.
3. Open a browser and navigate to: http://localhost/cgi-bin/hello.pl.

Output (Browser):

- A web page displaying:

Welcome to My First CGI Page!
This HTML page is returned by a CGI script.

4. Example Program in Python (Returning a Basic HTML Page)

CGI Script (hello.py):

```
#!/usr/bin/env python3

print("Content-Type: text/html\n")
print("<html>")
print("<head><title>Basic HTML Page</title></head>")
print("<body>")
print("<h1>Welcome to My First CGI Page!</h1>")
print("<p>This HTML page is returned by a CGI script written in Python.</p>")
print("</body>")
print("</html>")
```

Steps to Run:

1. Save the script as hello.py in cgi-bin.
2. Make it executable: chmod +x hello.py.
3. Access via browser: http://localhost/cgi-bin/hello.py.

Output (Browser):

- A web page with the heading and paragraph as defined.

5. Notes on Returning Basic HTML Pages

- **HTTP Header:** Every CGI script must start by sending the Content-Type: text/html header followed by a blank line.

- **Dynamic Content:** Even a basic HTML page can be made dynamic by embedding variables or processing input.
- **Debugging:** If the browser shows raw text instead of HTML, check:
  - Correct content-type header
  - Script permissions
  - Script path
- **Escaping Special Characters:** Use <,> for < and > in text to avoid HTML rendering issues.

6.6 Introduction to CGI.pm

1. Introduction

CGI.pm is a Perl module that simplifies the creation of CGI scripts. It provides built-in functions for:

- Handling form input (GET or POST)
- Generating HTML content dynamically
- Managing cookies and HTTP headers
- Creating query parameters and links

Key Points:

- Reduces the complexity of manually parsing query strings.
- Helps generate HTML elements like forms, tables, and lists easily.
- Makes CGI scripts more secure and maintainable.

2. Installing CGI.pm

Most modern Perl distributions include CGI.pm by default. If not, it can be installed via CPAN:

cpan install CGI

3. Using CGI.pm

Basic Steps:

1. Include the module in the script:

use CGI;

2. Create a CGI object:

my $cgi = CGI->new;

3. Generate HTTP headers and HTML content:

```
print $cgi->header();
print $cgi->start_html("Title of Page");
print $cgi->h1("Heading");
print $cgi->end_html();
```

4. Example Program: Returning a Basic HTML Page

CGI Script (hello_cgi_pm.pl):

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;

# Create CGI object
my $cgi = CGI->new;

# Send HTTP header and start HTML
print $cgi->header();
print $cgi->start_html("Welcome Page");

# Add heading and paragraph
print $cgi->h1("Welcome to CGI.pm Page!");
print $cgi->p("This page is generated using Perl's CGI.pm module.");

# End HTML
print $cgi->end_html();
```

Steps to Run:

1. Save as hello_cgi_pm.pl in the cgi-bin directory.
2. Make executable: chmod +x hello_cgi_pm.pl.
3. Access via browser: http://localhost/cgi-bin/hello_cgi_pm.pl.

Output (Browser):

- A web page displaying:

Welcome to CGI.pm Page!
This page is generated using Perl's CGI.pm module.

5. Example Program: Processing Form Input

HTML Form (form.html):

```
<form action="/cgi-bin/process_form.pl" method="post">
  Name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
```

CGI Script (process_form.pl):

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;

my $cgi = CGI->new;

# Get form input
my $name = $cgi->param('username');

# Send HTML response
print $cgi->header();
print $cgi->start_html("Form Response");
print $cgi->h1("Hello, $name!");
print $cgi->end_html();
```

Output (Browser after submitting "Alice"):

Hello, Alice!

6. Key Methods of CGI.pm

Method	Description
header()	Sends HTTP headers to the browser
start_html("Title")	Begins an HTML document with a title
end_html()	Ends an HTML document
h1("text")	Creates an <h1> heading
p("text")	Creates a paragraph <p>

Method	Description
param('name')	Retrieves form input value
textfield('name')	Generates a text input field
submit('value')	Generates a submit button

7. Notes

- CGI.pm is legacy but still widely used for simple Perl-based web applications.
- For large applications, frameworks like Dancer or Mojolicious are preferred.
- Always validate and sanitize form inputs to prevent security issues.

6.7 CGL.pm methods

1. Introduction

CGI.pm provides a rich set of methods to handle web-based interactions using Perl CGI scripts. These methods are used to:

- Create HTML elements dynamically
- Handle user input from forms
- Manage HTTP headers
- Generate cookies and hyperlinks

Using CGI.pm methods makes CGI scripting more structured, readable, and secure.

2. Common CGI.pm Methods

Here are the key methods categorized by functionality:

a) HTTP Header Methods

Method	Description
header()	Sends the HTTP header to the browser. Can specify content type, cookies, etc.

Example:

```
print $cgi->header(-type => "text/html", -charset => "UTF-8");
```

b) HTML Generation Methods

Method	Description
start_html("Title")	Starts an HTML page with a given title
end_html()	Ends the HTML page
h1("text")	Generates an <h1> heading
h2("text")	Generates an <h2> heading
p("text")	Creates a paragraph <p>
hr()	Inserts a horizontal line <hr>
b("text")	Makes text bold <b>
i("text")	Makes text italic <i>

Example:

```
print $cgi->start_html("My Page");
print $cgi->h1("Welcome!");
print $cgi->p("This page is created using CGI.pm.");
print $cgi->end_html();
```

Output:
A simple HTML page with heading **"Welcome!"** and a paragraph.

c) Form Handling Methods

Method	Description
param('name')	Retrieves the value of a form field
textfield('name')	Creates a text input field
password_field('name')	Creates a password input field
submit('value')	Creates a submit button
reset('value')	Creates a reset button
textarea('name')	Creates a multi-line text area
popup_menu(-name=>'menu', -values=>['A','B'])	Creates a dropdown menu

Method	Description
checkbox_group(-name=>'box', -values=>['X','Y'])	Creates multiple checkboxes
radio_group(-name=>'radio', -values=>['Yes','No'])	Creates radio buttons

Example: HTML Form using CGI.pm methods

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;

my $cgi = CGI->new;

print $cgi->header();
print $cgi->start_html("Form Example");
print $cgi->h1("User Feedback Form");

print $cgi->start_form(-method=>'POST', -action=>'/cgi-bin/process_form.pl');
print "Name: " . $cgi->textfield('username') . "<br>";
print "Feedback: " . $cgi->textarea('feedback') . "<br>";
print $cgi->submit('Submit');
print $cgi->end_form();

print $cgi->end_html();
```

Output (Browser):
A web form with fields for Name and Feedback and a Submit button.

d) Form Processing Example

process_form.pl

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;

my $cgi = CGI->new;

my $name = $cgi->param('username');
```

```
my $feedback = $cgi->param('feedback');

print $cgi->header();
print $cgi->start_html("Form Response");
print $cgi->h1("Thank You, $name!");
print $cgi->p("Your feedback: $feedback");
print $cgi->end_html();
```

Output (Browser after submitting form):

Thank You, Alice!
Your feedback: Great website!

e) Other Useful CGI.pm Methods

- cookie() – To create cookies
- redirect('url') – To redirect the user to another page
- img(-src=>'image.jpg', -alt=>'My Image') – To embed images
- a({-href=>'link'}, 'text') – To create hyperlinks
- ul(li('item1'), li('item2')) – To create unordered lists

3. Notes

- CGI.pm methods abstract away HTML and HTTP handling, making scripts cleaner.
- Always validate and sanitize user input to prevent XSS or injection attacks.
- For dynamic web apps, CGI.pm is suitable for small projects; larger applications may use frameworks like Dancer or Mojolicious.

6.8 CREATING HTML PAGES DYNAMICALLY

1. Introduction

Dynamic HTML pages are generated in real-time based on user input, database content, or server-side logic. Unlike static HTML pages, dynamic pages change content depending on parameters like form input, session data, or other backend computations. CGI (Common Gateway Interface) and Perl's CGI.pm module allow us to create these dynamic pages efficiently.

2. Key Concepts

1. **Server-side Scripting:** Code runs on the server and generates HTML content dynamically.
2. **User Input Handling:** Forms or URL parameters provide data to generate personalized content.

3. **Database Interaction:** Dynamic pages can display content from databases (e.g., MySQL).
4. **CGI.pm Methods:** Methods like param(), start_html(), end_html(), h1(), p(), etc., are used to create HTML dynamically.

3. Example 1: Display Dynamic Greeting

Program: greeting.pl

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;

my $cgi = CGI->new;

# Get user input from URL parameter 'name'
my $name = $cgi->param('name') || "Guest";

# Generate dynamic HTML page
print $cgi->header(-type=>'text/html', -charset=>'UTF-8');
print $cgi->start_html("Dynamic Greeting Page");
print $cgi->h1("Hello, $name!");
print $cgi->p("Welcome to our dynamically generated page.");
print $cgi->end_html();
```

Input (URL in browser):

http://localhost/cgi-bin/greeting.pl?name=Alice

Output (Browser):

Hello, Alice!
Welcome to our dynamically generated page.

If no name is provided, it defaults to **"Guest"**.

4. Example 2: Dynamic Form Handling

Program: feedback_form.pl

```
#!/usr/bin/perl
use strict;
use warnings;
use CGI;
```

```
my $cgi = CGI->new;

if ($cgi->param('submit')) {
   # Process form input dynamically
my $user = $cgi->param('username');
my $feedback = $cgi->param('feedback');

print $cgi->header();
print $cgi->start_html("Feedback Response");
print $cgi->h1("Thank you, $user!");
print $cgi->p("Your feedback: $feedback");
print $cgi->end_html();

} else {
   # Display the form dynamically
print $cgi->header();
print $cgi->start_html("Feedback Form");
print $cgi->h1("Provide Your Feedback");
print $cgi->start_form(-method=>'POST');
print "Name: " . $cgi->textfield('username') . "<br>";
print "Feedback: " . $cgi->textarea('feedback') . "<br>";
print $cgi->submit('submit','Submit');
print $cgi->end_form();
print $cgi->end_html();
}
```

Input (Browser):

- Name: Alice
- Feedback: Great website!

Output (Browser after submission):

Thank you, Alice!
Your feedback: Great website!

5. Example 3: Dynamic Table from Data

Program: students.pl

```
#!/usr/bin/perl
use strict;
use warnings;
```

```
use CGI;

my $cgi = CGI->new;

# Example data
my @students = (
    {id=>1, name=>'Alice', age=>14, grade=>8},
    {id=>2, name=>'Bob', age=>15, grade=>9},
    {id=>3, name=>'Charlie', age=>14, grade=>8},
);

# Generate HTML page with table
print $cgi->header();
print $cgi->start_html("Student List");
print $cgi->h1("List of Students");
print "<table border='1'>";
print "<tr><th>ID</th><th>Name</th><th>Age</th><th>Grade</th></tr>";

foreach my $s (@students) {
print "<tr>";
print "<td>$s->{id}</td>";
print "<td>$s->{name}</td>";
print "<td>$s->{age}</td>";
print "<td>$s->{grade}</td>";
print "</tr>";
}

print "</table>";
print $cgi->end_html();
```

Output (Browser):
A dynamically generated HTML table:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

6. Notes

- Dynamic pages are useful for personalized content, dashboards, and forms.
- CGI scripts should be placed in the cgi-bin directory or configured to execute on the server.

- Always sanitize input to prevent injection attacks.
- For larger applications, consider frameworks like Dancer or Mojolicious for dynamic web page generation.

6.9 SUMMARY

Creating HTML pages dynamically using CGI allows web applications to generate content in real time based on user input, server-side logic, or database content. Unlike static HTML pages, dynamic pages can change depending on parameters such as form submissions, URL queries, or session data. CGI (Common Gateway Interface) provides a way for servers to execute scripts and generate HTML output dynamically, and Perl's CGI.pm module simplifies this process with built-in methods. Key CGI.pm methods like param(), start_html(), end_html(), h1(), p(), textfield(), textarea(), and submit() allow developers to handle user input, generate HTML headers, and create structured page elements programmatically.

Dynamic pages can greet users based on URL parameters, display customized responses, or process form submissions in real time. They can also generate tables and lists from internal arrays or database queries, providing a flexible way to present data. Typically, scripts begin by creating a CGI object, processing input parameters, and then printing the HTTP header followed by HTML content. For example, a greeting page can display "Hello, Alice!" if the name parameter is provided, or default to "Guest" if no input exists. Feedback forms can capture user data, process it, and return a confirmation page dynamically. Similarly, student data can be displayed in a table format, allowing easy visualization of multiple records.

Dynamic pages enhance interactivity and personalization in web applications, making them essential for modern web development. However, security is crucial—user input must be sanitized to prevent injection attacks. CGI scripts are typically placed in the server's cgi-bin directory and require proper permissions to execute. For complex applications, frameworks like Dancer or Mojolicious can further simplify dynamic HTML generation. Overall, CGI and CGI.pm provide a foundational and efficient approach to creating dynamic, interactive, and data-driven web pages.

6.10 KEYTERMS

CGI, CGI.pm, Dynamic, HTML, Script, Parameter, HTTP, Method, Table, Textfield, Submit, Server.

6.11 SELF-ASSESSMENT QUESTIONS

1. What does CGI stand for in web development?
2. What is the primary purpose of a CGI script?
3. Which programming languages can be used to write CGI scripts?
4. What is the role of the CGI.pm module in Perl?
5. How do you retrieve form data using CGI.pm?
6. What is the difference between returning a static HTML page and creating a dynamic one?
7. Name two methods provided by CGI.pm for handling HTML form inputs.

8. How does a CGI script send output to the browser?
9. What is the importance of the HTTP header in CGI output?
10. How can you generate a dynamic HTML table using CGI.pm?

6.12 FURTHER READINGS

1. CGI Programming on the World Wide Web by Scott Guelich, ShishirGundavaram, and Shishir Pal; O'Reilly Media. – Comprehensive guide to CGI, Perl CGI.pm, and web application development.
2. Robert W. Sebesta, "Programming the World Wide Web", Third Edition, Pearson Education (2007).
3. Jeffrey C. Jackson, "Web Technologies - A Computer Science Perspective", Pearson Technologies", Addison Wesley (2006)Education (2008).
4. Perl and CGI for the World Wide Web by Elizabeth Castro and Jim Doxsey; Peachpit Press. – Covers developing CGI scripts, processing form data, and dynamic HTML generation.
5. Learning Perl by Randal L. Schwartz, Tom Phoenix, and brian d foy; O'Reilly Media. – Includes CGI.pm usage and Perl scripting for web applications.
6. CGI Programming with Perl by Simon Cozens; Manning Publications. – Focuses on CGI.pm methods, handling input/output, and creating dynamic web pages.
7. Programming the World Wide Web by Robert W. Sebesta; Pearson. – Discusses CGI scripting concepts, processing CGI, and generating HTML pages dynamically.
8. Modern Perl by chromatic; O'Reilly Media. – Updated Perl practices including web applications using CGI.pm.

Dr. Vasantha Rudramalla

LESSON-7
INTRODUCTION TO JDBC AND DATABASE CONNECTIVITY IN JAVA

AIMS AND OBJECTIVES:

- Gain insight into JDBC and recognize its role as a standardized interface for database interaction in Java.
- Identify and understand the key steps required to connect Java applications with relational databases.
- Investigate the different types of JDBC drivers and how they support communication between Java and databases.
- Examine the inner workings and stages of a JDBC connection from initiation to termination.
- Learn to create, handle, and optimize database connections efficiently within Java-based systems.

Structure:

7.1 JDBC: INTRODUCTION TO JDBC

JDBC (Java Database Connectivity) is a Java-based API (Application Programming Interface) that enables Java applications to interact with databases in a platform-independent and standardized way. Developed by Sun Microsystems (now part of Oracle Corporation), JDBC is part of the Java Standard Edition and is widely used for building robust, data-driven applications.

7.1.1 Purpose of JDBC

The main purpose of JDBC is to allow Java applications to perform the following operations on relational databases:

- Establish a connection to a database
- Send SQL queries and update statements
- Retrieve and process the results of SQL queries
- Handle errors and exceptions
- Perform transaction management

Need for JDBC

Before JDBC, accessing databases in Java required platform-specific and third-party APIs, making applications less portable and harder to maintain. JDBC solves this by providing a uniform interface for accessing different types of relational databases like MySQL, Oracle, PostgreSQL, SQL Server, etc., using the same code structure.

JDBC Architecture

JDBC architecture consists of two layers:

1. **JDBC API:** This provides the application-level interface for Java developers to write database code.
2. **JDBC Driver:** This handles the communication between the Java application and the actual database. The driver translates JDBC calls into database-specific calls.

The JDBC APIutilizes a DriverManager along with database-specific drivers to enable seamless and transparent connectivity to various types of relational databases, regardless of vendor differences.TheDriverManager is responsible for selecting and managing the appropriate driver for each database connection request. It supports the operation of multiple drivers simultaneously, allowing connections to several heterogeneous databases within a single Java application.

The following architectural figure: 7.1 illustrates the position of the DriverManager in relation to the JDBC drivers and the Java application:

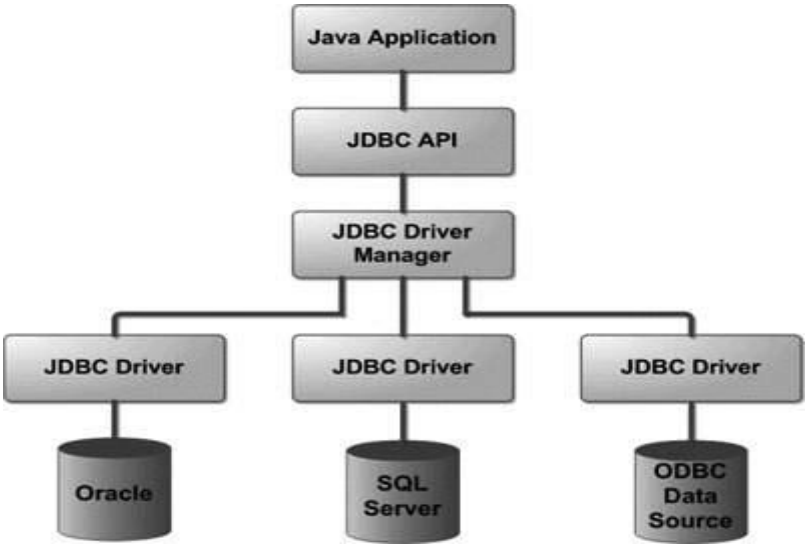


Figure:7.1 the position of the DriverManager in relation to the JDBC drivers and the Java application:

Core JDBC Components

1. **DriverManager:** Manages the list of database drivers and establishes a connection to the database.
2. **Connection:** Represents a session with a specific database.
3. **Statement:** Used to execute SQL queries.
4. **PreparedStatement:** A subclass of Statement that allows precompiled queries with parameters.
5. **CallableStatement:** Used to execute stored procedures.
6. **ResultSet:** Represents the result of a SQL query and allows traversal through query results.
7. **SQLException:** Handles database-related exceptions and errors.

What is a JDBC Driver?

A **JDBC Driver** is a software component that enables Java applications to interact with a specific database. It acts as a bridge between the Java application and the database**,** translating the standard JDBC API calls into database-specific calls that the database can understand and execute.

Since different databases (like MySQL, Oracle, SQL Server, PostgreSQL, etc.) use different communication protocols, JDBC drivers are tailored for each type of database.

7.1.2 Types of JDBC Drivers

There are **four types of JDBC drivers**, also known as JDBC driver types:

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

In a **Type 1 JDBC driver**, a JDBC-ODBC bridge is used to connect Java applications to databases through existing ODBC (Open Database Connectivity) drivers installed on each client machine. To use this approach, the system must be configured with a Data Source Name (DSN), which identifies the target database.When Java was first introduced, this driver type was practical because most relational databases primarily supported ODBC for connectivity. However, with the evolution of native JDBC drivers, the Type 1 driver is now considered outdated and is recommended only for testing or experimental purposes or in cases where no other driver is available.

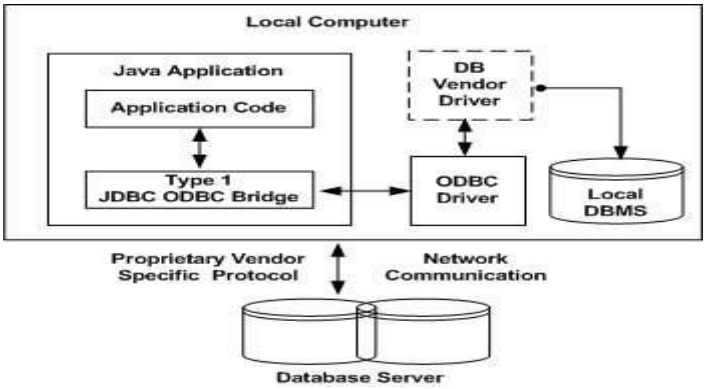


Figure: 7.2Type 1: JDBC-ODBC Bridge Driver

An example of this type of driver is the JDBC-ODBC Bridge provided with JDK 1.2.

2. **Type 2:Native-API Driver**

A Type 2 JDBC driver converts JDBC API calls into nativeC/C++ API calls that are specific to the database being used. These drivers are usually provided by the database vendors and function similarly to the JDBC-ODBC Bridge, but without relying on ODBC. However, they require the native driver to be installed on each client machine.

Since these drivers are database-specific, switching to a different database would require replacing the underlying native APIs, making them less portable. Although largely outdated today, Type 2 drivers can offer better performance than Type 1 drivers by avoiding the overhead introduced by ODBC.

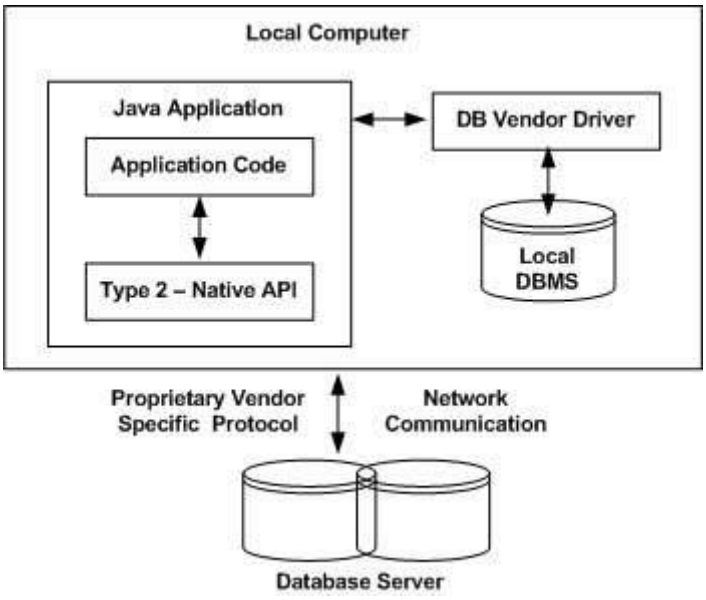


Figure7.3 Type 2: Native-API Driver

A well-known example of a Type 2 driver is the Oracle Call Interface (OCI) driver.

3. Type 3:Network Protocol Driver

A Type 3 JDBC driver follows a three-tier architecture to access databases. In this setup, the JDBC client communicates with a middleware application server using standard network sockets. The middleware server then translates these requests into database-specific calls and forwards them to the appropriate database server.

This driver type is highly flexible and scalable, as it does not require any database-specificcode on the client side. A single Type 3 driver can provide access to multiple types of databases through the middleware. Essentially, the application server acts as a JDBC proxy, handling database operations on behalf of the client.

To effectively use a Type 3 driver, you must be familiar with the configuration of the middleware server. Internally, the server may use a Type 1, 2, or 4 driver to communicate with the actual database, so understanding how it is set up can help optimize performance and compatibility.

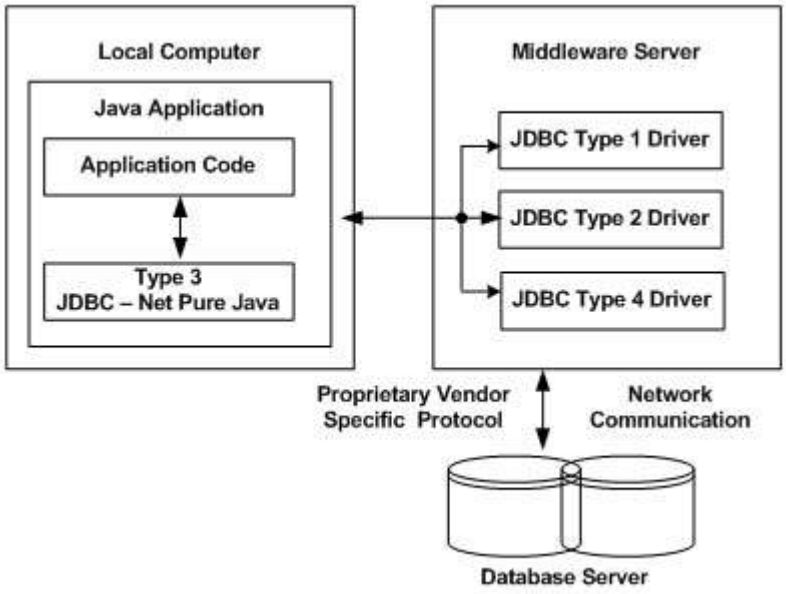


Figure: 7.4 Type 3: Network Protocol Driver

4. Type 4:Thin Driver (Pure Java driver)

A Type 4 JDBC driver is a pure Java driver that communicates directly with the databaseserver using the database vendor's native network protocol over a socket connection. This type of driver offers the highest performance and is typically provided by the database vendor.

Type 4 drivers are highly portable and easy to use, as they require no additional software installation on either the client or server. Additionally, they can often be dynamically downloaded at runtime, further simplifying deployment. An example of a Type 4 driver is MySQL's Connector/J. Due to the proprietary nature of database communication protocols, these drivers are usually developed and maintained by the database vendors themselves.

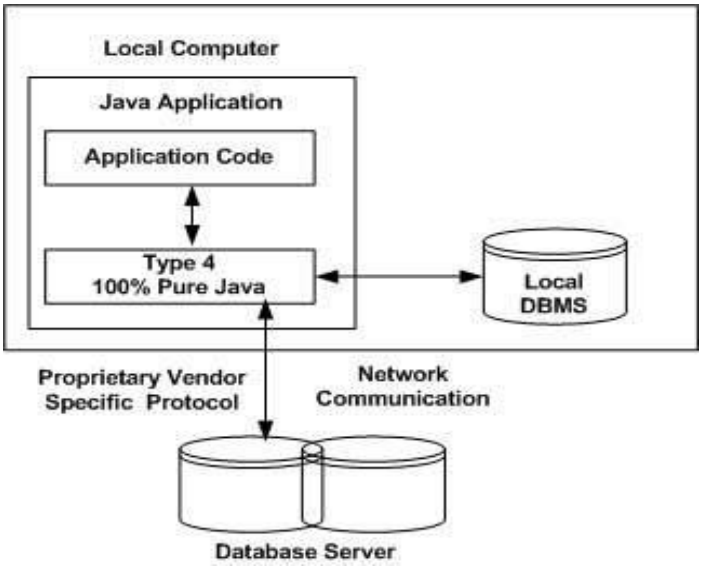


Figure 7.5: Type 4: Thin Driver (Pure Java driver)

Note:

A JDBC driver is essential for establishing communication between a Java application and a relational database. Choosing the right driver type depends on factors like performance, portability, and the specific use case. Type 4 drivers are typically preferred for their simplicity and cross-platform compatibility.

Basic JDBC Workflow

1. Load the JDBC driver.
2. Establish a connection using DriverManager.getConnection().
3. Create a Statement or PreparedStatement.
4. Execute a query using executeQuery() or update using executeUpdate().
5. Process the ResultSet.
6. Close the connection and other resources.

Example Code Snippet

```
import java.sql.*;

public class JDBCExample {
public static void main(String[] args) {
try {
        // Load the JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");
        // Establish the connection
        Connection conn = DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/mydatabase", "username", "password");
        // Create a statement
        Statement stmt = conn.createStatement();
        // Execute a query
ResultSetrs = stmt.executeQuery("SELECT * FROM students");
```

```
        // Process the result set
while (rs.next()) {
System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
        }
        // Close resources
rs.close();
stmt.close();
conn.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Advantages of JDBC

- Platform-independent database connectivity
- Enables connection to various databases using a uniform API
- Supports basic and advanced SQL operations
- Easy integration with enterprise Java technologies (JSP, Servlets, Spring, etc.)
- Good performance with Type 4 drivers

Note: JDBC is a powerful and essential API for Java developers who need to interact with relational databases. It abstracts the complexity of database communication and provides a clean, simple, and extensible framework for data access, making it a cornerstone of Java database programming.

7.2 CONNECTIONS

Establishing a JDBC Connection in Java

Establishing a JDBC (Java Database Connectivity) connection in Java is a straightforward process involving a few essential steps. These steps include importing the required packages, registering the JDBC driver, formulating the database URL, creating a connection object, and finally, closing the connection properly.

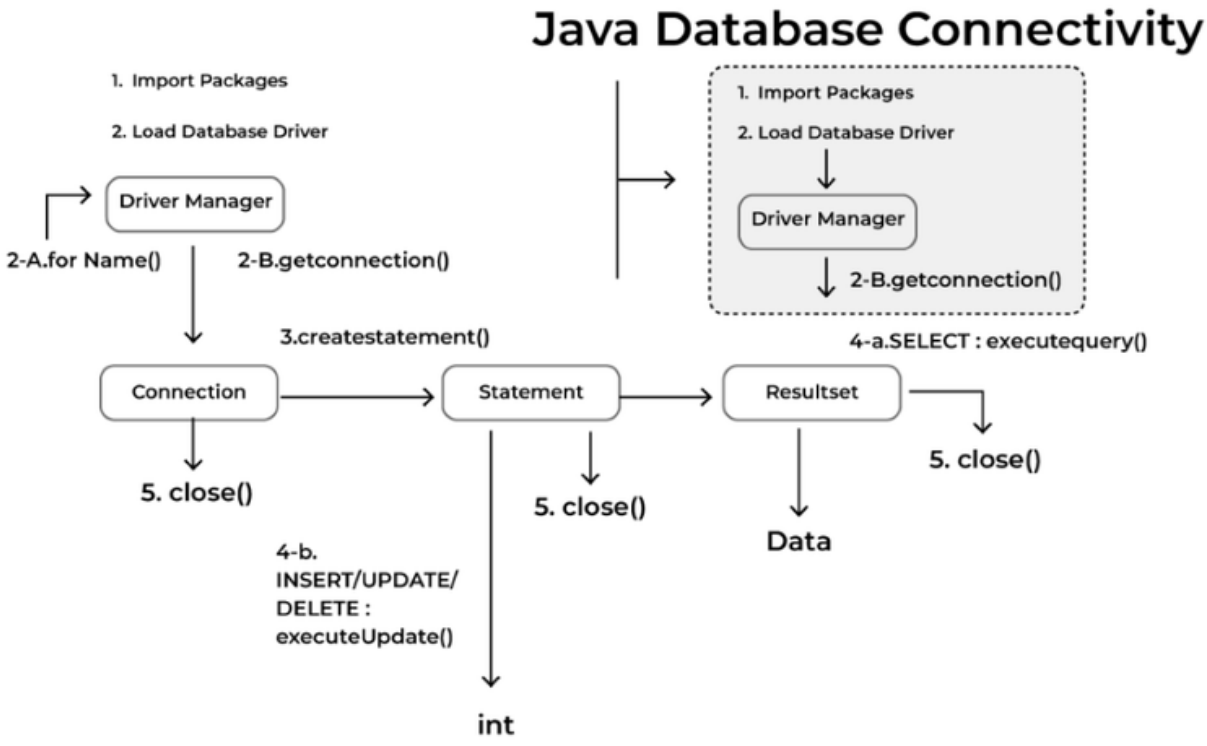


Figure 7.6: Establishing a JDBC Connection in Java

Step 1: Import JDBC Packages

Before working with JDBC, you need to import the necessary classes provided by the Java API. These imports allow your program to interact with databases by enabling operations like querying, inserting, updating, and deleting data.

Add the following import statements at the beginning of your Java source file:

```
import java.sql.*;   // For standard JDBC classes
importjava.math.*;  // For BigDecimal and BigInteger support
```

Step 2: Register the JDBC Driver: Registering the JDBC driver is the process of loading the database-specific driver class into memory. This step must be performed only once in your program. There are two approaches to register a driver:

Approach I – Using Class.forName()

This is the most commonly used and portable method.

```
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
```

```
System.exit(1);
}
```

For non-compliant JVMs, you can use newInstance() with exception handling:

```
try {
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch (ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
System.exit(1);
}
catch (IllegalAccessException ex) {
System.out.println("Error: access problem while loading!");
System.exit(2);
}
catch (InstantiationException ex) {
System.out.println("Error: unable to instantiate driver!");
System.exit(3);
}
```

Approach II – Using DriverManager.registerDriver()

This method is useful for non-JDK compliant JVMs:

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
DriverManager.registerDriver(myDriver);
}
catch (Exception ex) {
System.out.println("Error: unable to register driver!");
System.exit(1);
}
```

Step 3: Formulate the Database URL

A database URL is used to specify the location of the database to which you want to connect. The DriverManager.getConnection() method requires this URL.

Common URL formats for various databases are:

RDBMS	JDBC Driver Class	URL Format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/databaseName
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port/databaseName

RDBMS	JDBC Driver Class	URL Format
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname:port/databaseName

Note: Replace hostname, port, and databaseName with actual values based on your database configuration.

Step 4: Create the Connection Object

To establish a connection, use one of the three overloaded versions of the DriverManager.getConnection() method:

1. Using Database URL, Username, and Password

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password";
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

2. Using a Database URL with Embedded Credentials

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

3. Using a Database URL and Properties Object

```
importjava.util.*;
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties();
info.put("user", "username");
info.put("password", "password");
Connection conn = DriverManager.getConnection(URL, info);
```

Step 5: Close the Connection

After completing database operations, it is crucial to close the connection to free up database resources. Never rely on Java's garbage collector for this task, as it is a poor programming practice.

Always use the close() method in a finally block to ensure the connection is properly closed:

```
try {
  // database operations
} finally {
if (conn != null) conn.close();
}
```

Closing connections properly helps avoid memory leaks and ensures efficient use of DBMS resources.

Note:

In JDBC programming, establishing a connection follows five main steps:

1. **Import** the required JDBC classes.
2. **Register** the JDBC driver.
3. **Formulate** the database URL correctly.
4. **Create** the connection using DriverManager.getConnection().
5. **Close** the connection properly after use.

By following this structure, your JDBC application will maintain reliability, portability, and good resource management practices.7.3 Internal Database Connections

In Java applications, internal database connections refer to the underlying process of how a connection is established between a Java program and a database using the JDBC API. Internally, JDBC acts as a bridge between the Java application and the database, abstracting the complexity of database communication through a set of well-defined interfaces and classes.

Key Components Involved

1. **DriverManager Class**
   - Manages a list of database drivers.
   - Matches the connection request with the appropriate driver using the JDBC URL.
   - Returns a Connection object to the application.
2. **Driver Interface**
   - Every JDBC driver must implement the java.sql.Driver interface.
   - When the driver class is loaded, it automatically registers itself with DriverManager.
3. **Connection Interface**
   - Represents a session with the database.
   - Provides methods for creating Statement, PreparedStatement, and CallableStatement objects.

Internal Connection Flow

1. **Load the JDBC Driver**

2. Class.forName("com.mysql.cj.jdbc.Driver");
   - Registers the driver with DriverManager.
   - Enables the driver to handle future connection requests.
3. **Establish the Connection**
4. Connection con = DriverManager.getConnection(
5. "jdbc:mysql://localhost:3306/mydatabase", "username", "password");
   - The DriverManager scans through registered drivers.
   - The driver that understands the URL (jdbc:mysql://...) takes over.
6. **Driver Internally Performs:**
   - **URL Parsing**: Validates if the URL is compatible.
   - **Authentication**: Uses username and password to connect.

- **Socket Creation**: Establishes a network socket to the database server.
- **Protocol Handling**: Implements database-specific communication protocol.

7. **Return Connection Object**
   - Once the connection is successfully established, a Connection object is returned to the Java program.
   - The application can now execute SQL queries.

Example Code

```
importjava.sql.*;

public class InternalConnectionExample {
public static void main(String[] args) {
try {
        // Step 1: Load JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // Step 2: Establish connection
        Connection con = DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/mydatabase", "root", "password");

System.out.println("Connected Successfully!");

        // Close the connection
con.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Best Practices

- Always **close** the connection to avoid resource leakage.
- Use connection pooling (like HikariCP, Apache DBCP) for performance in enterprise applications.
- Prefer PreparedStatement for security and efficiency over plain Statement.

Note: Internal database connections in JDBC abstract the complex mechanics of establishing a link between a Java application and a relational database. Understanding this process helps developers write efficient and secure database applications.

7.3 INTERNAL DATABASE CONNECTIONS

Introduction

In web technology, internal database connections refer to the behind-the-scenes process by which a web application connects and communicates with a relational database managementsystem (RDBMS) like MySQL, Oracle, or PostgreSQL. These connections are crucial for storing, retrieving, and managing dynamic content in modern web applications.

Key Components Involved

1. **Client**: The end-user interface (browser or mobile app).
2. **Web Server**: Handles HTTP requests and responses.
3. **Application Logic**: Server-side scripts or programs (e.g., Servlets, JSP, PHP, ASP.NET).
4. **Database Driver**: A JDBC driver or equivalent that enables communication with the database.
5. **Database Server**: Stores application data (e.g., MySQL, Oracle DB).

Step-by-Step Internal Workflow

1. Client Sends Request

The process begins when a client (usually a web browser) submits an HTTP request, such as a login form or data entry form.

```
<form action="LoginServlet" method="post">
<input type="text" name="username">
<input type="password" name="password">
<input type="submit" value="Login">
</form>
```

2. Web Server Processes Request

The **Web Server** (e.g., Apache Tomcat) receives the request and forwards it to the appropriate component (Servlet, JSP, etc.).

3. Backend Logic Extracts Input

Server-side code (e.g., a Servlet) extracts input parameters.

```
String uname = request.getParameter("username");
String pwd = request.getParameter("password");
```

4. Load JDBC Driver

Before connecting to the database, the application loads the JDBC driver.
Class.forName("com.mysql.cj.jdbc.Driver");

Internally:

- The class loader loads the driver.
- It registers the driver with DriverManager using a static block:

```
static {
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
}
```

5. Establish Database Connection

Use DriverManager or a DataSource to establish a connection.

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/webapp", "root", "password");
```

Internally:

- DriverManager checks the URL and hands the request to the matching driver.
- A socket connection is established to the database.
- User authentication is validated.
- A Connection object is returned to the application.

6. Create SQL Statement

To perform queries, a Statement or PreparedStatement object is created.

```
PreparedStatementpst = con.prepareStatement(
  "SELECT * FROM users WHERE username=? AND password=?");
pst.setString(1, uname);
pst.setString(2, pwd);
```

Internally:

- The SQL is precompiled (in case of PreparedStatement).
- Parameters are safely injected.
- Execution plan may be reused for performance.

7. Execute SQL Query

The query is sent to the database server.

```
ResultSetrs = pst.executeQuery();
```

Internally:

- JDBC translates the query into native database protocol.
- The query is executed on the database engine.
- Results are streamed back to the application as a ResultSet.

8. Process Results

Results from the ResultSet are processed.

```
if(rs.next()) {
out.println("Login Successful");
} else {
```

```
out.println("Invalid credentials");
}
```

Internally:

- ResultSet manages cursor movement and type conversion.
- Data is converted from SQL types to Java types (e.g., VARCHAR to String).

7. Close Resources

Resources must be closed to release memory and database connections.

```
rs.close();
pst.close();
con.close();
```

Internally:

- JDBC notifies the driver to release sockets, buffers, and connections.
- Connections return to the pool if pooling is used.

Connection Lifecycle Summary

Client Request
↓
Web Server
↓
Servlet/JSP → JDBC Driver → Database
↓
ResultSet ← Response
↓
Client

Connection Pooling (Advanced Approach)

In enterprise web applications, **connection pooling** is used to avoid the overhead of frequent connection creation and closing.

Example using DataSource (JNDI in a Servlet):

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/MyDB");
Connection con = ds.getConnection();
```

Internally:

- Connections are managed in a pool.
- The same physical connection is reused, improving performance.

Best Practices

Practice	Benefit
Use PreparedStatement	Avoids SQL injection
Always close resources	Prevents memory leaks
Use connection pooling	Enhances performance
Validate inputs	Improves security
Use MVC architecture	Clean code organization

Technologies Involved

Layer	Example
Client	HTML, CSS, JS
Server-Side	Java Servlets, JSP, Spring MVC
Database	MySQL, Oracle
JDBC Driver	MySQL JDBC, Oracle JDBC
Web Server	Tomcat, GlassFish

Note: Internal database connections are a core part of web application architecture. They enable dynamic content generation, user authentication, and business logic execution. Understanding the internal workflow helps in writing efficient, secure, and scalable web applications.

7.4 SUMMARY

JDBC (Java Database Connectivity) is a Java API that provides platform-independent access to relational databases. It enables Java applications to connect to databases, execute SQL queries, and process results efficiently. The key components of JDBC include DriverManager, Connection, Statement, PreparedStatement, CallableStatement, and ResultSet, which work together to manage database communication. JDBC drivers translate Java calls into database-specific protocols, and there are four main types: Type 1 (ODBC Bridge), Type 2 (Native API), Type 3 (Network Protocol), and Type 4 (Pure Java/Thin Driver). The general workflow involves loading the driver, establishing a connection, executing SQL queries, processing results, and closing all resources properly.

In web applications, JDBC plays a crucial role in connecting server-side components like Servlets and JSPs with backend databases. When a client sends a request, the server processes it by extracting user input, loading the required JDBC driver, and executing database operations. Internally, the driver manages socket creation, protocol translation, and

authentication to communicate with the database. Best practices include using PreparedStatement to prevent SQL injection, applying connection pooling for better performance, and ensuring proper closure of database connections.

Web technologies such as HTML, JSP, and JDBC drivers work together with web servers like Apache Tomcat to build data-driven applications. Advanced resource management techniques, including connection pooling through JNDI and DataSource, enhance scalability and efficiency. Understanding JDBC architecture and its internal workflow is essential for developing reliable and high-performance Java-based web systems.

7.5 KEY TERMS:

JDBC(Java Database Connectivity), DriverManager, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, SQLException, JDBC,Type 1 Driver, Type 2 Driver, Type 3 Driver, Type 4 Driver, Connection Pooling, DataSource.

7.6 SELF-ASSESSMENT QUESTIONS

1. What is JDBC and why is it used in Java applications?
2. List any three core components of the JDBC API.
3. What are the two main layers of JDBC architecture?
4. Why was JDBC introduced, replacing earlier third-party or platform-specific APIs?
5. What is the role of theDriverManagerclass in JDBC?
6. Differentiate betweenStatementandPreparedStatement**.**
7. What is a Type 4 JDBC driver and why is it widely preferred?
8. What steps are involved in establishing a JDBC connection?
9. What does theConnectioninterface represent in JDBC?
10. How does internal database connection flow work in a web application using JDBC?

7.7 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Editionby Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.
3. Java Programming with Oracle *JDBC* by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBCby Gregory Brill. Sybex.

Dr. Vasantha Rudramalla

LESSON-8

EXECUTING SQL WITH JDBC

AIM AND OBJECTIVES:

- To understand the concept and purpose of JDBC as an interface between Java applications and databases.
- To learn how to establish, manage, and close connections between a Java program and a database using JDBC.
- To explain the working and components of JDBC, including DriverManager, Connection, Statement, and ResultSet.
- To explore the internal process of how JDBC drivers communicate with the database for query execution.
- To develop the ability to write and execute SQL statements in Java programs using JDBC connections effectively.

STRUCTURE:

8.1 INTRODUCTION TO JDBC

What is JDBC?

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with various databases. It provides a standard interface for connecting to relational databases, executing SQL queries, and retrieving results.

JDBC is crucial in web technologies because many web applications need to communicate with databases to store, retrieve, and manipulate data dynamically.

Why JDBC?

- **Database Independence**: JDBC abstracts the database-specific details, allowing developers to write database-agnostic code.
- **Integration**: Enables Java-based web applications (Servlets, JSP, Spring, etc.) to interact with backend databases.

- **Standardization**: Offers a uniform API to work with different relational databases like MySQL, Oracle, SQL Server, PostgreSQL, etc.
- **Supports CRUD Operations**: Create, Read, Update, Delete operations on the database.

JDBC Architecture

The JDBC API follows a client-server model and has four main components:

1. **JDBC Drivers**
   These are specific implementations provided by database vendors to communicate between Java applications and the database. Types include:
   - Type 1: JDBC-ODBC Bridge Driver
   - Type 2: Native-API Driver
   - Type 3: Network Protocol Driver
   - Type 4: Thin Driver (Pure Java)
2. **DriverManager**
   Manages the set of JDBC drivers and establishes connections to the database.
3. **Connection Interface**
   Represents a connection session with a specific database.
4. **Statement Interface**
   Used to execute SQL queries (Statement, PreparedStatement, CallableStatement).
5. **ResultSet Interface**
   Represents the result set from SQL query execution, allowing navigation and retrieval of data.

How JDBC Works in Web Technologies?
In web applications, typically the architecture looks like this:
Client (Browser) → Web Server (Servlet/JSP/Spring MVC) → JDBC Layer →

Database
Step-by-step workflow:

1. **Load the JDBC Driver**
   This registers the driver with the DriverManager.
   Class.forName("com.mysql.cj.jdbc.Driver");
2. **Establish a Connection**
   Use DriverManager.getConnection() with a database URL, username, and password.
   Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/mydb", "root", "password");
3. **Create a Statement**
   Create a statement object to send SQL commands.
   Statement stmt = conn.createStatement();
4. **Execute SQL Queries**
   - For SELECT queries, use executeQuery().
   - For INSERT, UPDATE, DELETE, use executeUpdate().

   ResultSetrs = stmt.executeQuery("SELECT * FROM users");
5. **Process the Results**
   Iterate through the ResultSet to read data.
   while(rs.next()) {
   System.out.println("User: " + rs.getString("username"));
   }

6. **Close Resources**
   Close ResultSet, Statement, and Connection to free resources.

```
rs.close();
    stmt.close();
    conn.close();
```

JDBC in Web Frameworks

- **Servlets and JSPs**: Use JDBC directly to interact with the database.
- **Spring Framework**: Uses JdbcTemplate for simplified database operations built on top of JDBC.
- **Hibernate / JPA**: ORM frameworks that abstract JDBC, but ultimately rely on JDBC to communicate with the database.

Benefits of Using JDBC in Web Technologies

- Enables dynamic content based on database info.
- Supports scalable, enterprise-level database applications.
- Portable across databases with minimal changes.
- Well-integrated into Java EE and Spring frameworks.

Common Challenges

- **Resource Management**: Forgetting to close connections can lead to memory leaks.
- **SQL Injection**: Must use PreparedStatement to safely pass parameters.
- **Error Handling**: Must handle SQLException properly.
- **Connection Pooling**: Needed for performance in high-traffic web applications.

Sample Code Snippet for JDBC in a Web Application

```
public class UserServlet extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
      Connection conn = null;
PreparedStatementps = null;
ResultSetrs = null;
try {
Class.forName("com.mysql.cj.jdbc.Driver");
conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb", "root", "password");
                  String sql = "SELECT username, email FROM users WHERE status =?";
ps = conn.prepareStatement(sql);
ps.setString(1, "active");
rs = ps.executeQuery();
while (rs.next()) {
            String username = rs.getString("username");
            String email = rs.getString("email");
            // process user data or add to request attributes
        }
              } catch (Exception e) {
e.printStackTrace();
      } finally {
try { if (rs != null) rs.close(); } catch (SQLException e) { e.printStackTrace(); }
try { if (ps != null) ps.close(); } catch (SQLException e) { e.printStackTrace(); }
```

```
try { if (conn != null) conn.close(); } catch (SQLException e) { e.printStackTrace(); }
        }
    }
}
```

Note:

- JDBC is a Java API that facilitates communication between Java web applications and databases.
- It provides a standard way to execute SQL commands and process database results.
- It's fundamental for dynamic, database-driven web applications.
- Used in Java Servlets, JSP, Spring, and other Java web frameworks.
- Requires proper resource management, security considerations, and may benefit from connection pooling for high performance.

8.2. CONNECTIONS

A **JDBC Connection** is the link between your Java application and the database, enabling the execution of SQL statements, retrieval of data, and management of transactions.

Purpose of JDBC Connection

The JDBC Connection object is responsible for:

- Establishing a **session** with the database.
- Sending SQL statements to the DB.
- Handling transactions.
- Providing access to metadata.
- Managing resources (connection closing, etc).

Basic JDBC Connection Workflow

Here's the typical process for using JDBC:

1. Load the JDBC Driver
2. Establish the Connection
3. Create SQL Statements
4. Execute SQL Queries
5. Process Results
6. Close the Connection

JDBC Connection in Detail

1. Loading the JDBC Driver

This step loads the JDBC driver class provided by the database vendor.

```
Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL
```

As of JDBC 4.0 (Java 6+), this step is often optional if the driver is available on the classpath.

2. Establishing a Connection

Use the DriverManager class to create a connection to the database.

```
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
```

Connection URL Format

jdbc:subprotocol:subname

Example (for MySQL):

jdbc:mysql://localhost:3306/mydatabase

Component	Description
jdbc	Protocol
mysql	Subprotocol (DB type)
localhost	Hostname of DB server
3306	Port number
mydatabase	Name of the database

3. Common JDBC Driver Connection URLs

Database	JDBC Driver Class	Example URL
MySQL	com.mysql.cj.jdbc.Driver	jdbc:mysql://localhost:3306/mydb
PostgreSQL	org.postgresql.Driver	jdbc:postgresql://localhost:5432/mydb
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@localhost:1521:xe
SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:sqlserver://localhost:1433;databaseName=mydb

4. Example Code: Establishing JDBC Connection

```
importjava.sql.*;
public class JdbcConnectionExample {
public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String user = "root";
        String password = "admin";
try (Connection conn = DriverManager.getConnection(url, user, password)) {
System.out.println("Connected to database successfully!");
        } catch (SQLException e) {
e.printStackTrace();
        }
    }
}
```

JDBC Connection Interface Methods

The java.sql.Connection interface provides many useful methods:

Method	Purpose
createStatement()	Creates a basic SQL statement
prepareStatement(String sql)	Creates a precompiled SQL statement
setAutoCommit(boolean)	Enables/disables auto-commit mode
commit()	Commits transaction manually
rollback()	Rolls back transaction manually
close()	Closes the connection
isClosed()	Checks if connection is closed
getMetaData()	Returns database metadata

Transaction Management

By default, JDBC is in **auto-commit mode**.

Auto-Commit Mode (Default)

Each SQL statement is committed immediately after execution.

```
conn.setAutoCommit(true);  // default
```

Manual Commit Mode

You can turn off auto-commit and commit manually:

```
conn.setAutoCommit(false);

try {
   // Execute SQL statements
conn.commit();  // Commit if all succeed
} catch (SQLException e) {
conn.rollback();  // Rollback if any fails
}
```

Handling Exceptions and Closing Connection

It's important to **close connections** to avoid memory leaks.

Try-With-Resources (Recommended)

```
try (Connection conn = DriverManager.getConnection(url, user, password)) {
   // Work with connection
} catch (SQLException e) {
e.printStackTrace();
}
```

This automatically closes the connection at the end of the block.

JDBC Connection in Web Applications

In Java web apps (Servlets, JSPs, Spring), JDBC is commonly used for database access.

Example in a Servlet:

```
public class UserServlet extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
try (Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"root", "pass")) {
PreparedStatementps = conn.prepareStatement("SELECT * FROM users");
ResultSetrs = ps.executeQuery();
while (rs.next()) {
            // process data
          }
      } catch (SQLException e) {
e.printStackTrace();
      }
   }
}
```

JDBC Connection Pooling (Advanced)

Creating a new connection for every request is expensive. Connection pooling improves performance by reusing existing connections.

Popular Connection Pooling Libraries:

Library	Features
HikariCP	Fast and lightweight

Library	Features
Apache DBCP	Easy integration
C3P0	Automatic testing and recovery

Spring Boot uses HikariCP by default.

Security Tips for JDBC Connections

- Avoid hardcoding DB credentials.
- Use config files or environment variables.
- Always close your connection objects.
- Use PreparedStatement to prevent SQL Injection.

JDBC Connection Best Practices

Practice	Why It's Important
Use try-with-resources	Ensures connections are closed properly
Use connection pooling	Improves performance for web applications
Close ResultSet and Statement	Frees resources and avoids memory leaks
Avoid hardcoded credentials	Improves security
Use transactions where needed	Maintains data integrity

Table

Feature	Description
Connection	Interface to connect Java app to DB
Created By	DriverManager.getConnection()
Needs Driver?	Yes, specific to DB (e.g., MySQL, Oracle)
Connection URL	Follows format: jdbc:subprotocol:subname
Auto-Commit	Default is true, can be set to manual
Use in Web Apps	JDBC connects backend to DB in Servlets, JSPs, Spring etc.
Connection Pooling	Reuses DB connections, improves performance

8.3. INTERNAL DATABASE CONNECTIONS

1. Internal Database Connection

An **Internal Database Connection** refers to the behind-the-scenes process through which a Java application communicates with a database via JDBC.

- It is not visible to the user but is essential for executing SQL commands, retrieving results, and maintaining sessions**.**
- JDBC abstracts these internal processes, so developers focus on Java code rather than database protocols.

2. Components Involved in Internal Connections

When a JDBC program connects to a database, several internal components work together:

Component	Role
DriverManager	Maintains a registry of JDBC drivers and selects the appropriate driver for a database URL.
JDBC Driver	Converts Java method calls into database-specific network protocol commands.

Component	Role
Connection Object	Represents a live session with the database, including metadata like database version, session info, and active transactions.
Statement / PreparedStatement / CallableStatement	Sends SQL queries to the database and processes results.
ResultSet Object	Stores the query results and provides methods to retrieve data row by row.

3. How Internal Database Connections Work

Here's a step-by-step explanation of the internal workflow when a Java program connects to a database:

Step 1: Driver Registration

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

- The JDBC driver class is loaded.
- The driver registers itself with DriverManager, making it available for connection requests.

Step 2: Requesting a Connection

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/world", "root", "12345");
```

- The DriverManager checks its list of registered drivers.
- It selects a driver that supports the provided database URL**.**

Step 3: Authentication and Session Creation

- The driver sends the username and password to the database server.
- The database validates credentials.
- If valid, a **session** is created, and a Connection object is returned to the Java program.

Step 4: SQL Query Execution

- SQL commands executed via Statement or PreparedStatement are internally converted into database-specific requests.
- The driver handles protocol translation, sending the commands over TCP/IP to the database server.

Step 5: Result Processing

- The database executes the query and returns results.
- The driver converts the database-specific response into a ResultSet object.
- The Java application iterates over the ResultSet to retrieve data.

Step 6: Closing the Connection

```
con.close();
```

- Closing the connection informs the driver and database to release resources.
- Internally, the session is terminated, and sockets are closed.

4. Key Features of Internal Database Connections

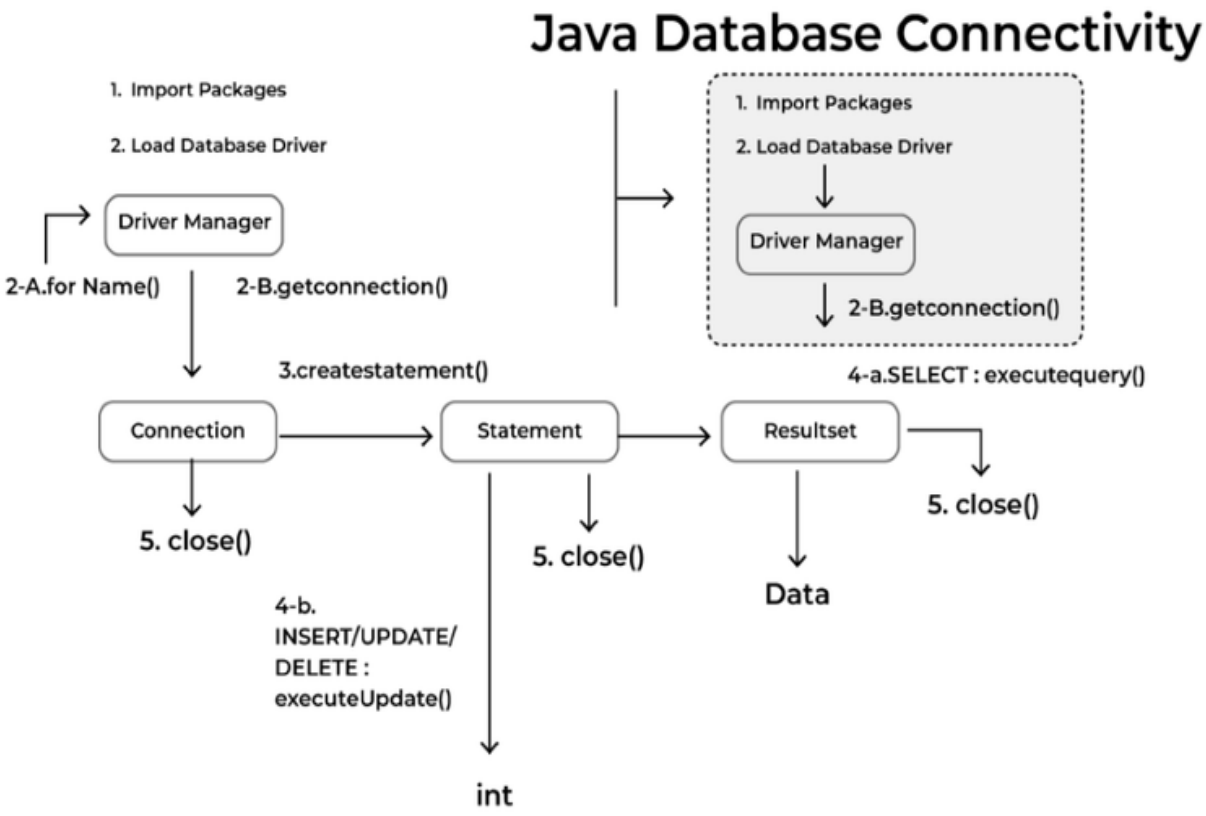
1. **Automatic Protocol Handling:** Converts Java calls into database-specific commands.
2. **Session Management:** Maintains database session metadata such as user, privileges, and active transactions.
3. **Result Handling:** Converts raw database results into ResultSet objects for Java applications.

4. **Security:** Handles authentication and prevents unauthorized access at the session level.
5. **Transparency:** Developers don't need to know the underlying network or protocol details.

5. Advantages of Understanding Internal Connections

- **Optimized Performance:** Helps in implementing connection pooling to reuse connections efficiently.
- **Better Debugging:** Understanding internal flow helps troubleshoot connection failures or slow queries.
- **Security Awareness:** Explains how credentials and queries travel from Java to the database.
- **Resource Management:** Encourages proper closing of connections, statements, and result sets.
- **Scalability:** Enables designing multi-tier applications with minimal connection overhead.

6. Diagram: Internal Database Connection Flow



7. Best Practices for Internal Database Connections

1. Always close connections, statements, and result sets in finally blocks or use try-with-resources**.**
2. Use PreparedStatement to improve performance and prevent SQL injection.
3. Use connection pooling in enterprise applications to reduce connection overhead.

4. Minimize open connections to reduce memory and server load.
5. Handle SQLException properly to catch connection or query failures.

Note:

- Internal database connections are the hidden processes that allow Java applications to communicate with databases via JDBC.
- JDBC DriverManager + Driver + Connection + Statement + ResultSet work together internally.
- Understanding internal connections helps with performance, security, and proper resource management**.**
- While transparent to developers, this internal workflow is critical for building robust, scalable, and secure database applications**.**

8.4. STATEMENTS

In Java, the **Statement** interface of JDBC (Java Database Connectivity) is used to create and execute SQL queries within Java applications.JDBC provides three types of statements to interact with the database:

- **Statement** ->For executing static SQL queries.
- **PreparedStatement** ->For executing parameterized queries.
- **CallableStatement** ->For executing stored procedures.

1. Statement

A Statement object is used for general-purpose access to databases and is useful for executing static SQL statements at runtime.

Syntax:

Statement statement = connection.createStatement();

Execution Methods

- **execute(String sql):** Executes any SQL (SELECT, INSERT, UPDATE, DELETE). Returns true if a ResultSet is returned.
- **executeUpdate(String sql):** Executes DML (INSERT, UPDATE, DELETE). Returns number of rows affected.
- **executeQuery(String sql):** Executes SELECT queries. Returns a ResultSet.

Example:

```
importjava.sql.*;

public class JDBCExample {
public static void main(String[] args) {
try {
        // Step 1: Load the MySQL JDBC Driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // Step 2: Establish a connection to the database
        Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/world", "root", "12345");

        // Step 3: Create a Statement object to send SQL queries
```

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

        // Step 4: Define and execute an SQL SELECT query
        String query = "SELECT * FROM people";
ResultSetrs = stmt.executeQuery(query);

        // Step 5: Process the ResultSet
while (rs.next()) {
            String name = rs.getString("name");
int age = rs.getInt("age");
System.out.println("Name: " + name + ", Age: " + age);
        }
        // Step 6: Close all resources
rs.close();
stmt.close();
con.close();

    } catch (Exception e) {
e.printStackTrace();
    }
  }
}
```

Output: Name and age are as shown for random inputs.

8.5 SUMMARY

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with relational databases in a standardized and efficient way. It enables executing SQL queries, retrieving results, and performing CRUD (Create, Read, Update, Delete) operations, making it essential for developing dynamic, database-driven web applications. JDBC provides a uniform interface for various databases such as MySQL, Oracle, PostgreSQL, and SQL Server, ensuring platform and database independence. The JDBC architecture follows a client-server model and includes key components like JDBC drivers, DriverManager, Connection, Statement, and ResultSet. JDBC drivers act as a bridge between Java programs and databases, converting Java calls into database-specific protocol commands, while the Connection object establishes a session with the database, manages transactions, and provides access to metadata.

The internal database connection process in JDBC involves driver registration, authentication, session creation, SQL execution, result processing, and resource cleanup. Developers use Statement objects to execute SQL queries—Statement for static queries, PreparedStatement for parameterized queries, and CallableStatement for executing stored procedures. Among these, PreparedStatement is preferred for preventing SQL injection and improving performance through precompilation. In web technologies, JDBC is widely used in Servlets, JSP, Spring, and other Java-based frameworks to enable interaction between backend databases and dynamic front-end content.

Proper resource management is vital for maintaining performance and preventing memory leaks; therefore, connections, statements, and result sets must always be closed after use. To enhance performance, connection pooling is employed, which reuses existing database connections instead of creating new ones for each request. JDBC also supports transaction management—both automatic and manual—using methods like commit() and rollback(). Understanding the internal working of JDBC connections helps developers improve security, debugging, and optimization of database operations. Overall, JDBC provides a robust, scalable, and secure foundation for integrating Java applications with relational databases effectively.

8.6 KEY TERMS

JDBC (Java Database Connectivity) **,**DriverManager **,**Connection , JDBC Driver , Connection Pooling , SQL Injection,Transaction Management, Internal Database Connection,Statement.

8.7 SELF-ASSESSMENT QUESTIONS

1. Define JDBC and explain its importance in Java web applications.
2. Explain the architecture of JDBC and describe the role of each component.
3. What are the different types of JDBC drivers? Explain with examples.
4. Describe the process of establishing a JDBC connection, including the connection URL format.
5. Explain the difference between Statement?
6. What is an internal database connection in JDBC? Describe the workflow from driver registration to closing the connection.
7. Explain how transaction management works in JDBC and the difference between auto-commit and manual commit modes.
8. What are the best practices for using JDBC in web applications? Explain with reasons.

8.8 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Editionby Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approachby Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.
3. Java Programming with Oracle JDBCby Donald Bales. O'Reilly Media.
4. Java EE 8 Application Developmentby David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applicationsby Nicholas S. Williams. Wrox/Wiley Publishing.

6. Java 2: Developer's Guide to Web Applications with JDBCby Gregory Brill. Sybex.

Dr. Vasantha Rudramalla

LESSON-9
ADVANCED JDBC: PREPARED AND CALLABLE STATEMENTS

AIM AND OBJECTIVES:

- Understand the purpose of Statement objects in executing SQL queries from Java programs.
- Learn how ResultSet stores and allows navigation through data retrieved from a database.
- Explore PreparedStatement for executing parameterized queries to prevent SQL injection and improve performance.
- Understand CallableStatement for invoking stored procedures in a database from Java applications.
- Develop the ability to perform database operations efficiently using different types of JDBC statements.

STRUCTURE:

9.1. STATEMENTS

1. Introduction

In JDBC (Java Database Connectivity), a **Statement** is an interface used to execute SQL queries against a database. It allows Java programs to interact with a database, retrieve data, and update it. Statements are best suited for static SQL queries that do not change often, unlike PreparedStatement, which is used for dynamic queries with parameters.

Types of Statements

1. **Statement** – Used for simple SQL queries without parameters.
2. **PreparedStatement** – Used for parameterized queries (dynamic queries).
3. **CallableStatement** – Used to execute stored procedures in a database.

2. Key Features of Statement

- Executes SQL queries like SELECT, INSERT, UPDATE, and DELETE.

- Returns ResultSet for queries that retrieve data.
- Simple to use for basic database operations.
- Not secure for dynamic user inputs (prone to SQL injection).

3. Creating a Statement

To create a Statement object, you need:

1. Load the JDBC driver.
2. Establish a connection to the database.
3. Create a Statement object.
4. Execute SQL queries using the statement.

4. Example Program Using Statement

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Java Program: Statement Example

```
importjava.sql.*;

public class StatementExample
 {
public static void main(String[] args)
{
Try
 {
        // 1. Load JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");
            // 2. Establish connection
        Connection con = DriverManager.getConnection(
             "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Create Statement object
        Statement stmt = con.createStatement();

        // 4. Execute SQL query
ResultSetrs = stmt.executeQuery("SELECT * FROM Students");

 // 5. Process ResultSet
System.out.println("ID\tName\tAge\tGrade");
while(rs.next()) {
```

```
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

 // 6. Close connections
rs.close();
stmt.close();
con.close();

      } catch(Exception e) {
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- The program does not take dynamic input; it directly queries the Students table.

Output (Console)

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

6. Executing Update Queries Using Statement

You can also execute INSERT, UPDATE, or DELETE queries using Statement.

```
// Insert a new student
introwsInserted = stmt.executeUpdate(
   "INSERT INTO Students (ID, Name, Age, Grade) VALUES (4, 'David', 13, 7)");
System.out.println(rowsInserted + " row(s) inserted.");
```

Output:

1 row(s) inserted.

7. Note:

- executeQuery() is used for SELECT queries and returns a ResultSet.
- executeUpdate() is used for INSERT**,** UPDATE**,** DELETE queries and returns the number of affected rows.

- For queries with dynamic user input, prefer PreparedStatement to avoid SQL injection.
- Statement is simple but less secure and less efficient for repeated queries.

9.2 RESULTS SETS

1. Introduction

In JDBC, a **ResultSet** is an object that holds the data retrieved from a database after executing a SELECT query using a Statement or PreparedStatement. It acts like a table in memory, allowing you to navigate through rows of data and fetch column values.

Key Points

- Represents tabular data from a database query.
- Cursor-based: can move forward, backward (depending on type), or jump to a specific row.
- Read-only or updatable (depending on how you create it).
- Often used with while(rs.next()) to iterate through rows.

2. Types of ResultSet

Type	Description
TYPE_FORWARD_ONLY	Default. Cursor moves only forward.
TYPE_SCROLL_INSENSITIVE	Cursor can move forward/backward. Changes in DB after query execution are not visible.
TYPE_SCROLL_SENSITIVE	Cursor can move forward/backward. Reflects changes in DB after query execution.

3. Creating a ResultSet

1. Create a Statement object.
2. Execute a SELECT query using executeQuery().
3. Store the returned data in a ResultSet.
4. Iterate through the ResultSet to process data.

4. Example Program Using ResultSet

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Java Program: ResultSet Example

```
importjava.sql.*;

public class ResultSetExample
{
public static void main(String[] args)
 {
try
{
        // 1. Load JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish connection
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Create Statement object
        Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);

        // 4. Execute SELECT query
ResultSetrs = stmt.executeQuery("SELECT * FROM Students");

        // 5. Process ResultSet
System.out.println("ID\tName\tAge\tGrade");
while(rs.next())
{
// iterate forward
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

        // 6. Moving cursor backwards (if scrollable)
System.out.println("\nLast record:");
if(rs.last())
 {
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
        }

        // 7. Close connections
rs.close();
stmt.close();
con.close();
```

```
      } catch(Exception e)
{
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- The program directly queries the Students table. No dynamic input is required.

Output (Console)

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

Last record:
3 Charlie 14 8

6. Updating Data Using ResultSet (Optional)

Some ResultSet objects can be updatable. Example:

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

ResultSetrs = stmt.executeQuery("SELECT * FROM Students WHERE ID=2");
if(rs.next())
{
rs.updateInt("Age", 16); // update age
rs.updateRow();          // commit changes
}
```

This directly updates the database without executing a separate UPDATE query.

7. Note:

- rs.getInt("columnName") or rs.getString("columnName") fetches column data.
- Always close ResultSet and Statement to free resources.
- For large datasets, consider using pagination because ResultSet loads data into memory.
- Scrollable and updatable ResultSets are more flexible but slightly slower.

9.3 PREPARED STATEMENTS

1. Introduction

A **PreparedStatement** is a feature of JDBC used to execute parameterized SQL queries. Unlike a regular Statement, which executes raw SQL strings, PreparedStatement allows placeholders (?) for values, which are supplied at runtime.

Advantages

1. **Prevents SQL Injection** – User input is treated as data, not code.
2. **Improves Performance** – The SQL query is precompiled by the database.
3. **Reusability** – The same query can be executed multiple times with different inputs.
4. **Type Safety** – You can explicitly set the data type of parameters.

2. Syntax

PreparedStatementpstmt = con.prepareStatement("SQL query with ?");

- ? is a placeholder for a value.
- Values are set using methods like:
  - setInt(index, value)
  - setString(index, value)
  - setDouble(index, value)

3. Example Program Using PreparedStatement

Database Setup

Assume a database SchoolDB with table Students:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9

Java Program: Insert Using PreparedStatement

```
importjava.sql.*;

public class PreparedStatementExample
 {
public static void main(String[] args)
{
Try
 {
        // 1. Load JDBC Driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish Connection
```

```
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");

        // 3. Prepare SQL Query with placeholders
        String sql = "INSERT INTO Students (ID, Name, Age, Grade) VALUES (?, ?, ?, ?)";
PreparedStatementpstmt = con.prepareStatement(sql);

        // 4. Set parameter values
pstmt.setInt(1, 3);          // ID
pstmt.setString(2, "Charlie"); // Name
pstmt.setInt(3, 14);          // Age
pstmt.setInt(4, 8);          // Grade

        // 5. Execute query
int rows = pstmt.executeUpdate();
System.out.println(rows + " row(s) inserted.");

        // 6. Close connection
pstmt.close();
con.close();

      }
catch(Exception e)
 {
e.printStackTrace();
      }
   }
}
```

4. Input / Output

Input

- Directly set in the program using pstmt.setXXX() methods.

Output

1 row(s) inserted.

- After execution, the table Students will have:

ID	Name	Age	Grade
1	Alice	14	8
2	Bob	15	9
3	Charlie	14	8

5. Example Program: Select Using PreparedStatement

```
String sql = "SELECT * FROM Students WHERE Age = ?";
PreparedStatementpstmt = con.prepareStatement(sql);
pstmt.setInt(1, 14); // parameter for age
ResultSetrs = pstmt.executeQuery();

while(rs.next()) {
System.out.println(rs.getInt("ID") + "\t" +
rs.getString("Name") + "\t" +
rs.getInt("Age") + "\t" +
rs.getInt("Grade"));
}
```

Output

```
ID   Name     Age   Grade
1    Alice    14    8
3    Charlie  14    8
```

6. Note:

- PreparedStatement is preferred over Statement for dynamic queries.
- Supports batchprocessing for multiple inserts efficiently:

```
pstmt.setInt(1, 4);
pstmt.setString(2, "David");
pstmt.setInt(3, 15);
pstmt.setInt(4, 9);
pstmt.addBatch(); // add to batch

int[] result = pstmt.executeBatch(); // execute all at once
```

- Always close PreparedStatement and Connection to free resources.

9.4 CALLABLE STATEMENTS

1. Introduction

A **CallableStatement** is used in JDBC to execute storedprocedures in a database. Stored procedures are precompiled SQL programs stored in the database.

Advantages

1. **Encapsulation** – Database logic is centralized in procedures.
2. **Performance** – Stored procedures are precompiled.
3. **Security** – Reduces SQL injection risk.
4. **Ease of Maintenance** – Changes in logic don't require Java code changes.

2. Syntax

CallableStatementcstmt = con.prepareCall("{call procedure_name(?, ?)}");

- {callprocedure_name(?, ?)} – ? are placeholders for input/output parameters.
- Parameters can be:
  - **IN** – Input to the procedure
  - **OUT** – Output from the procedure
  - **INOUT** – Input and output

Set parameters with:

- cstmt.setInt(index, value)
- cstmt.setString(index, value)

Register output parameters with:

- cstmt.registerOutParameter(index, type)

3. Example: Stored Procedure in MySQL

Create a stored procedure in MySQL SchoolDB:

```
DELIMITER //

CREATE PROCEDURE GetStudentByID(IN sid INT, OUT sname VARCHAR(50), OUT
sage INT)
BEGIN
   SELECT Name, Age INTO sname, sage FROM Students WHERE ID = sid;
END //

DELIMITER ;
```

4. Java Program Using CallableStatement

```
importjava.sql.*;

public class CallableStatementExample {
public static void main(String[] args) {
try {
        // 1. Load JDBC Driver
Class.forName("com.mysql.cj.jdbc.Driver");

        // 2. Establish Connection
        Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/SchoolDB", "root", "password");
```

```
        // 3. Prepare CallableStatement
CallableStatementcstmt = con.prepareCall("{call GetStudentByID(?, ?, ?)}");

        // 4. Set input parameter
cstmt.setInt(1, 1); // ID = 1

        // 5. Register output parameters
cstmt.registerOutParameter(2, Types.VARCHAR); // Name
cstmt.registerOutParameter(3, Types.INTEGER); // Age

        // 6. Execute stored procedure
cstmt.execute();

        // 7. Retrieve output
        String name = cstmt.getString(2);
int age = cstmt.getInt(3);

System.out.println("Student Name: " + name);
System.out.println("Student Age: " + age);

        // 8. Close connection
cstmt.close();
con.close();

      } catch(Exception e) {
e.printStackTrace();
      }
   }
}
```

5. Input / Output

Input

- ID = 1 (passed as input to stored procedure)

Output

Student Name: Alice
Student Age: 14

6. Note:

- CallableStatement can handle **IN**, **OUT**, and **INOUT** parameters.
- It is mainly used when complex database operations are handled inside stored procedures.
- Example of using **INOUT** parameter:

```
CREATE PROCEDURE IncreaseAge(INOUT sid INT, IN increment INT)
BEGIN
```

```
    UPDATE Students SET Age = Age + increment WHERE ID = sid;
    SELECT Age INTO sid FROM Students WHERE ID = sid;
END;
```

- In Java, register sid as INOUT:

```
cstmt.registerOutParameter(1, Types.INTEGER); // INOUT parameter
```

Difference between CallableStatement and PreparedStatement :

CallableStatement	PreparedStatement
It is used when the stored procedures are to be executed.	It is used when SQL query is to be executed multiple times.
You can pass 3 types of parameter IN, OUT, INOUT.	You can pass any type of parameters at runtime.
Used to execute functions.	Used for the queries which are to be executed multiple times.
Performance is very high.	Performance is better than Statement.
Used to call the stored procedures.	Used to execute dynamic SQL queries.
It extends PreparedStatement interface.	It extends Statement Interface.
No protocol is used for communication.	Protocol is used for communication.

Overview of the Statement PreparedStatementCallableStatement:

Feature	Statement	PreparedStatement	CallableStatement
Purpose	Executes static SQL queries	Executes parameterized queries	Executes stored procedures
SQL injection protection	No	Yes	Yes
Performance	Normal (re-parsed every time)	Faster (precompiled)	Depends (executed on DB side)
Used for	Simple queries	Dynamic queries with parameters	Stored procedures and functions

Table for the PreparedStatementCallableStatementResultSet :

Feature	PreparedStatement	CallableStatement	ResultSet
Main Use	Execute parameterized SQL queries	Execute stored procedures	Hold query results
SQL type	DML (SELECT, INSERT, UPDATE, DELETE)	Stored procedure/function	Output of a query
Parameter Handling	Uses ? placeholders	Supports IN, OUT, INOUT parameters	Access data via column names/index
Compilation	Precompiled (faster)	Precompiled (DB-side procedure)	Not compiled; returned as data
SQL Injection Safe	Yes	Yes	N/A
Return Type	ResultSet or update count	ResultSet or output params	Data from query
Example	SELECT * FROM emp WHERE dept=?	{callgetEmployeeByDept(?)}	Iterating over query results

9.5 SUMMARY

In JDBC, several key objects are used to interact with databases, each serving a distinct purpose. The **Statement** object is used to execute simple SQL queries such as SELECT, INSERT, UPDATE, and DELETE. However, since it directly embeds user inputs into SQL strings, it is vulnerable to SQL injection attacks, making it suitable only for static or simple queries. The **ResultSet** object stores the data retrieved from queries and provides cursor-based navigation to move through rows of results. Depending on its type, it can be forward-only or scrollable, and even updatable, allowing direct modification of data within the result set.

For more secure and efficient database operations, the PreparedStatement object is preferred. It allows parameterized queries, preventing SQL injection and improving performance by precompiling the SQL statement. PreparedStatements are reusable, making them ideal for executing similar queries multiple times with different parameters. They also support batch processing for executing multiple SQL commands efficiently. When working with stored procedures, the CallableStatement object is used. It allows the execution of precompiled database procedures that can accept IN, OUT, and INOUT parameters, offering better performance, encapsulation, and security for complex operations.

Proper resource management is crucial when using these JDBC objects—closing the ResultSet, Statement, and Connection objects ensures system efficiency and prevents memory leaks. Overall, using the right JDBC object based on the operation type—Statement for simple queries, PreparedStatement for dynamic queries, and CallableStatement for stored procedures—enables developers to build robust, secure, and maintainable database applications.

9.6 KEY TERMS

Statement,PreparedStatement,CallableStatement,Result Set, SQL, Database,Connection,Stored Procedure,Parameter,Cursor,executeQuery,executeUpdate.

9.7 SELF-ASSESSMENT QUESTIONS

1. What is the purpose of a Statement in JDBC?
2. How does PreparedStatement differ from Statement?
3. What is a CallableStatement used for?
4. What does a ResultSet represent in JDBC?
5. Name the three types of ResultSet cursors.
6. Which method is used to execute a SELECT query: execute Query() or execute Update()?
7. How do you set values in a PreparedStatement?
8. What is the advantage of using Prepared Statement over Statement?
9. What types of parameters can a Callable Statement handle?
10. Why is it important to close Result Set, Statement, and Connection objects?

9.8 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Mrs. Appikatla Pushpa Latha

LESSON-10
NETWORK PROGRAMMING AND REMOTE METHOD INVOCATION (RMI)

AIM AND OBJECTIVES:

- Understand the importance of networked Java for building distributed applications.
- Learn basic network concepts, including protocols, IP addresses, and ports.
- Gain the ability to look up Internet addresses and differentiate between URLs and URIs**.**
- Explore UDP datagrams and sockets for sending and receiving data over a network.
- Understand Remote Method Invocation (RMI) to enable communication between Java objects across different machines.

STRUCTURE:

10.1 NETWORK PROGRAMMING

- It is about writing programs that can send and receive data over a network (like the internet or a local network).
- **Why needed:** For client-server applications, chatting apps, online games, or distributed systems.
- **Key concepts:**
  - **IP Address & Ports:** Identify devices and communication endpoints.
  - **Sockets:** Points where programs connect and exchange data.
  - **Protocols:** TCP (reliable) and UDP (fast, less reliable).
  - **URLs and URIs:** Ways to locate and access resources on the web.

10.2 WHY NETWORKED JAVA

Networked Java refers to Java programs that communicate over a network, like the Internet or a local network. Java is widely used for network programming because it provides built-in support for networking via packages like java.net and java.rmi.

Reasons to use Networked Java:

1. **Platform Independence:** Java runs on any machine with a JVM, making networked applications portable.
2. **Built-in Networking API:** Java provides classes like Socket, ServerSocket, DatagramSocket, URL, etc., for easy network communication.
3. **Supports Client-Server Architecture:** You can easily build applications where a client requests data from a server.
4. **RMI (Remote Method Invocation):** Java allows calling methods on remote objects, making distributed systems easier to develop.
5. **Secure Networking:** Java provides SSL and security features for safe network communication.

Basic Java Network Programming Example

1. TCP Client-Server Program

Server Program (TCP)

```
import java.io.*;
import java.net.*;

public class TCPServer {
public static void main(String[] args) throws IOException {
ServerSocket server = new ServerSocket(5000);
System.out.println("Server is running and waiting for a client...");

    Socket client = server.accept();
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
PrintWriter out = new PrintWriter(client.getOutputStream(), true);

    String message = in.readLine();
System.out.println("Client says: " + message);
out.println("Hello Client! Message received.");

client.close();
server.close();
  }
}
```

Client Program (TCP)

```
import java.io.*;
import java.net.*;

public class TCPClient {
public static void main(String[] args) throws IOException {
    Socket socket = new Socket("localhost", 5000);
```

```
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

out.println("Hello Server! This is Client.");
      String response = in.readLine();
System.out.println("Server says: " + response);

socket.close();
   }
}
```

Input / Output Example

Client Input:

Hello Server! This is Client.

Server Output:

Server is running and waiting for a client...
Client says: Hello Server! This is Client.

Client Output:

Server says: Hello Client! Message received.

2. UDP Datagram Example

Server (UDP)

```
import java.net.*;

public class UDPServer {
public static void main(String[] args) throws Exception {
DatagramSocketserverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
byte[] sendData;

System.out.println("UDP Server is running...");

DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
      String message = new String(receivePacket.getData(), 0, receivePacket.getLength());
System.out.println("Client says: " + message);

InetAddressclientAddress = receivePacket.getAddress();
intclientPort = receivePacket.getPort();
      String reply = "Message received via UDP!";
sendData = reply.getBytes();

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length,
clientAddress, clientPort);
serverSocket.send(sendPacket);
serverSocket.close();
   }
}
```

Client (UDP)

```
import java.net.*;
```

```
public class UDPClient
 {
public static void main(String[] args) throws Exception
{
DatagramSocketclientSocket = new DatagramSocket();
InetAddressIPAddress = InetAddress.getByName("localhost");

byte[] sendData = "Hello UDP Server!".getBytes();
byte[] receiveData = new byte[1024];

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length, IPAddress,
9876);
clientSocket.send(sendPacket);

DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
      String response = new String(receivePacket.getData(), 0, receivePacket.getLength());
System.out.println("Server says: " + response);

clientSocket.close();
   }
}
```

Input / Output Example
Client Input:
Hello UDP Server!
Server Output:
UDP Server is running...
Client says: Hello UDP Server!
Client Output:
Server says: Message received via UDP!

Note:

- Networked Java allows communication between programs over TCP or UDP.
- RMI allows calling methods on remote objects as if they were local.
- Java's built-in networking libraries make creating client-server or distributed applications easy and platform-independent.

10.3 BASIC NETWORK CONCEPTS

Network programming involves communication between computers over a network. Java provides built-in support via the java.net package.

Key Concepts:

1. **IP Address:**
    - Unique address assigned to each device on a network.
    - Example: 192.168.1.1 (IPv4) or 2001:0db8:85a3::8a2e:0370:7334 (IPv6).

2. **Port Number:**
   - Used to identify a specific process/application on a device.
   - Range: 0–65535 (Ports <1024 are reserved).
3. **Socket:**
   - Endpoint for communication between two machines.
   - Socket class for clients, ServerSocket class for servers.
4. **TCP (Transmission Control Protocol):**
   - Connection-oriented protocol (reliable).
5. **UDP (User Datagram Protocol):**
   - Connectionless protocol (faster but unreliable).
6. **URL and URI:**
   - URL (Uniform Resource Locator) – identifies a resource on the internet.
   - URI (Uniform Resource Identifier) – more general, can be a URL or just a name.

Java Example Programs

1. Finding IP Address

```
import java.net.*;

public class IPAddressExample {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("www.google.com");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
   }
}
```

Output Example:

```
Host Name: www.google.com
IP Address: 142.250.72.196
```

2. Simple TCP Client-Server

Server (TCP)

```
import java.io.*;
import java.net.*;

public class SimpleTCPServer {
public static void main(String[] args) throws IOException {
ServerSocket server = new ServerSocket(6000);
System.out.println("Server waiting for connection...");
      Socket client = server.accept();

BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
PrintWriter out = new PrintWriter(client.getOutputStream(), true);

      String message = in.readLine();
System.out.println("Client says: " + message);
out.println("Message received!");

client.close();
```

```
server.close();
   }
}
```

Client (TCP)

```
import java.io.*;
import java.net.*;

public class SimpleTCPClient {
public static void main(String[] args) throws IOException {
      Socket socket = new Socket("localhost", 6000);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

out.println("Hello Server!");
System.out.println("Server says: " + in.readLine());

socket.close();
   }
}
```

Input / Output Example

Client Input:

Hello Server!

Server Output:

Server waiting for connection...
Client says: Hello Server!

Client Output:

Server says: Message received!

3. URL Example

```
import java.net.*;

public class URLExample {
public static void main(String[] args) throws Exception {
      URL url = new URL("https://www.example.com");
System.out.println("Protocol: " + url.getProtocol());
System.out.println("Host: " + url.getHost());
System.out.println("File: " + url.getFile());
System.out.println("Port: " + url.getPort());
   }
}
```

Output Example:

Protocol: https
Host: www.example.com
File: /
Port: -1
(Port -1 means default port 443 for HTTPS)

Note:

- **IP & Port** identify a machine and application.
- **Sockets** enable communication (TCP for reliability, UDP for speed).
- **URLs/URIs** locate resources on the internet.
- Java's java.net makes network programming straightforward.

10.4 LOOKING UP INTERNET ADDRESSES

Looking up internet addresses in Java involves retrieving IP addressesandhost names for computers or domain names. This is done using the InetAddress class from the java.net package.

Key Concepts:

1. **InetAddress Class:**
   - Represents an IP address (IPv4 or IPv6).
   - Can be used to get host name and host address.
2. **Methods of InetAddress:**
   - getByName(String host) – Returns the IP address of a given host.
   - getHostName() – Returns the host name.
   - getHostAddress() – Returns the IP address as a string.
   - getAllByName(String host) – Returns all IP addresses associated with a host.
3. **DNS Lookup:**
   - Java can resolve host names to IP addresses (forward lookup).
   - Can also perform reverse lookup (IP → Host name).

Java Examples

1. Lookup IP Address of a Domain

```
import java.net.*;

public class InetAddressExample {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("www.google.com");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
   }
}
```

Output Example:

Host Name: www.google.com
IP Address: 142.250.72.196
(IP may vary depending on DNS and location.)

2. Get Local Host Information

```
import java.net.*;

public class LocalHostExample {
public static void main(String[] args) throws Exception {
InetAddress local = InetAddress.getLocalHost();
System.out.println("Local Host Name: " + local.getHostName());
System.out.println("Local IP Address: " + local.getHostAddress());
```

```
    }
}
```

Output Example:

Local Host Name: MyPC
Local IP Address: 192.168.1.5

3. Get All IP Addresses of a Domain

```
import java.net.*;

public class AllIPAddresses
{
public static void main(String[] args) throws Exception
 {
InetAddress[] addresses = InetAddress.getAllByName("www.google.com");
System.out.println("All IP addresses for www.google.com:");
for (InetAddressaddr : addresses)
 {
System.out.println(addr.getHostAddress());
      }
   }
}
```

Output Example:

All IP addresses for www.google.com:
142.250.72.196
142.250.72.228
142.250.72.164
(Multiple IPs are returned because large websites use multiple servers for load balancing.)

4. Reverse DNS Lookup (IP → Hostname)

```
import java.net.*;

public class ReverseLookup {
public static void main(String[] args) throws Exception {
InetAddress address = InetAddress.getByName("8.8.8.8");
System.out.println("Host Name: " + address.getHostName());
System.out.println("IP Address: " + address.getHostAddress());
   }
}
```

Output Example:

Host Name: dns.google
IP Address: 8.8.8.8

Note:

- InetAddress class is used to lookup IP addresses and host names.
- Methods like getByName(), getHostName(), and getAllByName() help in DNS lookups.
- You can perform forward (name → IP) and reverse (IP → name) lookups.
- Useful for networkprogramming, pinging hosts, or validating connectivity.

10.5 URLS AND URIS

Introduction

In network programming, **URLs and URIs** are essential because they specify how to locate resources over a network. Java provides classes in the java.net package to handle them.

- **URI (Uniform Resource Identifier):** Identifies a resource. Can be a URL or URN.
- **URL (Uniform Resource Locator):** Specifies where a resource is and how to access it.

Example:
URL: https://www.example.com:443/index.html?user=123#section1
URI: https://www.example.com/index.html

Components of URL

1. **Protocol / Scheme**: http, https, ftp
2. **Host / Domain**: www.example.com
3. **Port**: Default 80 for HTTP, 443 for HTTPS
4. **Path**: /index.html
5. **Query**: ?user=123
6. **Fragment**: #section1

Java Classes

- **java.net.URI** – Represents a URI, allows parsing and constructing URIs.
- **java.net.URL** – Represents a URL, allows connecting to a resource.
- **java.net.URLConnection** – To read/write data from a URL.

4. Example Programs

Example 1: Parsing a URL

```
import java.net.URL;

public class URLExample {
public static void main(String[] args) {
try {
        URL url = new
URL("https://www.example.com:443/index.html?user=123#section1");

System.out.println("Protocol: " + url.getProtocol());
System.out.println("Host: " + url.getHost());
System.out.println("Port: " + url.getPort());
System.out.println("Path: " + url.getPath());
System.out.println("Query: " + url.getQuery());
System.out.println("Reference: " + url.getRef());
    } catch (Exception e) {
e.printStackTrace();
    }
```

```
    }
}
```

Input: URL is hardcoded.

Output:

```
Protocol: https
Host: www.example.com
Port: 443
Path: /index.html
Query: user=123
Reference: section1
```

Example 2: Parsing a URI

```
importjava.net.URI;

public class URIExample {
public static void main(String[] args) {
try {
        URI uri = new URI("https://www.example.com/index.html?user=123#section1");

System.out.println("Scheme: " + uri.getScheme());
System.out.println("Host: " + uri.getHost());
System.out.println("Port: " + uri.getPort());
System.out.println("Path: " + uri.getPath());
System.out.println("Query: " + uri.getQuery());
System.out.println("Fragment: " + uri.getFragment());
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Output:

```
Scheme: https
Host: www.example.com
Port: -1
Path: /index.html
Query: user=123
Fragment: section1
```

Note: Port -1 indicates default port is used (443 for HTTPS).

Example 3: Reading Data from a URL

```
import java.net.URL;
importjava.io.BufferedReader;
importjava.io.InputStreamReader;

public class URLReadExample {
public static void main(String[] args) {
try {
        URL url = new URL("https://www.example.com");
BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream()));

        String line;
```

```
while ((line = reader.readLine()) != null) {
System.out.println(line);
        }
reader.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Input: URL of a website.
Output: HTML content of the page (depends on the website).
Example snippet:

```
<!doctype html>
<html>
<head>
<title>Example Domain</title>
</head>
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents.</p>
</div>
</body>
</html>
```

5. Key Points

1. URLs are used to connect to resources over the network in Java.
2. URIs are used mainly for identification and parsing.
3. Java provides URL and URI classes for network programming.
4. You can also read content from URLs using URLConnection or openStream().
5. URL/URI handling is a fundamental part of network programming and web applications.

Difference Between URI and URL

Feature	URL (Uniform Resource Locator)	URI (Uniform Resource Identifier)
Definition	A URL is a type of URI that specifies the location of a resource and how to access it.	A URI is a generic identifier of a resource, which may or may not include its location.
Purpose	Used to locate and access resources over the internet.	Used to identify a resource uniquely, without necessarily specifying how to access it.
Components	Typically includes protocol, host, port, path, query, fragment.	Can include scheme, path, query, fragment, but location/access info is optional.
Example	https://www.example.com:443/index.html?user=123#section1	https://www.example.com/index.html#section1 or urn:isbn:0451450523

Feature	URL (Uniform Resource Locator)	URI (Uniform Resource Identifier)
Class in Java	java.net.URL	java.net.URI
Main Use in Networking	Connect to a web resource, read/write data (e.g., HTTP requests).	Parse, manipulate, or compare resource identifiers.

Key Point:

- All URLs are URIs, but not all URIs are URLs.
- URL = URI + access method/location.
- URI = identifier, may not point to an actual resource.

10.6 UDP DATAGRAMS AND SOCKETS

1. What is UDP?

- UDP (User Datagram Protocol) is a connectionless protocol used for sending short messages called datagrams over the network.
- It is faster than TCP because it doesn't establish a connection and has no guarantee of delivery, ordering, or error checking.
- Commonly used in:
  - Video streaming
  - Online gaming
  - DNS queries

2. UDP Concepts

Term	Description
Datagram	A packet of data sent over UDP
DatagramSocket	Socket for sending and receiving datagrams
DatagramPacket	Represents a packet to send or receive data
Port	Identifies the application on a host
Connectionless	No persistent connection between client and server

3. Java Classes for UDP

- java.net.DatagramSocket – Creates a socket to send/receive UDP packets.
- java.net.DatagramPacket – Represents data packets.

4. UDP Server Example

```
import java.net.*;

public class UDPServer {
public static void main(String[] args) {
try {
DatagramSocketserverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
```

```
System.out.println("Server is running... Waiting for client messages.");

while (true) {
DatagramPacketreceivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);

            String message = new String(receivePacket.getData(), 0,
receivePacket.getLength());
System.out.println("Received from client: " + message);

if (message.equalsIgnoreCase("exit")) {
System.out.println("Server exiting...");
break;
            }
          }

serverSocket.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

5. UDP Client Example

```
import java.net.*;
importjava.util.Scanner;

public class UDPClient {
public static void main(String[] args) {
try {
DatagramSocketclientSocket = new DatagramSocket();
InetAddressIPAddress = InetAddress.getByName("localhost");
          Scanner sc = new Scanner(System.in);

System.out.println("Enter messages to send to the server (type 'exit' to quit):");
while (true) {
            String message = sc.nextLine();
byte[] sendData = message.getBytes();

DatagramPacketsendPacket = new DatagramPacket(sendData, sendData.length, IPAddress,
9876);
clientSocket.send(sendPacket);

if (message.equalsIgnoreCase("exit")) {
System.out.println("Client exiting...");
break;
            }
          }
```

```
clientSocket.close();
sc.close();
        } catch (Exception e) {
e.printStackTrace();
        }
    }
}
```

6. Sample Input/Output
Client Input:
Hello Server
How are you?
exit

Server Output:
Server is running... Waiting for client messages.
Received from client: Hello Server
Received from client: How are you?
Received from client: exit
Server exiting...

Client Output:
Enter messages to send to the server (type 'exit' to quit):
Hello Server
How are you?
exit
Client exiting...

7. Key Points

- UDP is faster but unreliable compared to TCP.
- No connection is established; packets may arrive outoforder or getlost.
- Useful for real-timeapplications like streaming, VoIP, and gaming.

10.7 REMOTE METHOD INVOCATION

What is RMI?

- RMI (Remote Method Invocation) is a Java API that allows an object running in one Java virtual machine (JVM) to invoke methods on an object in another JVM.
- RMI abstracts the underlying network communication, allowing remote method calls as if they were local.
- It is part of java.rmi package.

Use Cases:
- Distributed applications
- Client-server architecture
- Remote services

Key Components of RMI

Component	Description
Remote Interface	Declares the methods that can be called remotely
Remote Object	Implements the remote interface and contains the actual business logic
Stub	Client-side proxy for the remote object (generated automatically)
Skeleton	Server-side entity that dispatches client calls to the remote object (Java 2+ not required explicitly)
RMI Registry	Service that maps names to remote objects, allowing clients to look them up

Steps to Create RMI Application

1. Define the Remote Interface **(**extends java.rmi.Remote)
2. Implement the Remote Interface
3. Create and start the RMI Server
4. Bind the remote object in the RMI Registry
5. Create the RMI Client to lookup and invoke methods

Example: Simple RMI – Adding Two Numbers

Step 1: Remote Interface

```
importjava.rmi.Remote;
importjava.rmi.RemoteException;

public interface AddInterface extends Remote {
int add(int a, int b) throws RemoteException;
}
```

Step 2: Remote Object Implementation

```
importjava.rmi.server.UnicastRemoteObject;
importjava.rmi.RemoteException;

public class AddImplementation extends UnicastRemoteObject implements AddInterface {
publicAddImplementation() throws RemoteException {
super();
   }

publicint add(int a, int b) throws RemoteException {
return a + b;
   }
}
```

Step 3: RMI Server

```
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class RMIServer {
public static void main(String[] args) {
try {
```

```
AddImplementationaddObj = new AddImplementation();
        Registry registry = LocateRegistry.createRegistry(1099); // default port
registry.rebind("AddService", addObj);
System.out.println("Server is running...");
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Step 4: RMI Client

```
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class RMIClient {
public static void main(String[] args) {
try {
        Registry registry = LocateRegistry.getRegistry("localhost");
AddInterface stub = (AddInterface) registry.lookup("AddService");

int result = stub.add(10, 20);
System.out.println("Result of addition: " + result);
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

How to Run RMI Application

1. Compile all Java files:
   javac *.java
2. Start the RMI registry:
   rmiregistry

(Keep it running in the background)

3. Start the server:
   javaRMIServer
4. Run the client:
   javaRMIClient

Sample Output

Server Output:
Server is running...
Client Output:
Result of addition: 30

Key Points

- RMI allows transparent communication between JVMs.
- Remote interfaces must extend java.rmi.Remote.
- Remote methods must declare throws RemoteException.
- RMI uses stubs and skeletons (proxy objects) to handle network communication.
- Useful for distributed computing and enterprise applications.

10.8 SUMMARY

Network programming in Java allows applications to communicate over networks by using classes from the **java.net** package. It supports both **TCP** (Transmission Control Protocol) and **UDP** (User Datagram Protocol), catering to different communication needs—TCP for reliable, connection-oriented data transfer, and UDP for faster, connectionless communication. Fundamental concepts such as IP addresses, ports, and sockets are essential in establishing communication between devices. Java provides classes like Socket and ServerSocket for TCP-based communication and DatagramSocket and DatagramPacket for UDP-based data exchange.

Additionally, Java simplifies working with web resources through the **URL** and **URI** classes, enabling developers to connect to web servers, parse links, and retrieve data directly from websites. The InetAddress class assists in resolving hostnames and IP addresses, making DNS lookups straightforward. For applications requiring high-speed, lightweight communication—such as gaming, live streaming, or chat systems—UDP programming is ideal due to its low overhead and faster transmission.

For building distributed applications, Java provides **RMI (Remote Method Invocation)**, which allows objects running in different JVMs (Java Virtual Machines) to communicate seamlessly as if they were local. RMI involves defining remote interfaces, implementing them on the server, and using an RMI registry for client-server interaction. Overall, Java's networking features make it a powerful and platform-independent choice for developing secure, scalable, and cross-platform networked applications.

10.9 KEY TERMS

Socket**,** ServerSocket**,** DatagramSocket**,** IP Address**,** Port Number**,** TCP (Transmission Control Protocol)**,** UDP (User Datagram Protocol)**,** URL (Uniform Resource Locator)**,** InetAddress**,** RMI (Remote Method Invocation).

10.10 SELF-ASSESSMENT QUESTIONS

1. What is network programming and why is it used?
2. What is the purpose of an IP address in network communication?
3. What is the difference between TCP and UDP?
4. Which Java class is used to create a server that listens for client connections?
5. What is the function of the Socket class in Java?
6. What does the InetAddress class do?
7. What is the role of a port number in networking?
8. What does URL stand for, and what is it used for?
9. What is RMI and why is it important in Java network programming?
10. Name two real-world applications that use network programming.

10.11 Further Readings

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*

3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Mrs. Appikatla Pushpa Latha

LESSON-11
INTRODUCTION TO WEB SERVERS AND THE TOMCAT ENVIRONMENT

AIM AND OBJECTIVES:

- Define what a web server is and describe its basic functions.
- Explain the features, architecture, and components of the Apache Tomcat web server.
- Install and configure the Java Software Development Kit (JDK) required for running Tomcat.
- Set up and configure the Apache Tomcat server environment.
- Test and verify the Tomcat installation using the default local host URL.
- Deploy a sample Java web application on Tomcat.
- Identify and troubleshoot common issues related to Tomcat configuration.

STRUCTURE:

Introduction

The web is the foundation of today's online world. Every time a user accesses a website, a web server works behind the scenes to deliver web pages, handle user requests, and manage communication between the client and the application.

In Java-based web applications, one of the most widely used servers is the Apache Tomcat Web Server**,** which provides an efficient and easy-to-use environment for running Java Servlets and Java Server Pages (JSP)**.**

This chapter introduces the concept of web servers, explains the role of the Tomcat server, and provides step-by-step guidance for installing and testing a Tomcat environment.

11.1 WEB SERVERS

11.1.1 Definition

A web server is software (and sometimes hardware) that uses the HTTP (Hypertext Transfer Protocol) to respond to client requests made through a web browser. Its main function is to receive HTTP requests from clients and send HTTP responses containing web pages, images, or data.

11.1.2 Functions of a Web Server

- **Serving Static Content:** Delivers HTML pages, CSS, images, and scripts stored on the server.
- **Dynamic Content Generation:** Executes programs such as Servlets, JSP, or PHP scripts to generate dynamic content.
- **Request Handling:** Manages incoming client requests and sends appropriate responses.
- **Logging and Monitoring:** Keeps records of user activity and server performance.
- **Security Management:** Implements SSL/TLS encryption for secure communication.

11.1.3 Examples of Popular Web Servers

Web Server	**Developer**	**Primary Use**
Apache HTTP Server	Apache Software Foundation	Static & dynamic content hosting
Nginx	Nginx Inc.	High-performance reverse proxy
Microsoft IIS	Microsoft Corporation	Windows-based applications
Apache Tomcat	Apache Software Foundation	Java-based web applications
Jetty	Eclipse Foundation	Lightweight Java web applications

11.2 TOMCAT WEB SERVER

11.2.1 Introduction

Apache Tomcat is an open-source web server and servlet container developed by the Apache Software Foundation. It is used to run Java-based web applications that use technologies such as Servlets, JSP (JavaServer Pages), and WebSocket.

Tomcat implements the Jakarta Servlet and Jakarta Server Pages (JSP) specifications and serves as the runtime environment for executing servlets and rendering dynamic web content.

11.2.2 Key Features

- Supports Servlet and JSP specifications.
- Lightweight, easy to install, and open-source.
- Handles HTTPrequestsand responses efficiently.
- Provides an administrative GUI and management console.
- Integrates easily with IDEs like Eclipse, IntelliJ, or NetBeans.

How Tomcat Works

1. A client (web browser) sends an HTTP request (e.g., form submission).
2. Tomcat receives the request and passes it to the servlet container.
3. The servlet processes the request (e.g., accesses a database, processes data).
4. The servlet generates an HTTP response (usually HTML).
5. Tomcat sends the response back to the client's browser.

11.2.3 Installing Tomcat

1. Download Tomcat from:https://tomcat.apache.org
2. Extract it to a folder (e.g., C:\apache-tomcat-10.1).
3. Set environment variables:
   - JAVA_HOME → JDK installation path
   - CATALINA_HOME → Tomcat folder
4. Run startup.bat (Windows) or startup.sh (Linux/Mac) in the bin folder.
5. Open a browser and go to:
   👉http://localhost:8080
   You'll see the Tomcat welcome page if it's running properly.

11.2.4 Tomcat Directory Structure

Example Servlet Program Using Tomcat
Step 1 – Directory Structure

```
Tomcat
└── webapps
    └── MyApp
        ├── WEB-INF
        │   ├── web.xml
        │   └── classes
        │       └── HelloServlet.class
        └── index.html
```

Step 2 – Servlet Source Code

HelloServlet.java

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

public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Hello from Tomcat Servlet!</h2>");
out.println("<p>This response is generated by a Java servlet running on Tomcat.</p>");
out.println("</body></html>");
   }
}
```

Step 3 – Deployment Descriptor

web.xml (inside WEB-INF)

```
<web-app>
<servlet>
<servlet-name>HelloServlet</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Step 4 – HTML File (Optional)

index.html

```
<html>
<head><title>Welcome Page</title></head>
<body>
<h1>Welcome to My Java Web Application</h1>
<p><a href="hello">Click here to invoke the servlet</a></p>
</body>
</html>
```

Step 5 – Compile and Deploy

1. Compile the servlet:
2. javac -classpath "C:\apache-tomcat-10.1\lib\servlet-api.jar" HelloServlet.java
3. Place HelloServlet.class inside:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\classes
4. Restart Tomcat.

5. Open your browser and go to:
   👉http://localhost:8080/MyApp/hello

Example Input and Output
Input (User Action)
User opens browser and visits:
http://localhost:8080/MyApp/hello

Output (Browser Display)
<html>
<body>
<h2>Hello from Tomcat Servlet!</h2>
This response is generated by a Java servlet running on Tomcat.
</body>
</html>

Advantages of Tomcat

- Easy to set up and lightweight.
- Excellent support for Java Servlets and JSP.
- Open-source and actively maintained.
- Can be embedded in Java applications.
- Integrates well with development tools and build systems like Maven.

Note:

- Apache Tomcat is a popular web server and servlet container for running Java web applications**.**
- It processes HTTP requests and executes servlets to generate dynamic responses.
- Using Tomcat, developers can create and deploy powerful Java-based web systems easily.

11.3 JSDK

11.3.1 Introduction

JSDK (Java Servlet Development Kit) is a toolkit provided by Sun Microsystems (now Oracle) that contains the libraries, classes, and tools required to develop, test, and run Java Servlets.

It was introduced to help developers build dynamic web applications before the release of full enterprise editions like J2EE and later Jakarta EE.

Today, the JSDK's functionality is integrated into modern Java EE / Jakarta EE servers (like Tomcat, GlassFish, and WildFly), but understanding it remains essential for the fundamentals of servlet development.

11.3.2. Purpose of JSDK

JSDK provides:

- APIs to develop Servlets and JSPs.

- Tools for compiling, running, and testing servlets locally.
- The Servlet API classes like javax.servlet and javax.servlet.http.
- A small web server (in older versions) to test servlets.

Modern servlet containers (like Tomcat) already include these libraries, but the concept of JSDK is foundational for understanding servlet development.

11.3.3. Important Packages in JSDK

Package Name	Description
javax.servlet	Contains core classes and interfaces for building servlets.
javax.servlet.http	Contains classes for HTTP-specific functionalities (GET, POST requests, sessions, cookies).

Common Classes/Interfaces:

- Servlet – Basic interface for all servlets.
- GenericServlet – Provides a framework for non-HTTP servlets.
- HttpServlet – Provides methods for handling HTTP requests (doGet(), doPost()).
- ServletRequest, ServletResponse – Represent client request and server response objects.

11.3.4. JSDK Architecture

A servlet works with the help of:

- **Client (Browser):** Sends HTTP requests.
- **Web Server / Servlet Container:** Runs servlets and manages the servlet lifecycle.
- **Servlet:** Processes requests and sends responses.

The JSDK provides the API layer that enables this communication between client and server.

Example Program Using JSDK
HelloServlet.java

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

public class HelloServlet extends HttpServlet {
public void init() throws ServletException {
System.out.println("Servlet Initialized");
   }

public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Welcome to JSDK Servlet Example</h2>");
out.println("<p>This servlet is running using the Servlet API from JSDK.</p>");
```

```
out.println("</body></html>");
   }

public void destroy() {
System.out.println("Servlet Destroyed");
   }
}
```

Deployment Descriptor (web.xml)

```
<web-app>
<servlet>
<servlet-name>HelloServlet</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Steps to Run the Example Using Tomcat (Modern Equivalent of JSDK)

1. Install Apache Tomcat.
2. Save HelloServlet.java in:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF\classes
3. Save web.xml in:
   C:\apache-tomcat-10.1\webapps\MyApp\WEB-INF
4. Compile the servlet:
5. javac -classpath "C:\apache-tomcat-10.1\lib\servlet-api.jar" HelloServlet.java
6. Start Tomcat and open browser:
   ☞http://localhost:8080/MyApp/hello

Example Input and Output
Input (User Action)
User opens:
http://localhost:8080/MyApp/hello
Output (Browser Display)

```
<html>
<body>
<h2>Welcome to JSDK Servlet Example</h2>
<p>This servlet is running using the Servlet API from JSDK.</p>
</body>
</html>
```

Tomcat Console Output
Servlet Initialized
When stopping the server:
Servlet Destroyed
Explanation

- The JSDK API provides the servlet classes used here (HttpServlet, ServletRequest, ServletResponse).
- When the servlet is first requested, the container loads and initializes it by calling init().
- Each browser request calls the **d**oGet**()** method.
- When the server shuts down or redeploys the application, destroy**()** is called.

Modern Equivalent
Today, instead of using the old standalone JSDK, developers use:

- **Apache Tomcat**
- **Jakarta EE (Servlet 5.0 and above)**
- **Maven/Gradle** for dependency management, using:
- <dependency>
- <groupId>jakarta.servlet</groupId>
- <artifactId>jakarta.servlet-api</artifactId>
- <version>5.0.0</version>
- <scope>provided</scope>
- </dependency>

Advantages of JSDK

- Simplifies servlet and JSP development.
- Provides standard API for all servlet containers.
- Promotes platform independence and code reusability.
- Lays the foundation for modern Java web frameworks.
- Allows testing servlets locally before deployment.

Note:

Aspect	**Description**
Full Form	Java Servlet Development Kit
Purpose	Provides API and tools for developing servlets
Main Packages	javax.servlet, javax.servlet.http
Lifecycle Methods	init(), service(), destroy()
Modern Equivalent	Apache Tomcat / Jakarta Servlet API
Output	Dynamic HTML content via Java code

11.3.5 Installing the Java Software Development Kit
What is JDK?

JDK (Java Development Kit) is a software package that provides all the tools required to develop and run Java applications, including:

- JVM (Java Virtual Machine) – executes Java bytecode
- JRE (Java Runtime Environment) – runtime environment for running Java programs
- Compiler (javac) – converts Java source code into bytecode
- Development tools – java, javac, jar, javadoc, etc.

Steps to Install JDK

Step 1: Download JDK

- Visit the official Oracle website: Java SE Downloads
- Select the latest Java SE Development Kit and download for your operating system (Windows, Mac, or Linux).

Step 2: Install JDK

- Run the downloaded installer.
- Follow instructions and choose an installation directory (e.g., C:\Program Files\Java\jdk-21).
- Click **Next** and finish the installation.

Step 3: Set Environment Variables (Windows)

1. Open System Properties → Advanced → Environment Variables
2. Add a new system variable:
3. Variable name: JAVA_HOME
4. Variable value: C:\Program Files\Java\jdk-21
5. Update the PATH variable:
6. %JAVA_HOME%\bin
7. Open Command Prompt and check installation:
8. java -version
9. javac -version

   Example Output:
   java version "21"
   Java(TM) SE Runtime Environment (build 21+35)
   Java HotSpot(TM) 64-Bit Server VM (build 21+35, mixed mode)
   javac 21

Writing and Running a Java Program

Once JDK is installed, you can write and run Java programs using Command Prompt or IDE (Eclipse, IntelliJ, NetBeans).

Step 1 – Create a Java Program

File: HelloWorld.java

```
// A simple Java program to display a message
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, Java is installed successfully!");  } }
```

Step 2 – Compile Java Program

1. Open Command Prompt
2. Navigate to the folder containing HelloWorld.java
3. Compile using javac:

javac HelloWorld.java

- This generates a HelloWorld.class file (Java bytecode) in the same folder.

Step 3 – Run Java Program

javaHelloWorld

Output:

Hello, Java is installed successfully!

Another Example: Add Two Numbers

File: AddNumbers.java

```
importjava.util.Scanner;
public class AddNumbers {
public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
System.out.print("Enter first number: ");
int a = sc.nextInt();
System.out.print("Enter second number: ");
int b = sc.nextInt();
int sum = a + b;
System.out.println("Sum = " + sum);   }}
```

Input (from user):
Enter first number: 10
Enter second number: 20
Output:
Sum = 30

11.4 TOMCAT SERVER & TESTING TOMCAT

What is Tomcat?

Apache Tomcat is a web server and servlet container developed by Apache Software Foundation. It is used to:

- Run Java Servlets and JavaServer Pages (JSP).
- Serve as a platform for Java-based web applications.
- Provide HTTP web server functionality without needing a separate web server like Apache HTTPD.

Key Components:

Component	Description
Catalina	Servlet container
Coyote	HTTP connector
Jasper	JSP engine (converts JSP to Servlet)
Cluster	Handles session replication for high availability

Installing Apache Tomcat

Step 1: Download Tomcat

- Visit: https://tomcat.apache.org
- Choose the latest Tomcat version (e.g., Tomcat 10.x or 9.x).
- Download Core zip or installer for your OS.

Step 2: Install Tomcat

- **Windows**: Extract the zip file to C:\Tomcat
- **Linux/Mac**: Extract to /usr/local/tomcat or preferred location.

Step 3: Set Environment Variables

- CATALINA_HOME = Tomcat installation folder (e.g., C:\Tomcat)
- Add %CATALINA_HOME%\bin to PATH.

Step 4: Start Tomcat

- Windows: Run startup.bat in C:\Tomcat\bin
- Linux/Mac: Run startup.sh in terminal

Test Tomcat:

Open browser and go to:
http://localhost:8080/
You should see the Tomcat Welcome Page.

Deploying JSP Programs in Tomcat

Step 1: Tomcat Directory Structure

C:\Tomcat

```
├── bin/        # Startup and shutdown scripts
├── webapps/    # Place your JSP/Servlet projects here
│   └── ROOT/   # Default web application folder
├── conf/       # Configuration files
├── logs/       # Log files
└── lib/        # Library files
```

- JSP files should go inside webapps/ROOT/ or a subfolder like webapps/MyApp/.

Step 2: Create a Sample JSP Page

File:hello.jsp (inside C:\Tomcat\webapps\ROOT\)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<html>
<head>
<title>Hello JSP</title>
</head>
<body>
<h2>Hello, Tomcat is working perfectly!</h2>
</body>
</html>
```

Step 3: Access JSP in Browser

Open:
http://localhost:8080/hello.jsp

Output:

Hello, Tomcat is working perfectly!

4. Example: JSP with Form Input

File:greet.jsp (inside ROOT/)

```
<%@ page import="java.util.*" %>
<html>
<head><title>Greeting Page</title></head>
<body>
<h2>Enter Your Name</h2>
<form action="greet.jsp" method="post">
<input type="text" name="username">
```

```
<input type="submit" value="Submit">
</form>
<%
      String name = request.getParameter("username");
if(name != null && !name.isEmpty()) {
out.println("<h3>Welcome, " + name + "! Your JSP program is working.</h3>");   }
   %>
</body>
</html>
```

Input (from form): Name: Alice

Output (in browser): Welcome, Alice! Your JSP program is working.

Stopping Tomcat

- Windows: shutdown.bat in C:\Tomcat\bin
- Linux/Mac: shutdown.sh in terminal

Folder Structure Example for JSP Project

```
webapps/
└── MyApp/
    ├── index.jsp
    ├── hello.jsp
    ├── greet.jsp
    └── WEB-INF/
        ├── web.xml
        └── classes/
```

- JSP pages → MyApp/
- Servlets and Java classes → WEB-INF/classes/
- Deployment descriptor → WEB-INF/web.xml

11.5 SUMMARY

The Java Servlet Development Kit (JSDK), developed by Sun Microsystems (now Oracle), provides the essential tools, libraries, and APIs required for building, testing, and deploying Java servlets—programs that run on web servers to handle client requests and generate dynamic responses. Servlets form the foundation of Java-based web applications and are managed by servlet containers like Apache Tomcat, which implement the Servlet and JSP specifications. The servlet lifecycle includes three key phases: initialization using init(), request handling using service() (or doGet()/doPost() for HTTP requests), and cleanup through destroy(). This lifecycle ensures efficient resource management and reliable request processing across multiple clients.

Using the JSDK, developers can compile servlets, deploy them under the WEB-INF/classes directory, and configure them in the web.xml deployment descriptor. Servlets are significantly faster and more scalable than traditional CGI programs, as they support multithreading—allowing multiple client requests to be processed simultaneously within the same server process. The JSDK includes core packages such as javax.servlet and javax.servlet.http, which define essential classes and interfaces like HttpServlet, ServletRequest, and ServletResponse. These enable interaction between web clients and servers, session management, and dynamic HTML generation.

Modern Java web development builds upon the foundation laid by JSDK, often using Apache Tomcat, Jakarta EE, and build tools like Maven or Gradle for streamlined deployment and dependency management. Despite advancements in frameworks, understanding servlets and the JSDK remains essential for grasping the underlying principles of Java web technologies. Together, they provide a robust, scalable, and maintainable framework for creating interactive, platform-independent, and secure web applications that respond dynamically to user input.

11.6 KEY TERMS

Web Server, Apache Tomcat, Servlet Container, JDK (Java Development Kit), JAVA_HOME, Catalina, Coyote, Jasper, Cluster

11.7 SELF-ASSESSMENT QUESTIONS

1. Explain the architecture and working of the Apache Tomcat web server with a neat diagram.
2. Describe the steps involved in installing and configuring the JDK and Tomcat server.
3. Discuss how to test and verify Tomcat installation and deploy a sample JSP application.
4. Compare Apache Tomcat with other popular web servers like Apache HTTP Server and Nginx.
5. Explain the directory structure of Tomcat and describe the purpose of each folder.
6. Explain the purpose of the JAVA_HOME environment variable.

11.8 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*

Mrs. Appikatla Pushpa Latha

LESSON-12

INTRODUCTION TO SERVLETS

AIM AND OBJECTIVES:

- To provide a standardized framework for developing server-side Java programs (Servlets).
- To enable dynamic content generation and client-server communication via HTTP.
- To support session management, cookies, and request-response handling in web applications.
- To ensure platform-independent, reusable, and maintainable web application components.
- To integrate easily with Java APIs like JDBC, RMI, and JSP for building robust web applications.

STRUCTURE:

12.1. INTRODUCTION TO SERVLETS

A **Servlet** is a **server-side Java program** that extends the capabilities of a web server. It dynamically processes client requests and generates responses, typically in the form of **HTML**, **JSON**, or **XML**. Servlets run inside a **Servlet container** (e.g., Apache Tomcat), which manages their lifecycle and provides services such as request handling, session management, and security.

Servlets are an integral part of **Java EE (Jakarta EE)** for developing web applications that follow the **client-server model**. They act as a bridge between a web browser (client) and databases or other server resources.

2. Definition

A **Servlet** is a Java class that runs on a server and responds to client requests by generating dynamic content.

Definition (by Java):
A Servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model.

3. Need for Servlets

Before Servlets, web developers used **CGI (Common Gateway Interface)** scripts (usually in Perl or C) to create dynamic web content. However, CGI had several disadvantages:

- Each client request created a new process — inefficient for large-scale applications.
- Platform-dependent and hard to maintain.
- Slow in performance.

Servlets overcome these limitations by:
- Running within a single process managed by the JVM.
- Using multithreading to handle multiple requests efficiently.
- Being portable, secure, and integrated with Java APIs.

4. Advantages of Servlets

1. **Platform Independent:**
   Written in Java, Servlets are portable across different servers and operating systems.
2. **Efficient and Scalable:**
   Use **multithreading**—each request is handled by a separate thread instead of a new process.
3. **Robust and Secure:**
   Benefit from Java's strong exception handling and security features.
4. **Integration with Java APIs:**
   Can interact easily with JDBC, RMI, EJB, and other APIs.
5. **Persistence:**
   Loaded once and reused for multiple client requests.
6. **Extensibility:**
   Servlets can be easily extended or modified as applications grow.
7. **Easy to Maintain:**
   Since the logic is written in Java, it supports modular, object-oriented design.

5. Servlet Architecture

A Servlet follows the **request-response model**.
The architecture involves:

1. **Client (Browser):** Sends a request (usually an HTTP request).
2. **Web Server / Servlet Container:** (e.g., Apache Tomcat) receives and forwards the request to the Servlet.
3. **Servlet:** Processes the request, interacts with databases or business logic, and generates a response.
4. **Response:** Sent back to the client (typically as HTML or data).

Diagram (Text Representation):

Client (Browser)
|
v
HTTP Request
|
v
Web Server / Servlet Container
|
v
Servlet (Java Class)
|
v
Database / Application Logic
|
v
Response (HTML/JSON)
|
v
Client (Browser)

6. Life Cycle of a Servlet

The **Servlet life cycle** defines the steps from creation to destruction of a Servlet instance. The **javax.servlet.Servlet** interface defines the methods for these stages:

1. init()
- Called only once when the Servlet is first loaded.
- Used for initialization tasks (like loading configuration or setting up database connections).

2. service()
- Called for **each client request**.
- Handles request and response objects.
- Determines the request method (GET, POST, etc.) and processes accordingly.

3. destroy()
- Called when the Servlet is unloaded or server shuts down.
- Used to release resources (like closing database connections).

4. getServletConfig()
- Returns configuration information for the Servlet.

5. getServletInfo()
- Returns information about the Servlet (version, author, etc.).

Life Cycle Diagram (Text Representation):

```
+-----------------+
| Servlet Loaded |
+--------+--------+
         |
       init()
         |
         v
   service() <---- Multiple requests handled here
```

```
        |
        v
    destroy()
        |
        v
Servlet Unloaded
```

7. Types of Servlets

1. **Generic Servlet**
   - Extends javax.servlet.GenericServlet
   - Protocol-independent (can handle any request type)
   - Must override the service() method.
2. **HTTP Servlet**
   - Extends javax.servlet.http.HttpServlet
   - Handles HTTP requests and responses (GET, POST, PUT, DELETE)
   - Commonly used in web applications.

8. Servlet API Packages

1. **javax.servlet**
   - Defines classes and interfaces for basic Servlet functionality.
   - Key interfaces: Servlet, ServletRequest, ServletResponse, ServletConfig, ServletContext.
2. **javax.servlet.http**
   - Provides classes for HTTP-specific functionality.
   - Important classes: HttpServlet, HttpServletRequest, HttpServletResponse, HttpSession, Cookie.

9. Servlet Communication

Servlets communicate using the **request-response mechanism**:

- The client sends a request (HttpServletRequest) to the server.
- The server responds with an output (HttpServletResponse).

Example methods:

- getParameter(String name) — to get form data from a client.
- setContentType("text/html") — to define the response type.
- getWriter() — to send text output to the client.

10. Example: A Simple Servlet

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

public class HelloServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h2>Welcome to Java Servlets!</h2>");
      out.println("</body></html>");
   }
}
```

11. Deployment Descriptor (web.xml)
The web.xml file (located in WEB-INF folder) configures Servlet mappings:

```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Access URL:
http://localhost:8080/YourApp/hello

12. Limitations of Servlets

- Complex to design large-scale HTML pages (since output is generated using Java code).
- Separation of presentation and business logic is difficult.
- JSP (Java Server Pages) was introduced to overcome these limitations.

Note: Servlets are Java programs that execute on the server side to create dynamic web content. They provide a robust, efficient, and secure way to handle client requests compared to older technologies like CGI. By managing requests and responses within a single JVM process, Servlets form the backbone of modern Java web applications and frameworks like JSP and Spring MVC.

12.2 LIFE CYCLE OF A SERVLET

1. Introduction

A **Servlet life cycle** represents the entire proce**ss** by which a Servlet is loaded, initialized, handles client requests, and is finally destroyed by the web container.

This cycle is **managed by the Servlet Container** (for example, **Apache Tomcat**, **Jetty**, or **GlassFish**), which ensures that each Servlet operates efficiently and securely in response to client requests.

Understanding the Servlet life cycle is essential for developing efficient and robust web applications.

2. What is Servlet Life Cycle?
The **Servlet Life Cycle** defines the **stages through which a Servlet passes**, from its creation to its destruction.

A Servlet is:

1. **Loaded** into memory by the container.
2. **Initialized** using the init() method.
3. **Invoked repeatedly** using the service() method to handle client requests.

4. **Destroyed** using the destroy() method when the container shuts down or no longer needs the Servlet.

Each of these stages corresponds to specific methods defined in the javax.servlet.Servlet interface.

3. Servlet Life Cycle Methods

The **javax.servlet.Servlet** interface defines five main methods involved in the life cycle:

Method	Description
init(ServletConfig config)	Called once when the Servlet is first loaded into memory; used for initialization.
service(ServletRequest req, ServletResponse res)	Called for every client request; processes the request and generates the response.
destroy()	Called once when the Servlet is being removed from service; used for cleanup.
getServletConfig()	Returns configuration information about the Servlet.
getServletInfo()	Returns a string with information about the Servlet (author, version, etc.).

4. Servlet Life Cycle Phases

Let's explore each stage in detail:

1. Loading and Instantiation

- The **Servlet container** loads the Servlet class into memory when:
  - The first request for the Servlet is received, or
  - The web.xml file specifies **<load-on-startup>** (to preload the Servlet at server startup).
- Once loaded, the container **creates an instance** of the Servlet class using its no-argument constructor.

2. Initialization Phase – init() Method

- The container calls the init() method **only once** after creating the Servlet instance.
- It is used to **initialize resources** such as:
  - Database connections
  - Reading configuration parameters
  - Setting up log files or network connections

Syntax:

```
public void init(ServletConfig config) throws ServletException {
   // Initialization code
}
```

Example:

```
public void init() throws ServletException {
   System.out.println("Servlet is being initialized");
}
```

After init() is executed successfully, the Servlet is **ready to handle client requests**.

3. Request Handling Phase – service() Method

- The service() method is called **for each client request**.
- It receives two objects:

- ServletRequest — to get input data from the client.
- ServletResponse — to send output data back to the client.

- In the case of **HTTP Servlets**, this method is usually overridden by **doGet()** or **doPost()** depending on the request type.

Syntax:

```
public void service(ServletRequest request, ServletResponse response)
   throws ServletException, IOException {
   // Code to process request and generate response
}
```

Example (for HTTP Servlet):

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<h3>Hello, this is from doGet() method!</h3>");
}
```

4. Destruction Phase – destroy() Method

- The destroy() method is called **once** when:
  - The Servlet container is shutting down, or
  - The Servlet is no longer needed.
- It allows the Servlet to **release resources** such as:
  - Database connections
  - File handles
  - Network sockets

Syntax:

```
public void destroy() {
   // Cleanup code
}
```

Example:

```
public void destroy() {
   System.out.println("Servlet is being destroyed");
}
```

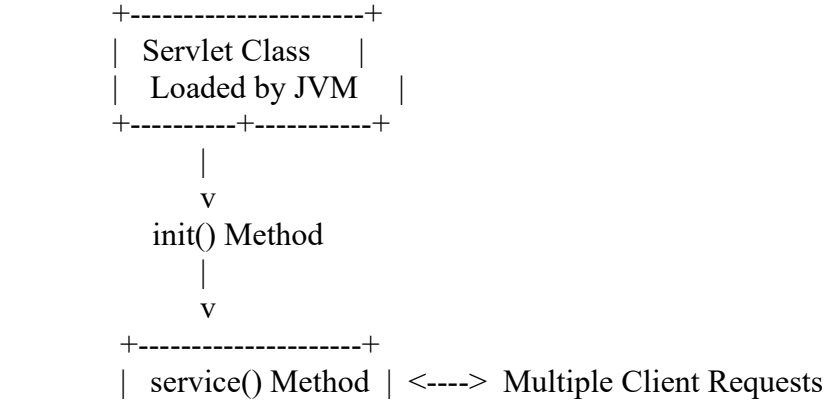
After this stage, the Servlet instance is **eligible for garbage collection**.

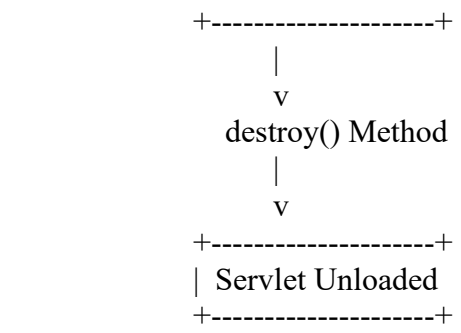
5. Unloading Phase

- After the destroy() method executes, the Servlet instance is **removed from memory**.
- The **JVM's Garbage Collector** reclaims the memory.

5. Servlet Life Cycle Diagram

Here's a textual representation of the life cycle:





6. Example: Servlet Life Cycle Demonstration

Program: LifecycleServlet.java

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

public class LifecycleServlet extends HttpServlet {

   public void init() throws ServletException {
      System.out.println("Servlet is initializing...");
   }

   public void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      System.out.println("Request is being serviced...");
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h2>Servlet Life Cycle Demonstration</h2>");
      out.println("<p>Request processed successfully!</p>");
      out.println("</body></html>");
   }

   public void destroy() {
      System.out.println("Servlet is being destroyed...");
   }
}
```

web.xml Configuration:

```
<web-app>
  <servlet>
    <servlet-name>LifecycleServlet</servlet-name>
    <servlet-class>LifecycleServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>LifecycleServlet</servlet-name>
    <url-pattern>/life</url-pattern>
  </servlet-mapping>
</web-app>
```

Access URL:

http://localhost:8080/MyApp/life

7. Key Interfaces and Classes Involved

Component	Description
Servlet	Main interface defining life cycle methods.
GenericServlet	Abstract class implementing Servlet and ServletConfig.
HttpServlet	Subclass of GenericServlet used for HTTP-specific handling.
ServletConfig	Provides initialization parameters for a Servlet.
ServletContext	Provides global information shared among all Servlets in the application.

8. Key Points to Remember

- init() → Called once when the Servlet is created.
- service() → Called for each client request.
- destroy() → Called once before the Servlet is unloaded.
- Servlet instances are **not recreated for every request** (unlike CGI scripts).
- service() method in HttpServlet automatically dispatches requests to doGet(), doPost(), etc., based on the HTTP method.
-

9. Advantages of Servlet Life Cycle Management

- **Automatic Resource Management:** The container handles initialization and cleanup.
- **Performance:** Servlet instance is created once and reused for multiple requests.
- **Scalability:** Thread-based model supports multiple simultaneous requests.
- **Security:** The container ensures secure handling of requests and responses.

12.3 JSDK (JAVA SERVLET DEVELOPMENT KIT)

Introduction

The **Java Servlet Development Kit (JSDK)** is a set of tools, libraries, and utilities provided by **Sun Microsystems (now Oracle)** to help developers create, test, and deploy **Java Servlets**. It serves as the **development and testing environment** for servlet-based web applications before deploying them on a full-scale web server.

The JSDK provides the **Servlet API** and a **simple web server** for running servlets, allowing developers to learn and experiment with servlet technology easily.

Objectives of JSDK

- To provide a **development environment** for creating and testing servlets.
- To supply **core classes and interfaces** required for servlet programming.
- To enable developers to **compile, load, and run** servlets without requiring a commercial web server.
- To serve as a **reference implementation** for servlet specification.
- To ensure **portability and compatibility** of servlets across all compliant web servers.

Components of JSDK

JSDK mainly consists of the following components:

1. **Servlet API Packages**
   - The JSDK includes the servlet classes and interfaces defined in:
     - javax.servlet
     - javax.servlet.http
   - These provide the building blocks for developing servlets.
2. **Web Server (Servlet Runner)**
   - A simple **built-in web server** is provided with JSDK (often called servletrunner) for testing servlets locally.
   - It can process HTTP requests and responses, enabling developers to test servlet functionality.
3. **Tools and Utilities**
   - JSDK provides command-line tools for:
     - Compiling servlet programs (javac)
     - Running servlet demos
     - Testing servlet deployment using servletrunner
4. **Servlet Examples**
   - Example servlet programs demonstrating various servlet features (handling forms, cookies, sessions, etc.) are included.

Directory Structure of JSDK

After installing JSDK, you will typically find the following structure:

```
JSDK/
├── bin/          → Contains executables like servletrunner
├── lib/          → Contains JAR files (e.g., servlet.jar)
├── docs/         → API documentation and guides
├── examples/     → Sample servlet programs
└── README.txt    → Information about installation and usage
```

Setting up JSDK

To use JSDK, the environment must be configured properly:

1. **Install JDK**
   - Make sure the **Java Development Kit (JDK)** is installed on your system.
   - Set the JAVA_HOME environment variable.
2. **Install JSDK**
   - Download and install JSDK from the official source.
   - Note the installation directory path (e.g., C:\JSDK).
3. **Set Environment Variables**
   - Add the JSDK bin directory to the system PATH.
   - Add servlet.jar (usually in the lib folder) to your Java classpath.
   - set CLASSPATH=%CLASSPATH%;C:\JSDK\lib\servlet.jar;.
4. **Start the Servlet Runner**
   - Navigate to the JSDK bin directory.
   - Run:
   - servletrunner -d C:\JSDK\examples
   - This starts a small HTTP server that can execute servlets.

5. **Access Servlets via Browser**
    - Open a web browser and type:
    - http://localhost:8080/servlet/ServletName

Working of JSDK

When a servlet request is made using JSDK:

1. The **servletrunner** (JSDK's lightweight server) receives the HTTP request.
2. It loads the servlet class (if not already loaded).
3. The servlet's service() method is invoked to process the request.
4. The response is generated and sent back to the browser.
5. The servlet remains in memory for subsequent requests (until the server stops).

Advantages of JSDK

- **Lightweight testing environment** – ideal for development and learning.
- **Portable and platform-independent** – works wherever Java works.
- **Includes complete Servlet API** – supports all core classes and interfaces.
- **Supports rapid prototyping** – no need for complex web server setup.
- **Good for educational and demonstration purposes.**

Limitations of JSDK

- Not suitable for **production environments**.
- Limited **performance and scalability** compared to enterprise servers like Tomcat.
- No advanced features such as JSP support or servlet filters (in older versions).
- Deprecated — replaced by **Tomcat** and **Java EE containers**.

Replacement for JSDK

The **JSDK** has now been replaced by:

- **Apache Tomcat**, which implements the **Servlet and JSP APIs** fully.
- **Jakarta EE (formerly Java EE)**, which provides full enterprise web application support.

Table

Feature	Description
Full Form	Java Servlet Development Kit
Developed By	Sun Microsystems
Purpose	To develop and test servlets
Main Packages	javax.servlet, javax.servlet.http
Includes	Servlet API, servlet runner, examples
Replaced By	Apache Tomcat and Jakarta EE
Use	Educational and small-scale servlet testing

12.4 THE SERVLET API

1. Introduction

The **Servlet API** defines a **set of classes and interfaces** that enable communication between a servlet and its **servlet container (web server)**.

It provides the necessary methods for handling client requests, generating responses, managing sessions, and accessing server information.

The Servlet API is included in the **Java Servlet Development Kit (JSDK)** and now maintained as part of the **Jakarta EE (formerly Java EE)** platform.

The two main packages of the Servlet API are:

1. **javax.servlet**
2. **javax.servlet.http**

2. Purpose of the Servlet API

The main purpose of the Servlet API is to:

- Allow servlets to **interact with web clients and servers**.
- Provide **interfaces and classes** for writing portable and efficient web components.
- Support **HTTP-specific features** such as sessions, cookies, and request/response handling.
- Provide a **standardized programming model** that can run on any servlet container (like Tomcat, Jetty, etc.).

3. The Servlet API Packages

A. javax.servlet Package
This package provides **generic, protocol-independent** classes and interfaces.
It defines the **core features** every servlet must have — regardless of the communication protocol.

Key Interfaces:

1. **Servlet**
    - The root interface for all servlets.
    - Defines the **basic lifecycle methods**:
    - void init(ServletConfig config)
    - void service(ServletRequest req, ServletResponse res)
    - void destroy()
    - ServletConfig getServletConfig()
    - String getServletInfo()
2. **ServletRequest**
    - Encapsulates the client's request information.
    - Provides methods to get form data, attributes, and input streams.
    - String getParameter(String name)
    - Enumeration getParameterNames()
    - Object getAttribute(String name)

3. **ServletResponse**
    - Encapsulates the server's response to the client.
    - Provides methods to send content and control output.
    - PrintWriter getWriter()
    - void setContentType(String type)
4. **ServletConfig**
    - Provides configuration information to a servlet (like init parameters).
    - Common methods:
    - String getInitParameter(String name)
    - ServletContext getServletContext()
5. **ServletContext**
    - Represents the web application environment shared by all servlets.
    - Used for **inter-servlet communication** and accessing application-level resources.
    - String getInitParameter(String name)
    - Object getAttribute(String name)
    - void setAttribute(String name, Object value)

B. javax.servlet.http Package
This package extends the javax.servlet package and provides classes **specific to HTTP protocol**.

It adds more specialized interfaces and classes for handling web-based requests and responses.

Key Interfaces and Classes:

1. **HttpServlet (class)**
    - An abstract class extending GenericServlet.
    - Provides built-in methods to handle HTTP requests:
    - protected void doGet(HttpServletRequest req, HttpServletResponse res)
    - protected void doPost(HttpServletRequest req, HttpServletResponse res)
    - protected void doPut(HttpServletRequest req, HttpServletResponse res)
    - protected void doDelete(HttpServletRequest req, HttpServletResponse res)
    - Developers extend this class and override doGet() or doPost() for web apps.

2. **HttpServletRequest**
    - Extends ServletRequest to include HTTP-specific data.
    - Provides methods to read **headers, cookies, session data**, and **form parameters**.
    - String getHeader(String name)
    - Cookie[] getCookies()
    - HttpSession getSession()
    - String getMethod()
3. **HttpServletResponse**
    - Extends ServletResponse and provides methods to **send HTTP responses**.
    - void addCookie(Cookie cookie)
    - void sendRedirect(String location)
    - void setStatus(int sc)
    - void setHeader(String name, String value)

4. **HttpSession**
   - Provides session management for tracking user data across multiple requests.
   - void setAttribute(String name, Object value)
   - Object getAttribute(String name)
   - void invalidate()

5. **Cookie**
   - Used for managing state between client and server.
   - Cookie c = new Cookie("username", "pushpa");
   - res.addCookie(c);

4. Servlet API Class Hierarchy

Below is a simplified hierarchy of important classes and interfaces:

```
javax.servlet
├── Servlet (interface)
│   └── GenericServlet (abstract class)
│       └── javax.servlet.http.HttpServlet (abstract class)
├── ServletRequest (interface)
│   └── HttpServletRequest (interface)
├── ServletResponse (interface)
│   └── HttpServletResponse (interface)
├── ServletConfig
└── ServletContext
```

5. Lifecycle Methods from Servlet API

Method	Defined In	Description
init()	Servlet	Initializes the servlet instance.
service()	Servlet	Handles client requests.
destroy()	Servlet	Cleans up resources before servlet unloads.
doGet()	HttpServlet	Handles HTTP GET requests.
doPost()	HttpServlet	Handles HTTP POST requests.

6. Example: Simple HttpServlet

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

public class HelloServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h2>Welcome to the Servlet API Example!</h2>");
      out.println("</body></html>");
   }
}
```

7. Advantages of the Servlet API

- Provides a **standardized interface** for server communication.
- Enables **platform-independent** web application development.
- Supports **session management and state tracking**.
- Promotes **reusability and maintainability** through class-based design.
- Integrates easily with **JSP and JDBC** for dynamic web content.

8. Table

Aspect	Description
Purpose	Provides interfaces and classes for servlet–server communication.
Main Packages	javax.servlet, javax.servlet.http
Core Interfaces	Servlet, ServletRequest, ServletResponse, ServletConfig, ServletContext
HTTP Classes	HttpServlet, HttpServletRequest, HttpServletResponse, HttpSession, Cookie
Lifecycle Methods	init(), service(), destroy()
Use	Developing web applications that process requests and generate responses dynamically.

12. 5 SUMMARY

A Servlet is a server-side Java program that extends the functionality of web servers by processing client requests and generating dynamic responses such as HTML, JSON, or XML. Servlets run inside a Servlet container, like Apache Tomcat, which manages their lifecycle, session handling, and security. They replaced CGI scripts by providing a more efficient, scalable, and platform-independent solution. Servlets handle multiple requests using multithreading and are easily integrated with Java APIs like JDBC, RMI, and EJB. The Servlet lifecycle consists of loading, initialization using init(), request handling via service(), and destruction with destroy(). HTTP Servlets extend HttpServlet and override methods like doGet() and doPost() to manage HTTP requests. The Servlet API provides packages javax.servlet and javax.servlet.http to define classes and interfaces for requests, responses, sessions, and cookies. Servlets communicate with clients through the request-response model and can interact with databases or business logic. The Java Servlet Development Kit (JSDK) offered a lightweight environment for learning and testing servlets, though it is now replaced by servers like Tomcat. Key advantages of servlets include performance, security, scalability, and reusability. The Servlet API supports modular and maintainable web applications while ensuring portability across servers. Servlets remain loaded in memory to efficiently handle multiple client requests. They provide a bridge between web browsers and server-side resources. By managing initialization and cleanup automatically, servlets reduce developer effort and improve application reliability. Overall, servlets form the backbone of Java-based web applications and are essential for modern web development.

12.6 KEY TERMS

Servlet, Servlet Container, HTTPServlet, init(), service(), destroy(), ServletRequest, ServletResponse, JSDK,Session.

12.7 SELF-ASSESSMENT QUESTIONS

1. What is a Servlet?
2. Name two advantages of using Servlets over CGI.
3. What method is called when a Servlet is first loaded?
4. Which method handles client requests in a Servlet?
5. What is the purpose of the destroy() method in a Servlet?
6. Differentiate between GenericServlet and HttpServlet.
7. Name the two main packages of the Servlet API.
8. What is the role of HttpSession in Servlets?
9. What is JSDK used for in Servlet development?
10. Explain the request-response model in Servlet architecture.

12.8 FURTHER READINGS

1. Java Servlet Technology Documentation**.** Oracle. Retrieved from https://docs.oracle.com/javaee/7/tutorial/servlets.htm
2. Java Servlet API Specification**.** Oracle. Retrieved from https://jakarta.ee/specifications/servlet/
3. Head First Servlets & JSP by Bryan Basham, Kathy Sierra, Bert Bates. O'Reilly Media.
4. Java Servlet Development Kit (JSDK) Guide**.** Oracle. Retrieved from https://docs.oracle.com/javase/

Mrs. Appikatla Pushpa Latha

LESSON-13
UNDERSTANDING SERVLETS PACKAGE

AIM AND OBJECTIVES:

- Understand the purpose and functionality of the Servlet API in Java web development.
- Learn the core interfaces and classes provided by the javax.servletpackage.
- Explore how to read client request parameters using servlet request methods.
- Understand how to access and use initialization parameters for servlet configuration.
- Study the javax.servlet.httppackage for handling HTTP-specific requests and responses.

STRUCTURE:

13.1 THE JAVAX.SERVLET PACKAGE

1. What is javax.servlet?

javax.servlet is a Java package that provides classes and interfaces for building server-side web application**s** — primarily Servlets and Filters — that run on a web server or application server (like Apache Tomcat, Jetty, GlassFish, etc.).

It is part of Java EE (Jakarta EE**)** and used to handle HTTP requests and responses.

2. Key Components of javax.servlet Package

Component	Description
Servlet Interface	Defines methods all servlets must implement (init(), service(), destroy(), etc.).
GenericServlet Class	Implements Servlet interface; can be extended for non-HTTP protocols.
HttpServlet Class	Extends GenericServlet and adds HTTP-specific methods like doGet() and doPost().
ServletRequest	Represents client request; gives access to parameters, headers, etc.

Component	Description
ServletResponse	Represents response sent back to the client.
ServletConfig	Provides servlet configuration data (init parameters).
ServletContext	Provides information shared among servlets (application-wide context).
RequestDispatcher	Forwards requests to another servlet or resource.

3. Servlet Lifecycle

1. **Loading and Instantiation** → Servlet class is loaded.
2. **Initialization (init())** → Called once.
3. **Request Handling (service(), doGet(), doPost())** → Called for each request.
4. **Destruction (destroy())** → Called once before servlet is destroyed.

4. Example 1 — Basic "Hello World" Servlet

Directory Structure

```
MyServletApp/
├── WEB-INF/
│   ├── web.xml
│   └── classes/
│       └── HelloServlet.class
└── index.html
```

(a) HelloServlet.java

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

public class HelloServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

out.println("<html><body>");
out.println("<h2>Hello, Welcome to the Java Servlet Example!</h2>");
out.println("</body></html>");
   }
}
```

(b) web.xml (Deployment Descriptor)

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>hello</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
<servlet-name>hello</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>

</web-app>
```

(c) index.html

```
<!DOCTYPE html>
<html>
<head><title>Servlet Test</title></head>
<body>
<h1>Click the link to test the servlet</h1>
<a href="hello">Say Hello</a>
</body>
</html>
```

Input

User opens:
http://localhost:8080/MyServletApp/hello

Output (Browser)

```
<html><body>
<h2>Hello, Welcome to the Java Servlet Example!</h2>
</body></html>
```

5. Example 2 — Servlet with Request Parameters (GET/POST)

(a) FormServlet.java

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

public class FormServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");

out.println("<html><body>");
out.println("<h3>Hello, " + name + "! Welcome to Servlets.</h3>");
out.println("</body></html>");
   }
}
```

(b) form.html

```
<!DOCTYPE html>
<html>
<head><title>Form Example</title></head>
<body>
<form action="form" method="post">
      Enter your name: <input type="text" name="username" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

(c) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
<servlet>
<servlet-name>formServlet</servlet-name>
<servlet-class>FormServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>formServlet</servlet-name>
<url-pattern>/form</url-pattern>
</servlet-mapping>
</web-app>
```

Input

User fills out form:
Enter your name: Alice
and clicks “Submit”.

Output (Browser)

```
<html><body>
<h3>Hello, Alice! Welcome to Servlets.</h3>
</body></html>
```

6. Common Interfaces in javax.servlet

Interface	Description
Servlet	Base interface for all servlets.
ServletRequest	Request object (data from client).
ServletResponse	Response object (data to client).
Filter	Used to preprocess or postprocess requests.
RequestDispatcher	Forward/include requests to another resource.
ServletContext	Application-level information.
ServletConfig	Servlet-specific configuration data.

7. Deployment & Execution

1. Compile .java → .class
2. Place .class files in WEB-INF/classes/
3. Add mappings in web.xml
4. Deploy project to server (like Tomcat’s webapps folder)
5. Start Tomcat → Access URL in browser.

8. Example Output

Servlet	Input	Output (Browser)
HelloServlet	URL /hello	"Hello, Welcome to the Java Servlet Example!"
FormServlet	POST form with "Alice"	"Hello, Alice! Welcome to Servlets."

9. Transition to Jakarta Servlet

Since **Jakarta EE 9**, the package name changed from
javax.servlet.* → jakarta.servlet.*
But functionality remains the same.

13.2 READING SERVLET PARAMETERS

1. What Are Servlet Parameters?

Servlet parameters are data sent by a client (browser**)** to the server (servlet**)** through an HTTPrequest.

There are two types of parameters you can read in servlets:

1. **Request Parameters** – Sent by the client through HTML forms, query strings, or links.
2. **Initialization Parameters** – Configured in web.xml for a servlet or application.

2. Methods to Read Request Parameters
From the HttpServletRequest object:

Method	Description
getParameter(String name)	Returns a single parameter value (as String).
getParameterValues(String name)	Returns multiple values (like checkboxes).
getParameterNames()	Returns all parameter names (as Enumeration).
getParameterMap()	Returns all parameters and their values (as a Map).

3. Example 1 — Reading Parameters from an HTML Form (POST Method)
(a) form.html

```
<!DOCTYPE html>
<html>
<head><title>User Form</title></head>
<body>
<h2>User Registration</h2>
<form action="register" method="post">
    Name: <input type="text" name="username" /><br><br>
    Email: <input type="text" name="email" /><br><br>
    Gender:
<input type="radio" name="gender" value="Male"> Male
<input type="radio" name="gender" value="Female"> Female<br><br>
    Hobbies:
<input type="checkbox" name="hobby" value="Reading"> Reading
<input type="checkbox" name="hobby" value="Music"> Music
```

```
<input type="checkbox" name="hobby" value="Sports"> Sports<br><br>
<input type="submit" value="Register" />
</form>
</body>
</html>
```

(b) RegisterServlet.java

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

public class RegisterServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

      // Set response type
response.setContentType("text/html");
PrintWriter out = response.getWriter();

      // Reading parameters from the form
      String name = request.getParameter("username");
      String email = request.getParameter("email");
      String gender = request.getParameter("gender");
String[] hobbies = request.getParameterValues("hobby");

      // HTML Output
out.println("<html><body>");
out.println("<h2>User Registration Details</h2>");
out.println("<p><b>Name:</b> " + name + "</p>");
out.println("<p><b>Email:</b> " + email + "</p>");
out.println("<p><b>Gender:</b> " + gender + "</p>");

if (hobbies != null) {
out.println("<p><b>Hobbies:</b></p><ul>");
for (String h : hobbies) {
out.println("<li>" + h + "</li>");
          }
out.println("</ul>");
      } else {
out.println("<p><b>Hobbies:</b> None selected</p>");
      }

out.println("</body></html>");
   }
}
```

(c) web.xml (Deployment Descriptor)

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
```

```
<servlet-name>register</servlet-name>
<servlet-class>RegisterServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>register</servlet-name>
<url-pattern>/register</url-pattern>
</servlet-mapping>

</web-app>
```

Input (From Browser Form)
Name: Alice Johnson
Email: alice@example.com
Gender: Female
Hobbies: Reading, Music

Output (Browser)

```
<html>
<body>
<h2>User Registration Details</h2>
<p><b>Name:</b> Alice Johnson</p>
<p><b>Email:</b> alice@example.com</p>
<p><b>Gender:</b> Female</p>
<p><b>Hobbies:</b></p>
<ul>
<li>Reading</li>
<li>Music</li>
</ul>
</body>
</html>
```

4. Example 2 — Reading Multiple Parameters Using getParameterNames()

(a) ParameterListServlet.java

```
import java.io.*;
importjava.util.*;
importjavax.servlet.*;
importjavax.servlet.http.*;

public class ParameterListServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      Enumeration<String>paramNames = request.getParameterNames();

out.println("<html><body>");
out.println("<h3>All Parameters Received:</h3>");
```

```
out.println("<ul>");

while (paramNames.hasMoreElements()) {
        String paramName = paramNames.nextElement();
String[] values = request.getParameterValues(paramName);

out.print("<li><b>" + paramName + ":</b> ");
for (inti = 0; i<values.length; i++) {
out.print(values[i]);
if (i<values.length - 1)
out.print(", ");
        }
out.println("</li>");
      }

out.println("</ul></body></html>");
   }
}
```

Input URL

http://localhost:8080/MyApp/params?user=Bob&city=Delhi&hobby=Music&hobby=Cricket

Output (Browser)

```
<html><body>
<h3>All Parameters Received:</h3>
<ul>
<li><b>user:</b> Bob</li>
<li><b>city:</b> Delhi</li>
<li><b>hobby:</b> Music, Cricket</li>
</ul>
</body></html>
```

5. Table

Method	Usage	Example
getParameter("name")	Get single value	String n = req.getParameter("name");
getParameterValues("hobby")	Get multiple values	String[] h = req.getParameterValues("hobby");
getParameterNames()	Get all parameter names	Enumeration e = req.getParameterNames();
getParameterMap()	Get all parameters as map	Map m = req.getParameterMap();

6. Key Points

- Request parameters are **always strings** (even if numbers are entered).
- They can be sent through:
  - HTML forms (GET or POST)
  - Query strings (?param=value)
  - AJAX requests
- You can convert numeric strings to integers with Integer.parseInt() if needed.
- For file uploads, use getPart() from javax.servlet.http.Part (Servlet 3.0+).

13.3 READING INITIALIZATION PARAMETERS

1. What Are Initialization Parameters?

Initialization parameters are configuration values defined in the web.xml file (or via annotations in newer Java EE/Jakarta EE versions).
They are not sent by the client — instead, they are set by the developer to configure servlet behavior, like:

- Database connection info
- File paths
- Application settings

There Are Two Types:

Type	Description	Accessed By
Servlet Initialization Parameters	Specific to a single servlet.	ServletConfig object
Context Initialization Parameters	Shared across the entire web app.	ServletContext object

2. Methods to Read Initialization Parameters

From ServletConfig (for one servlet)

Method	Description
getInitParameter(String name)	Returns the value of a specific parameter.
getInitParameterNames()	Returns all parameter names.

From ServletContext (shared for all servlets)

Method	Description
getInitParameter(String name)	Returns the value of a context-wide parameter.
getInitParameterNames()	Returns all context-wide parameter names.

3. Example 1 — Servlet-Specific Initialization Parameters

(a) DatabaseServlet.java

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

public class DatabaseServlet extends HttpServlet {

   private String dbUrl;
   private String dbUser;
   private String dbPassword;

   public void init() throws ServletException {
      // Get initialization parameters from web.xml
      ServletConfig config = getServletConfig();
      dbUrl = config.getInitParameter("dbURL");
      dbUser = config.getInitParameter("username");
      dbPassword = config.getInitParameter("password");
```

```
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html><body>");
        out.println("<h2>Database Configuration Details</h2>");
        out.println("<p><b>Database URL:</b> " + dbUrl + "</p>");
        out.println("<p><b>Username:</b> " + dbUser + "</p>");
        out.println("<p><b>Password:</b> " + dbPassword + "</p>");
        out.println("</body></html>");
    }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>dbServlet</servlet-name>
<servlet-class>DatabaseServlet</servlet-class>

<!-- Servlet-specific initialization parameters -->
<init-param>
<param-name>dbURL</param-name>
<param-value>jdbc:mysql://localhost:3306/mydb</param-value>
</init-param>
<init-param>
<param-name>username</param-name>
<param-value>admin</param-value>
</init-param>
<init-param>
<param-name>password</param-name>
<param-value>secret123</param-value>
</init-param>
</servlet>

<servlet-mapping>
<servlet-name>dbServlet</servlet-name>
<url-pattern>/dbinfo</url-pattern>
</servlet-mapping>

</web-app>
```

Input (Browser Request)

http://localhost:8080/MyApp/dbinfo

Output (Browser)

```
<html><body>
<h2>Database Configuration Details</h2>
<p><b>Database URL:</b> jdbc:mysql://localhost:3306/mydb</p>
```

```
<p><b>Username:</b> admin</p>
<p><b>Password:</b> secret123</p>
</body></html>
```

4. Example 2 — Application-Wide Initialization Parameters (ServletContext)

These are shared across allservlets in your application.

(a) ConfigServlet.java

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

public class ConfigServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      ServletContext context = getServletContext();
      String company = context.getInitParameter("company");
      String supportEmail = context.getInitParameter("supportEmail");

      out.println("<html><body>");
      out.println("<h2>Application Configuration</h2>");
      out.println("<p><b>Company Name:</b> " + company + "</p>");
      out.println("<p><b>Support Email:</b> " + supportEmail + "</p>");
      out.println("</body></html>");
   }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<!-- Context-wide (application) initialization parameters -->
<context-param>
<param-name>company</param-name>
<param-value>Tech Innovators Pvt. Ltd.</param-value>
</context-param>

<context-param>
<param-name>supportEmail</param-name>
<param-value>support@techinnovators.com</param-value>
</context-param>

<servlet>
<servlet-name>configServlet</servlet-name>
<servlet-class>ConfigServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>configServlet</servlet-name>
```

<url-pattern>/config</url-pattern>
</servlet-mapping>
</web-app>

Input (Browser Request)

http://localhost:8080/MyApp/config

Output (Browser)

<html><body>
<h2>Application Configuration</h2>
Company Name: Tech Innovators Pvt. Ltd.
Support Email: support@techinnovators.com
</body></html>

5. Comparison Table

Type	Defined In	Accessed Using	Scope	Example Usage
Servlet Init Parameter	Inside <servlet> tag in web.xml	ServletConfig	Specific to one servlet	DB username/password
Context Init Parameter	At top level <context-param> in web.xml	ServletContext	Shared among all servlets	Company name, global email, version info

6. Key Points

- init() method is called only once, when the servlet is first loaded.
- Initialization parameters are read-only.
- They are useful for configuration values that might change per deployment (like DB URLs).
- Using ServletContext, all servlets in the web app can share the same parameters.

7. Example Output Summary

Servlet	Input (URL)	Output (Browser)
DatabaseServlet	/dbinfo	Shows servlet-specific DB parameters
ConfigServlet	/config	Shows app-wide company & support info

13.4 THE JAVAX.SERVLET HTTP PACKAGE

1. What Is javax.servlet.http?

The javax.servlet.http package extends the javax.servlet package to support HTTP-specific functionality.

It provides classes and interfaces for handling:

- HTTP requests and responses
- Cookies
- Sessions
- State management
- HTTP methods like GET, POST, PUT, DELETE, etc.

2. Major Classes & Interfaces in javax.servlet.http

Class / Interface	Description
HttpServlet	Base class for creating HTTP servlets. You extend this class to create your own servlet.
HttpServletRequest	Represents the client's HTTP request (headers, parameters, cookies, etc.).
HttpServletResponse	Represents the HTTP response sent to the client.
HttpSession	Used for session tracking (stores data between multiple requests).
Cookie	Represents HTTP cookies for client-side state management.
HttpSessionBindingListener	Notified when objects are bound/unbound to a session.
HttpServletRequestWrapper / HttpServletResponseWrapper	Used to modify requests or responses.

3. The HttpServlet Class

Important Methods

Method	Description
doGet(HttpServletRequest req, HttpServletResponse res)	Handles HTTP GET requests.
doPost(HttpServletRequest req, HttpServletResponse res)	Handles HTTP POST requests.
doPut(), doDelete()	Handle other HTTP methods.
getServletInfo()	Returns info about the servlet.

4. Example 1 — Basic HTTP Servlet Using doGet()

(a) HelloHttpServlet.java

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

public class HelloHttpServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Set the content type
      response.setContentType("text/html");

      // Get output writer
      PrintWriter out = response.getWriter();

      // Write response
      out.println("<html><body>");
      out.println("<h2>Welcome to javax.servlet.http Example!</h2>");
      out.println("<p>This response is generated by doGet() method.</p>");
      out.println("</body></html>");
   }
}
```

(b) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">

<servlet>
<servlet-name>helloHttp</servlet-name>
<servlet-class>HelloHttpServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>helloHttp</servlet-name>
<url-pattern>/helloHttp</url-pattern>
</servlet-mapping>

</web-app>
```

Input (Browser URL)

http://localhost:8080/MyApp/helloHttp

Output (Browser)

```
<html><body>
<h2>Welcome to javax.servlet.http Example!</h2>
<p>This response is generated by doGet() method.</p>
</body></html>
```

5. Example 2 — Reading Request Data Using HttpServletRequest

(a) UserInfoServlet.java

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

public class UserInfoServlet extends HttpServlet {

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      // Set content type
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Read parameters from form
      String name = request.getParameter("username");
      String city = request.getParameter("city");

      out.println("<html><body>");
      out.println("<h2>User Information</h2>");
      out.println("<p><b>Name:</b> " + name + "</p>");
      out.println("<p><b>City:</b> " + city + "</p>");
      out.println("</body></html>");
   }
}
```

(b) form.html

```
<!DOCTYPE html>
<html>
<head><title>User Info</title></head>
```

```
<body>
<form action="userinfo" method="post">
      Name: <input type="text" name="username"><br><br>
      City: <input type="text" name="city"><br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

(c) web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
<servlet>
<servlet-name>userinfo</servlet-name>
<servlet-class>UserInfoServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>userinfo</servlet-name>
<url-pattern>/userinfo</url-pattern>
</servlet-mapping>
</web-app>
```

Input

User submits the form:
Name: Alice
City: New York

Output (Browser)

```
<html><body>
<h2>User Information</h2>
<p><b>Name:</b> Alice</p>
<p><b>City:</b> New York</p>
</body></html>
```

6. Example 3 — Using Cookies (javax.servlet.http.Cookie)

(a) CookieServlet.java

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

public class CookieServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Create a new cookie
      Cookie userCookie = new Cookie("username", "JohnDoe");
      userCookie.setMaxAge(60 * 60); // 1 hour
      response.addCookie(userCookie);

      out.println("<html><body>");
```

```
        out.println("<h3>Cookie has been set!</h3>");
        out.println("</body></html>");
    }
}
```

Input

http://localhost:8080/MyApp/cookie

Output (Browser)

```
<html><body>
<h3>Cookie has been set!</h3>
</body></html>
```

(Browser stores cookie username=JohnDoe)

7. Example 4 — Managing Sessions (HttpSession)

(a) SessionServlet.java

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

public class SessionServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session = request.getSession();
        Integer count = (Integer) session.getAttribute("count");

        if (count == null) {
            count = 1;
        } else {
            count++;
        }

        session.setAttribute("count", count);

        out.println("<html><body>");
        out.println("<h2>Session Example</h2>");
        out.println("<p>Session ID: " + session.getId() + "</p>");
        out.println("<p>You have visited this page " + count + " times.</p>");
        out.println("</body></html>");
    }
}
```

Input (Repeated Browser Visits)

http://localhost:8080/MyApp/session

Output (Browser)

First visit:
<h2>Session Example</h2>
Session ID: A12F5C9D2345E...
You have visited this page 1 times.
Second visit:
<h2>Session Example</h2>
Session ID: A12F5C9D2345E...
You have visited this page 2 times.

8. Summary of javax.servlet.http Components

Class / Interface	Purpose	Example Usage
HttpServlet	Base class for HTTP servlets	Extend to create custom servlets
HttpServletRequest	Access client request data	req.getParameter("name")
HttpServletResponse	Send data to client	res.getWriter().println()
HttpSession	Store user session info	session.setAttribute()
Cookie	Manage client-side state	response.addCookie()

9. Key Points

- All HTTP servlets must extend HttpServlet.
- doGet() and doPost() handle client requests.
- HttpServletRequest gives you:
  - Parameters
  - Headers
  - Cookies
  - Session
- HttpServletResponse allows sending HTML, JSON, or file data back.
- Cookies and sessions are used for state management across multiple requests.

10. Output Summary

Servlet	Input (URL or Form)	Output (Browser)
HelloHttpServlet	/helloHttp	Static HTML greeting
UserInfoServlet	POST form	Displays submitted name & city
CookieServlet	/cookie	Sets a cookie
SessionServlet	/session	Shows visit count per session

13.5 SUMMARY

The Servlet API is a core part of Java EE (Jakarta EE) used to create dynamic web applications that process client requests and generate responses. It provides two main packages — **b** for generic servlet functionality and javax.servlet.http for HTTP-specific features. The Servlet interface defines the basic structure and lifecycle of a servlet, including methods like init(), service(), and destroy(). Servlets operate within a servlet container such as Apache Tomcat, which manages their lifecycle and communication with clients. Using

ServletRequest and **S**ervletResponse, developers can read client data and send HTML or other types of output to the browser.

The javax.servletpackage includes key components like ServletConfig for servlet-specific configuration and ServletContext for application-wide settings. It also supports forwarding requests using RequestDispatcher and filtering through the Filter interface. Servlets can read request parameters from forms using methods like getParameter(), getParameterValues(), and getParameterNames(), allowing them to process user input efficiently. Additionally, initialization parameters defined in web.xml provide configurable values for servlets or the entire application through ServletConfig and ServletContext.

The javax.servlet.httppackage extends this functionality to handle HTTP requests and responses. It introduces classes such as HttpServlet, HttpServletRequest, and HttpServletResponse, which manage web-based interactions. It also provides HttpSession for maintaining user sessions and Cookie for client-side state tracking. Methods like doGet() and doPost() are used to handle different HTTP request types. Together, these APIs enable Java developers to build secure, scalable, and interactive web applications capable of handling sessions, cookies, and dynamic content effectively.

13.6 KEY TERMS

Servlet, ServletAPI, ServletRequest, ServletResponse, ServletConfig, ServletContext, HttpServlet, HttpSession, Cookie, Initialization Parameters.

13.7 SELF-ASSESSMENT QUESTIONS

1. What is a Servlet?
2. Which interface do all servlets implement?
3. What is the purpose of the ServletRequest object?
4. What does the ServletResponse object do?
5. How is ServletConfig different from ServletContext?
6. What class do most HTTP servlets extend?
7. How can you store data for a user session?
8. What is a Cookie used for?
9. How do you pass initialization parameters to a servlet?
10. Which method is called first when a servlet is loaded?

13.8 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition by Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and Seppe vanden Broucke. Wiley.
3. Java Programming with Oracle JDBC by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBC by Gregory Brill. Sybex.

Dr. U. Surya Kameswari

LESSON-14

COOKIES AND SECURITY IN SERVLETS

AIM AND OBJECTIVES:

- Reading request and initialization parameters
- Generating HTTP responses
- Using cookies and session tracking
- Introduction to security concerns in web applications

STRUCTURE:

14.1 Handling Http Request & Responses

1. Introduction

In web applications, HTTP (HyperText Transfer Protocol) is the foundation of communication between a client (like a web browser) and a server.
When a client sends an HTTP request to a server, the server processes it and sends back an HTTP response.

Java provides several APIs and frameworks to handle HTTP requests and responses, most commonly through Servlets and JSP (JavaServer Pages).

2. HTTP Request

An HTTP request is sent by a client to request data or perform an action on the server.
It consists of:

- **Request Line** (method, URL, version)
- **Headers** (metadata like Content-Type, User-Agent, etc.)
- **Body** (optional; contains data for POST/PUT methods)
-

Common HTTP Methods:

Method	Description
GET	Requests data from the server
POST	Sends data to the server for processing

Method	Description
PUT	Updates an existing resource
DELETE	Deletes a resource
HEAD	Retrieves headers only
OPTIONS	Describes communication options

3. HTTP Response

The HTTP response is sent by the server to the client after processing the request.
It includes:

- **Status Line** (protocol, status code, message)
- **Headers** (metadata such as content type, length, etc.)
- **Body** (data sent to the client)

Common HTTP Status Codes:

Code	Message	Description
200	OK	Request successful
404	Not Found	Resource not found
500	Internal Server Error	Server encountered an error
302	Found	Redirection
403	Forbidden	Access denied

4. Handling HTTP Requests and Responses using Servlets

Servlets are Java programs that run on a server and handle requests/responses dynamically.
Example 1: Handling GET Request
File: GetExampleServlet.java

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

public class GetExampleServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");

out.println("<html><body>");
out.println("<h2>Welcome, " + name + "!</h2>");
out.println("<p>This is a GET request example.</p>");
out.println("</body></html>");
   }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="GetExampleServlet" method="get">
   Enter your name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Input:

username = ABC

Output (Browser):

Welcome, ABC!
This is a GET request example.

Example 2: Handling POST Request

File: PostExampleServlet.java

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

public class PostExampleServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String user = request.getParameter("user");
      String email = request.getParameter("email");

out.println("<html><body>");
out.println("<h3>User Registration Details</h3>");
out.println("<p>Username: " + user + "</p>");
out.println("<p>Email: " + email + "</p>");
out.println("</body></html>");
   }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="PostExampleServlet" method="post">
   Username: <input type="text" name="user"><br>
   Email: <input type="text" name="email"><br>
<input type="submit" value="Register">
</form>
</body>
</html>
```

Input:
user = Rani
email = rani@example.com
Output (Browser):
User Registration Details
Username: Rani
Email: rani@example.com

5. Accessing Request Data
You can access information from the request using the following methods:

Method	Description
getParameter(String name)	Returns the value of a form parameter
getHeader(String name)	Returns the value of a request header
getMethod()	Returns the HTTP method (GET/POST)
getRequestURI()	Returns the requested URI
getRemoteAddr()	Returns the IP address of the client

6. Setting Response Data
You can control what the server sends back to the client using:

Method	Description
setContentType(String type)	Sets response MIME type
setStatus(intsc)	Sets the HTTP status code
addHeader(String name, String value)	Adds a header to the response
getWriter()	Returns a writer to output response text

7. Example: Using Both GET and POST
File: RequestResponseServlet.java

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

public class RequestResponseServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
processRequest(request, response, "GET");
   }

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
processRequest(request, response, "POST");
   }

private void processRequest(HttpServletRequest request, HttpServletResponse response,
String method)
throwsIOException {
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
        String name = request.getParameter("name");

out.println("<html><body>");
out.println("<h3>Request Method: " + method + "</h3>");
out.println("<p>Hello, " + name + "!</p>");
out.println("</body></html>");
    }
}
```

HTML Form:

```
<!DOCTYPE html>
<html>
<body>
<form action="RequestResponseServlet" method="post">
   Name: <input type="text" name="name"><br>
<input type="submit" value="Send Request">
</form>
</body>
</html>
```

Input:

name = ABC123

Output:

Request Method: POST
Hello, ABC123!

8. Table

Concept	Description
HTTP Request	Data sent from client to server
HTTP Response	Data sent from server to client
GET method	Used to retrieve data
POST method	Used to send data securely
Servlet	Java class to handle request/response dynamically
Response writer	Used to generate HTML or data back to client

14.2 USING COOKIES-SESSION TRACKING

1. Introduction

In web technologies, HTTP is a stateless protocol, meaning the server does not remember any informationabout users between requests.To maintain information (like login details, user preferences, or shopping cart items) across multiple requests, SessionTracking is used.

2. What is Session Tracking?

Session Tracking is the process of maintaining user state and data across multiple requests between a client (browser) and a server.

When a user visits a website, a session begins. During this session, the server can store user-specific information and recall it in subsequent interactions.

3. Need for Session Tracking

HTTP is stateless, so:

- Each request from the browser is treated as independent.
- The server cannot identify whether two requests came from the same user.

To overcome this, web applications use session tracking mechanisms to identify returningusers and retain information during their visit.

4. Session Tracking Techniques

Technique	Description
Cookies	Information stored on the client side (browser)
Hidden Form Fields	Hidden input fields within HTML forms
URL Rewriting	Adding session data in the URL
HttpSession	Server-side session object managed by servlet container

Part A: Using Cookies

5. What is a Cookie?

A Cookie is a small piece of text data stored by the browser on the client'scomputer. It helps the server recognize users during subsequent requests.

Key Properties of Cookies:

- Stored as key–value pairs.
- Sent automatically with each request to the same domain.
- Can have an expiration time (persistent or session cookies).

6. Types of Cookies

Type	Description
Session Cookie	Temporary; deleted when browser closes
Persistent Cookie	Stored on disk until expiry date or manually deleted

7. Example 1 – Creating and Reading Cookies

HTML Form: cookie_form.html

```
<!DOCTYPE html>
<html>
<head>
<title>Cookie Example</title>
</head>
<body>
<h3>Enter Your Name</h3>
<form action="SetCookieServlet" method="post">
    Name: <input type="text" name="username"><br><br>
<input type="submit" value="Set Cookie">
```

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

Servlet 1: SetCookieServlet.java

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

public class SetCookieServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {

response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String name = request.getParameter("username");
      Cookie cookie = new Cookie("user", name);
response.addCookie(cookie);

out.println("<html><body>");
out.println("<h3>Cookie has been set successfully!</h3>");
out.println("<a href='GetCookieServlet'>Click here to read the cookie</a>");
out.println("</body></html>");
   }
}
```

Servlet 2: GetCookieServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class GetCookieServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
Cookie[] cookies = request.getCookies();
      String userName = "Guest";
if (cookies != null) {
for (Cookie c : cookies) {
if (c.getName().equals("user")) {
userName = c.getValue();
            }
         }
      }

out.println("<html><body>");
out.println("<h2>Welcome Back, " + userName + "!</h2>");
out.println("</body></html>");
   }
```

}

Sample Input:
Name = ABC

Output:
1. After submitting form:
Cookie has been set successfully!
Click here to read the cookie
2. After clicking the link:
Welcome Back, ABC!

8. Cookie Methods

Method	Description
Cookie(String name, String value)	Creates a cookie
getName()	Returns cookie name
getValue()	Returns cookie value
setMaxAge(int expiry)	Sets cookie lifetime (in seconds)
response.addCookie(Cookie c)	Adds cookie to response
request.getCookies()	Returns cookies sent by client

Part B: Using HttpSession

9. What is an HttpSession?
HttpSession is a server-side object that stores user-specific information. It automatically assigns a unique Session ID to each client.
Unlike cookies, session data is stored on the server, making it more secure.

10. Session Lifecycle
1. Session Created — when a user first accesses the servlet.
2. Data Stored — attributes are added using setAttribute().
3. Data Accessed — using getAttribute().
4. Session Expired — automatically after inactivity (default 30 mins).

11. Example 2 – Using HttpSession
HTML Form: session_form.html

```
<!DOCTYPE html>
<html>
<head>
<title>Session Tracking Example</title>
</head>
<body>
<h3>User Login</h3>
<form action="SessionServlet1" method="post">
      Name: <input type="text" name="username"><br><br>
<input type="submit" value="Login">
</form>
</body>
</html>
```

Servlet 1: SessionServlet1.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class SessionServlet1 extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
      String name = request.getParameter("username");
HttpSession session = request.getSession();
session.setAttribute("user", name);
out.println("<html><body>");
out.println("<h3>Welcome, " + name + "!</h3>");
out.println("<a href='SessionServlet2'>Go to next page</a>");
out.println("</body></html>");
   }
}
```

Servlet 2: SessionServlet2.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class SessionServlet2 extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
HttpSession session = request.getSession(false); // don't create a new one
      String userName = (String) session.getAttribute("user");
out.println("<html><body>");
out.println("<h2>Hello again, " + userName + "!</h2>");
out.println("<p>Your session is still active.</p>");
out.println("</body></html>");
   }
}
```

Sample Input:
username = Rani

Output:
Page 1:
Welcome, Rani!
Go to next page
Page 2 (after clicking link):
Hello again, Rani!
Your session is still active.

12. HttpSession Methods

Method	Description
request.getSession()	Creates a new session if not exists
getId()	Returns session ID
getCreationTime()	Returns time of creation
setAttribute(String name, Object value)	Stores value in session
getAttribute(String name)	Retrieves stored value
invalidate()	Terminates session

13. Difference Between Cookies and Sessions

Feature	Cookies	HttpSession
Storage	Client-side	Server-side
Security	Less secure	More secure
Capacity	Limited (~4KB)	Larger (server memory)
Lifetime	Depends on expiry	Until session timeout
Usage	Lightweight info	Sensitive user data

14. Example 3 – Using Cookies + Session Together

In real-world applications, cookies often store the Session ID, while the server uses that ID to retrieve session data.
For example:

- Browser stores a cookie named JSESSIONID.
- Server uses this ID to map user data stored in the session.

15. Table

Concept	Description
Cookie	Small client-side storage mechanism
Session	Server-side user tracking mechanism
Session ID	Unique identifier for each user
Session Tracking	Maintaining user data across requests
Servlet API	Provides HttpSession and Cookie classes

16. Best Practices

- Use HttpSession for sensitive data.
- Set appropriate cookie expiration and security flags.
- Invalidate sessions on logout.
- Avoid storing large data in sessions.
- Encrypt cookie values if used for authentication.

17. Output Snapshot
Page 1:
Welcome, ABC!
Cookie has been set successfully!
Page 2:
Welcome Back, ABC!
Your session is still active.

18. Note:
Session tracking through Cookies and HttpSession is vital in building dynamic web applications that remember users and personalize content.
They form the core of state management in web technologies, enabling features like:

- User login systems
- Shopping carts
- Personalized dashboards

14.3 SECURITY ISSUES

1. **Introduction**
2.

In Web Technologies**,** security refers to the protection of data and resources from unauthorized access, modification, or destruction.Since web applications are accessed over the internet, they are vulnerable to a wide range of security threats.
A secure web application ensures:

- **Confidentiality** – data is accessible only to authorized users
- **Integrity** – data cannot be altered by unauthorized users
- **Availability** – services are always available to legitimate users
- **Authentication** – verifies user identity
- **Authorization** – grants appropriate access based on privileges

2. Why Web Security Is Important
Web applications often handle sensitive data such as:

- Login credentials
- Banking and financial data
- Personal information (email, address, contact numbers)

If not properly protected, attackers can exploit vulnerabilities to:

- Steal data
- Alter website content
- Impersonate users
- Gain administrative control of systems

3. Common Security Threats in Web Applications

Threat	Description	Example
Cross-Site Scripting (XSS)	Inserting malicious scripts into webpages	Attacker injects JavaScript into form fields
SQL Injection	Injecting SQL commands to manipulate databases	OR '1'='1' in login fields
Cross-Site Request	Forcing a logged-in user to	Attacker uses hidden forms to

Threat	Description	Example
Forgery (CSRF)	perform unwanted actions	make a user send a request
Session Hijacking	Stealing valid session ID to impersonate users	Capturing cookies to access user account
Broken Authentication	Weak login or password handling	No password encryption or reuse
Data Exposure	Sending sensitive data without encryption	Transmitting credentials over HTTP instead of HTTPS
Denial of Service (DoS)	Overloading a server to crash it	Flooding the server with fake requests

4. Major Security Issues Explained

4.1. SQL Injection

Description:

Occurs when unvalidated input is directly used in SQL queries, allowing attackers to manipulate database operations.

Vulnerable Example:

```
String user = request.getParameter("username");
String pass = request.getParameter("password");

Statement stmt = conn.createStatement();
String query = "SELECT * FROM users WHERE username='" + user + "' AND password='"
+ pass + "'";
ResultSetrs = stmt.executeQuery(query);
```

If input:

username = admin
password = ' OR '1'='1

Then query becomes:

SELECT * FROM users WHERE username='admin' AND password='' OR '1'='1'
This condition always returns **true**, allowing unauthorized access.

Secure Example (Using PreparedStatement):

```
String user = request.getParameter("username");
String pass = request.getParameter("password");
PreparedStatementps = conn.prepareStatement(
   "SELECT * FROM users WHERE username=? AND password=?");
ps.setString(1, user);
ps.setString(2, pass);
ResultSetrs = ps.executeQuery();
```

Prevents SQL Injection by treating user input as data, not code.

Sample Input:

username = admin
password = ' OR '1'='1

Output:
Login failed. Invalid credentials.

4.2. Cross-Site Scripting (XSS)

Description:
XSS allows attackers to inject malicious JavaScript into web pages viewed by other users.
Vulnerable Example:
```
String name = request.getParameter("username");
out.println("<h3>Welcome " + name + "</h3>");
```
Input:
```
<script>alert('Hacked!');</script>
```
Output on browser:
- A popup appears: Hacked!

Secure Example:
Use HTML encoding to sanitize user input.
```
String name = request.getParameter("username");
name = name.replaceAll("<", "&lt;").replaceAll(">", "&gt;");
out.println("<h3>Welcome " + name + "</h3>");
```
Input:
```
<script>alert('Hacked!');</script>
```
Output:
Welcome <script>alert('Hacked!');</script>
(Script is displayed as text, not executed)

4.3. Session Hijacking
Description:
Session Hijacking occurs when an attacker steals a valid session ID (stored in cookies) to impersonate a legitimate user.

Prevention Techniques:

- Always use HTTPS
- Regenerate Session ID on login
- Set cookies as HttpOnly and Secure
- Invalidate sessions on logout

Example:
```
HttpSession session = request.getSession();
session.setAttribute("user", username);
// Set secure cookie
Cookie c = new Cookie("JSESSIONID", session.getId());
c.setHttpOnly(true);
c.setSecure(true);
response.addCookie(c);
```

4.4. Cross-Site Request Forgery (CSRF)

Description:
An attacker tricks a logged-in user into performing actions they didn't intend (e.g., submitting a form).
Prevention:

- Use a CSRF token with each form submission.

Example:
In form:

```
<form action="TransferServlet" method="post">
<input type="hidden" name="csrfToken" value="${token}">
   Amount: <input type="text" name="amount">
<input type="submit" value="Transfer">
</form>
```

In servlet:

```
String token = (String) session.getAttribute("csrfToken");
String formToken = request.getParameter("csrfToken");
if (token != null &&token.equals(formToken)) {
out.println("Transaction Successful");
} else {
out.println("Security Error: Invalid CSRF Token");
}
```

4.5. Data Encryption

Description:
Sensitive data (like passwords) should never be stored or transmitted as plain text**.**
Example:

```
importjava.security.MessageDigest;

String password = request.getParameter("password");
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] hash = md.digest(password.getBytes());
String encrypted = new String(hash);
```

Output (hashed password):
E99A18C428CB38D5F260853678922E03ABD8334A

5. Example Program – Secure Login Using PreparedStatement and Session

HTML Form (login.html)

```
<!DOCTYPE html>
<html>
<head><title>Secure Login</title></head>
<body>
<h3>User Login</h3>
<form action="LoginServlet" method="post">
      Username: <input type="text" name="username"><br><br>
      Password: <input type="password" name="password"><br><br>
<input type="submit" value="Login">
</form>
```

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

Servlet: LoginServlet.java

```
import java.io.*;
import java.sql.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class LoginServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();

      String uname = request.getParameter("username");
      String pass = request.getParameter("password");

try {
Class.forName("com.mysql.jdbc.Driver");
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/webdb", "root", "admin");

PreparedStatementps = con.prepareStatement(
              "SELECT * FROM users WHERE username=? AND password=?");
ps.setString(1, uname);
ps.setString(2, pass);

ResultSetrs = ps.executeQuery();

if (rs.next()) {
HttpSession session = request.getSession();
session.setAttribute("user", uname);
out.println("<h3>Welcome, " + uname + "!</h3>");
         } else {
out.println("<h3>Invalid username or password</h3>");
         }
con.close();
      } catch (Exception e) {
e.printStackTrace();
      }
   }
}
```

Sample Input:

username = Rani
password = rani123

Output:

Welcome, Rani!
If invalid credentials:
Invalid username or password

6. Best Security Practices

- Always use PreparedStatement to prevent SQL Injection.
- Validate and sanitize all userinputs.
- Use HTTPS for secure data transmission.
- Implement strongauthentication (use hashed passwords).
- Regenerate session IDs after login and invalidateon logout**.**
- Use HttpOnly and Secure cookies.
- Apply CSRFtokens in forms.
- Never expose sensitive data in URLs.
- Keep all frameworks and servers up to date**.**
- Use exception handling to avoid leaking system details.

7. Table

Security Issue	Description	Prevention Technique
SQL Injection	Malicious SQL statements	Use PreparedStatement
XSS	JavaScript injection	Sanitize HTML input
CSRF	Unauthorized actions	Use CSRF tokens
Session Hijacking	Stealing session ID	Secure cookies & HTTPS
Data Exposure	Plain text data	Encrypt sensitive data

8. Note:

Security in web applications is not optional — it's essential.
By understanding and implementing these measures, developers can protect users from:

- Data theft
- Unauthorized access
- Financial loss
- System compromise

Secure coding, input validation, and encryption are the pillarsof safe web development**.**

14.4 SUMMARY

Web applications communicate through the HTTP protocol, where clients send requests and servers respond with appropriate data. In Java, this interaction is primarily managed using Servlets and JSP (JavaServer Pages). Common HTTP methods include GET for retrieving information and POST for securely sending data to the server. Each server response contains an HTTP status code, such as 200 (OK) for successful operations or 404 (Not Found) for missing resources. Because HTTP is a stateless protocol, it does not retain user information between requests. To overcome this, web applications use session tracking mechanisms like Cookies and HttpSession. Cookies store small amounts of data on the client side, while HttpSession securely maintains user details on the server, enabling features like user authentication, personalized dashboards, and shopping carts.

Security is a critical concern in web applications, as they are vulnerable to threats such as SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and Session Hijacking. These attacks can compromise data integrity, steal credentials, or execute malicious scripts. To mitigate such risks, developers use PreparedStatements to prevent SQL injection, perform input validation to filter harmful data, and implement CSRF tokens and

secure cookies to protect session data. Using HTTPS for encrypted communication, regenerating session IDs after login, and properly invalidating sessions after logout further strengthen security.

By following best practices—such as sanitizing user inputs, employing secure transport protocols, and enforcing proper authentication and authorization—developers can ensure confidentiality, integrity, and availability of web applications. A secure and state-aware web design not only enhances user trust but also protects sensitive information from unauthorized access and data breaches, forming the foundation of reliable and professional web systems.

14.5 KEY TERMS

HTTP (HyperText Transfer Protocol), Servlet, HTTP Request, HTTP Response, GET Method, POST Method, Session Tracking, Cookie, HttpSession, PreparedStatement, SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Session Hijacking, Encryption.

14.6 SELF-ASSESSMENT QUESTIONS

1. What does HTTP stand for?
2. What is the role of a Servlet in a web application?
3. Which HTTP method is used to retrieve data from the server?
4. Which HTTP method is used to send data to the server?
5. What is a Cookie used for?
6. What is an HttpSession?
7. Why do we need session tracking in web applications?
8. What is SQL Injection?
9. What is the purpose of encryption in web security?
10. Name one way to prevent Cross-Site Scripting (XSS) attacks.

14.7 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition by Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approach by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.
3. Java Programming with Oracle JDBC by Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development by David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applications by Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBC by Gregory Brill. Sybex.

Dr. U. Surya Kameswari

LESSON-15

INTRODUCTION TO JSP

AIM AND OBJECTIVES:

- Understand the need for JSP and its relationship with Servlets.
- Explain the shortcomings of Servlets and how JSP overcomes them.
- Describe the architecture and anatomy of a JSP page.
- Identify and use various JSP components such as scripting elements, directives, and implicit objects.
- Differentiate between scriptlets, expressions, and declarations in JSP.
- Develop simple JSP programs to generate dynamic web content.

STRUCTURE:

15.1 INTRODUCTION TO JAVA SERVER PAGES (JSP)

Introduction

In modern web development, the need for dynamic and interactive web applications has led to the evolution of several server-side technologies. Among these, Java Server Pages (JSP) plays a vital role in creating dynamic, platform-independent, and efficient web-based solutions. JSP is built on top of Java technology and provides a simplified way to develop web pages that can dynamically generate content in response to client requests.

JSP was introduced by Sun Microsystems as an extension to the existing Java Servlet technology. It overcomes the limitations associated with Servlets by providing a more convenient and flexible approach to web page development. With JSP, developers can easily separate presentation logic from business logic, thus improving maintainability and readability of web applications.

What is JSP?

Java Server Pages (JSP) is a server-side scripting technology used for developing dynamic web applications. It enables developers to embed Java code within HTML pages using special JSP tags. When a client (such as a web browser) sends a request to a JSP page, the web server processes the page, executes the embedded Java code, and dynamically generates the HTML response that is sent back to the client.

JSP is a part of the Java Enterprise Edition (Java EE) platform and runs on a web container such as Apache Tomcat or Jetty. It provides a high-level abstraction for creating dynamic web content and simplifies the process of integrating server-side logic with front-end design.

Relationship between Servlets and JSP

JSP is often described as an extension of Servlets, as both technologies are closely related and serve complementary purposes. While Servlets are Java programs that run on the server and generate dynamic content, JSP provides a more convenient way to achieve the same result using HTML-like syntax.

In fact, behind the scenes, every JSP file is internally translated into a Servlet by the JSP engine. This Servlet is then compiled and executed by the web container. Therefore, JSP and Servlets are two sides of the same coin — JSP focuses on presentation (view), while Servlets handle request processing and business logic (controller).

JSP enhances the Servlet model by offering additional features such as:

- Expression Language (EL) for simplifying data access.
- JSTL (JSP Standard Tag Library) for reusable tag-based programming.
- Custom Tags for extending functionality.

These features make JSP a more powerful and flexible tool for developing dynamic web applications.

Issues in Servlet-Based Development:

Although Servlets provide a strong foundation for server-side programming in Java, they have several limitations that make web development complex and less efficient. These shortcomings led to the creation of JSP as a more user-friendly alternative.

a. Inability to Use IDEs for Designing HTML Views

Servlets primarily involve writing HTML content within Java code using out.println() statements. This makes it extremely difficult to design and visualize HTML pages using graphical Integrated Development Environments (IDEs). As a result, developers cannot use drag-and-drop or WYSIWYG (What You See Is What You Get) tools for designing the user interface, leading to increased development time and complexity.

b. Inability to Use HTML Designers

Since HTML content in Servlets is embedded within Java code, it becomes impossible for non-programmers, such as web designers, to work on the presentation layer independently. The presentation logic is tightly coupled with Java code, forcing the development team to rely entirely on Java programmers for even minor visual changes.

c. Dependence on Web Configuration Files
Servlets are heavily dependent on web configuration settings, particularly the web.xml deployment descriptor file. Every Servlet must be declared and mapped within this configuration file. This dependency introduces additional overhead during the development and deployment process, especially when the number of Servlets grows in a large web application.

d. Recurring Recompilation and Server Restart
When any modification is made in the Servlet code, developers must manually stop the server, recompile the source code, and restart the server to reflect the changes. This repetitive process slows down development and testing, making Servlets less efficient for rapid application development.

Understanding JSP Architecture
A JSP document consists of two main components:

1. HTML — used to design the static structure and layout of a web page.
2. JSP Tags and Scripting Elements — used to insert dynamic content and execute server-side logic.

When a JSP page is requested by a client:

1. The JSP engine (part of the web server) translates the JSP page into a Servlet.
2. This translated Servlet is then compiled into bytecode and executed by the Java Virtual Machine (JVM).
3. The output of the execution (typically HTML) is sent back to the client's browser as the final web page.

Thus, JSP combines the flexibility of HTML with the power of Java to create efficient, dynamic web pages.

Advantages of JSP:
JSP offers several advantages over Servlets and other server-side technologies. Some of the key benefits are discussed below:

a. Tag-Based Programming
JSP supports tag-based programming, where developers use JSP tags instead of embedding raw Java code within HTML. This approach simplifies web page development and allows even those with limited Java knowledge to create dynamic content easily.

b. Accessibility for Both Java and Non-Java Programmers
Since JSP files closely resemble standard HTML files with embedded tags, they can be easily understood and maintained by both Java developers and non-Java designers. This promotes collaboration between web designers and backend programmers.

c. Automatic Recognition of Modifications
When changes are made to a JSP file, the web container automatically detects and recompiles the modified file. There is no need to stop or restart the server manually, as required in Servlets. This feature greatly enhances productivity and efficiency.

d. Built-in Exception Handling
JSP provides mechanisms for error handling and exception management directly within the page. Developers can use predefined error pages or the <error-page> element in configuration files to manage exceptions gracefully.

e. Separation of Presentation and Business Logic
JSP encourages the Model-View-Controller (MVC) architecture by separating presentation logic (HTML, CSS, and JSP tags) from business logic (Java classes and Servlets). This separation leads to more modular, maintainable, and scalable applications.
f. Enhanced Code Readability
The use of tags and Expression Language in JSP improves the readability of the code, making it easier to debug, maintain, and extend over time.
g. Support for Built-in and Custom Tags
JSP provides a comprehensive set of built-in tags, such as those from the JSP Standard Tag Library (JSTL), and also supports the creation of custom tags. Developers can define their own tags or integrate third-party tag libraries to enhance functionality.
h. Session Management
JSP provides built-in session management features. Developers can easily store and retrieve user-specific information using implicit session objects without explicitly managing session handling code.
i. Implicit Objects
JSP offers a set of implicit objects such as request, response, session, application, out, and others. These objects are automatically available in every JSP page, simplifying server-side programming and data exchange between the client and the server.

Example 1: Simple JSP Program
File: hello.jsp

```
<html>
<head><title>Welcome Page</title></head>
<body>
<h2>Welcome to JSP!</h2>
<%
      String name = "Student";
out.println("<p>Hello, " + name + "! This is your first JSP program.</p>");
   %>
</body>
</html>
```

Output in Browser:
Welcome to JSP!
Hello, Student! This is your first JSP program.

Example 2: JSP with User Input
File: input.jsp

```
<html>
<head><title>User Input Example</title></head>
<body>
<form action="display.jsp" method="post">
      Enter your name: <input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

File: display.jsp

```
<html>
```

```
<head><title>Display Page</title></head>
<body>
<%
      String user = request.getParameter("username");
if (user == null || user.trim().equals("")) {
out.println("<h3>Please enter a valid name.</h3>");
      } else {
out.println("<h3>Hello, " + user + "! Welcome to JSP programming.</h3>");
      }
   %>
</body>
</html>
```

Input:
User enters: John
Output:
Hello, John! Welcome to JSP programming.

Advantages of Tag-Based Approach in JSP

- Reduces the amount of Java code embedded in HTML.
- Simplifies web application development.
- Easier to work with JavaBeans and reusable components.
- Supports separation of presentation and business logic.
- Enhances readability and maintainability of code.

Why JSP?
Servlets require writing a lot of Java code to generate HTML responses. For example, to display a simple message using a servlet, you must write:

```
out.println("<html>");
out.println("<body>");
out.println("<h1>Welcome to Java Servlets</h1>");
out.println("</body>");
out.println("</html>");
```

This approach mixes Java code and HTML, making maintenance difficult.
JSP solves this problem by allowing you to write HTML normally and embed Java logic only where needed.

15.2 THE PROBLEM WITH SERVLET

Servlets are Java programs that run on a web server and dynamically generate web pages. They handle client requests and responses using Java code.
However, when building large web applications, Servlets become difficult to maintain, hard to read, and time-consuming for web designers — mainly because they mix HTML and Java logic together.

Typical Servlet Workflow

1. The client (browser) sends a request to the server.
2. The server calls the Servlet, which contains Java code.
3. The Servlet generates HTML output using Java code.
4. The generated HTML is sent back to the client.

The Core Problem
While Servlets are powerful, they have some serious drawbacks when used to create complex web pages.

Main Problems with Servlets:

Problem	Description
1. Mixed Code (HTML + Java)	Writing HTML inside Java out.println() statements makes the code long and messy.
2. Poor Readability	Hard for web designers to edit HTML within Java source code.
3. Maintenance Difficulty	Any small change in the webpage design requires Java recompilation.
4. No Clear Separation	Servlets combine both business logic and presentation logic.
5. Slower Development	Requires both Java and HTML knowledge for every modification.

Example: Servlet Generating HTML Output
File: HelloServlet.java

```
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class HelloServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>Servlet Example</title></head>");
out.println("<body>");
out.println("<h2>Welcome to Java Servlets!</h2>");
out.println("<p>This page is generated using Java code.</p>");
out.println("</body>");
out.println("</html>");
out.close();
   }
}
```

Web Deployment Descriptor (web.xml):

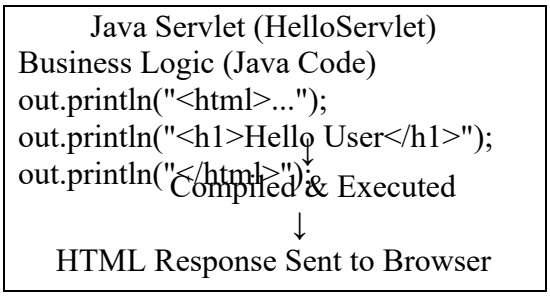
```
<web-app>
<servlet>
<servlet-name>hello</servlet-name>
<servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>hello</servlet-name>
<url-pattern>/hello</url-pattern>
</servlet-mapping>
</web-app>
```

Browser Request (Input):
http://localhost:8080/YourApp/hello

Output (in Browser):
Welcome to Java Servlets!
This page is generated using Java code.
It works fine — but the HTML is buried inside Java code!

Why This Problem?
Imagine the designer wants to change the HTML layout, color, or font.
They would need to open the Java file, find the out.println() lines, and modify the HTML — then recompile and redeploy the servlet.
That's inefficient and error-prone.
Overview-The below is a conceptual flowdiagram showing how Servlets mix business and presentation logic:



Problem: The same file handles both application logic and HTML layout, which is not modular.

Example of JSP Solution
Let's see how JSP fixes the same problem:
File: hello.jsp

```
<html>
<head><title>JSP Example</title></head>
<body>
<h2>Welcome to JSP!</h2>
<p>This page is generated using JSP — HTML is clean and easy to read.</p>
</body>
</html>
```

Output:
Welcome to JSP!
This page is generated using JSP — HTML is clean and easy to read.
No messy out.println() statements — only simple HTML!

Servlet Problems

Issue	Explanation
Mixing of HTML & Java	Reduces readability and maintainability.
Difficult Design Changes	Designers can't edit HTML without Java knowledge.
No MVC Separation	Business logic and view are tightly coupled.

Issue	Explanation
Compilation Overhead	Every design change requires recompiling the servlet.

Comparison: Servlet vs JSP

Feature	Servlet	JSP
Code Type	Java code with embedded HTML	HTML with embedded Java
Ease of Design	Difficult for web designers	Easier for web designers
Maintenance	Requires recompilation	JSP auto-compiles
Use Case	Good for backend logic	Good for presentation layer

Example: To print current date in Servlets and in JSP.

1. Servlet File

```
// Import required classes
import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
importjavax.servlet.annotation.WebServlet;
importjava.util.Date;

// Map this servlet to a URL
@WebServlet("/DateSrv")
public class DateSrv extends GenericServlet {

   // Implement service() method
public void service(ServletRequestreq, ServletResponse res)
throwsIOException, ServletException {

      // Set response content type
res.setContentType("text/html");

      // Get the PrintWriter to write response
PrintWriter pw = res.getWriter();

      // Get current date and time
      Date date = new Date();

      // Write HTML output
pw.println("<html>");
pw.println("<head><title>Display Current Date & Time</title></head>");
pw.println("<body>");
pw.println("<h2>Current Date & Time: " + date.toString() + "</h2>");
pw.println("</body>");
pw.println("</html>");

      // Close the PrintWriter
pw.close();
   }
}
```

2. JSP File

```
<%@ page import="java.util.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<h2>Current Date &Time:</h2>
<%
// Create a Date object
Datedate=newDate();

// Print current date and time
out.print(date.toString());
   %>
</body>
</html>
```

15.3 ANATOMY OF A JSP PAGE

A Java Server Page (JSP) file is essentially a text document that combines HTML markup with Java code. It is used to build dynamic web pages, where the static content is written in HTML and the dynamic content is generated using JSP elements.

Each JSP file is eventually translated into a Servlet by the web server (e.g., Apache Tomcat). This allows the Java code to be executed on the server before sending the HTML result to the client.

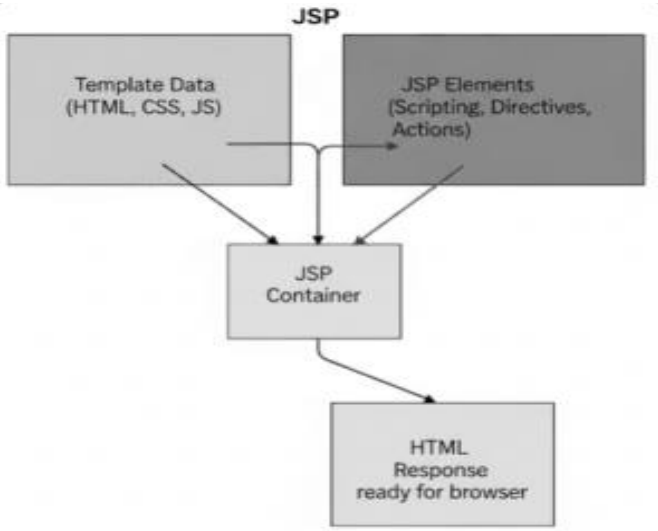


Figure: Anatomy of a JSP Page

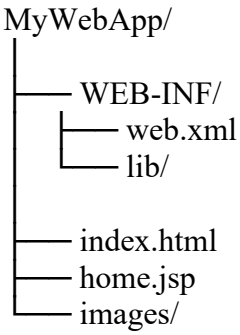
15.3.1 Components of a JSP Page

A JSP page may include the following components:

Component	Description
Scripting Elements	Used to insert Java code inside JSP.
Directives	Provide instructions to the JSP container on how to translate the page.
Action Tags	Used to control the behavior of the servlet engine.
Implicit Objects	Predefined variables provided by the JSP engine.
Comments	Used to document the code within the JSP page.

15.3.2 DIRECTORY STRUCTURE OF JSP

The directory structure of a JSP-based web application is similar to that of a Servlet application.



Typical Directory Structure of a JSP Application

Basic Structure of a JSP Page

Here's what a simple JSP file looks like:

```
<%@ page language="java" contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head><title>My First JSP Page</title></head>
<body>
<h2>Welcome to JSP!</h2>
<%
      // Scriptlet: Java code inside JSP
      String name = "Student";
out.println("<p>Hello, " + name + "!</p>");
  %>
</body>
</html>
```

Output in Browser:

Welcome to JSP!
Hello, Student!

15.3.3 JSP Scripting Elements

Scripting elements allow Java code to be embedded directly within a JSP file. These elements are processed by the JSP engine during page translation and converted into equivalent servlet code.
There are three types of scripting elements in JSP:

1. **Scriptlets**
2. **Expressions**
3. **Declarations**

1.Scriptlet Tag (<% ... %>)
A **scriptlet** contains Java statements that are executed every time the JSP page is requested. Scriptlets are enclosed between <% and %> tags.
Syntax:

```
<%
  // Java statements
%>
```

Example 1:

```
<html>
<body>
<% out.print("Welcome to JSP!"); %>
</body>
</html>
```

Explanation:
The text inside <% %> is Java code that runs during the request. The output is sent to the client browser through the out implicit object.

Example 2: Reading Input Using Scriptlet

```
<!-- HTML Form -->
<form action="welcome.jsp">
<input type="text" name="uname" placeholder="Enter your name">
<input type="submit" value="Submit">
</form>
<!-- welcome.jsp -->
<html>
<body>
<%
  String name = request.getParameter("uname");
out.print("Welcome " + name);
%>
</body>
</html>
```

2. Expression Tag (<%= ... %>)
The **expression tag** is used to output values directly to the client. It evaluates the given expression and inserts the result into the response.
Syntax:

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

Example:

```
<html>
```

```
<body>
<h2>JSP Expression Example</h2>
<p>Sum of 5 and 7 is: <%= 5 + 7 %></p>
<p>Current Date: <%= new java.util.Date() %></p>
</body>
</html>
```

Explanation:

Anything inside <%= %> is evaluated and printed automatically, similar to out.print().

3. Declaration Tag (<%! ... %>)

A **declaration tag** is used to define variables and methods that are shared across multiple service requests. It is enclosed within <%! and %>.

Syntax:

```
<%! dataTypevariableName; %>
<%! returnTypemethodName() { ... } %>
```

Example:

```
<html>
<body>
<%! int data = 50; %>
<%= "Value of data: " + data %>
</body>
</html>
```

Example with Method Declaration:

```
<html>
<body>
<%!
int cube(int n) {
return n * n * n;
    }
  %>
<%= "Cube of 4 is: " + cube(4) %>
</body>
</html>
```

Complete Example: Combining All Elements

File: anatomyExample.jsp

```
<%@ page import="java.util.*" %>
<%! int counter = 0; %>
<html>
<head><title>JSP Anatomy Example</title></head>
<body>
<%-- This is a JSP comment, not visible in browser --%>
<h2>Understanding JSP Components</h2>
<%
counter++;
   String user = "Alice";
   Date now = new Date();
%>
<p>Hello, <%= user %>!</p>
<p>You are visitor number <%= counter %>.</p>
<p>Current date and time: <%= now %></p>
<jsp:include page="footer.jsp" />
```

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

File: footer.jsp

```
<hr>
<p>Thank you for visiting our JSP demo page.</p>
```

Output in Browser:

```
Understanding JSP Components
Hello, Alice!
You are visitor number 1.
Current date and time: Wed Oct 29 13:05:10 IST 2025
-----------------------------------------------
Thank you for visiting our JSP demo page.
```

Difference betweenScriptlet, Expression, and Declaration

Feature	Scriptlet	Expression	Declaration
Purpose	To write logic and control flow statements	To output values directly	To declare variables and methods
Syntax	<% ... %>	<%= ... %>	<%! ... %>
Placement in Servlet	Inside _jspService()	Inside out.print()	Outside _jspService()
Execution	Runs for every request	Evaluated and printed	Executed when class is loaded

15.3.4 JSP Implicit Objects

Implicit objects are predefined variables created by the JSP container that are directly accessible in JSP pages without explicit declaration. They simplify coding by providing ready-to-use objects for handling requests, responses, sessions, etc.

There are nine implicit objects in JSP.

Object	Class	Description
request	HttpServletRequest	Represents the client request.
response	HttpServletResponse	Used to send response to client.
out	JspWriter	Used to send output to the browser.
config	ServletConfig	Provides initialization parameters.
application	ServletContext	Provides information about the application.
session	HttpSession	Used for session management.
pageContext	PageContext	Provides context for JSP page operations.
page	Object	Refers to the current JSP page instance (similar to this).
exception	Throwable	Used for exception handling in error pages.

The request Object:

- Represents the client's request to the server.
- Used to access form data, headers, and cookies.
- Belongs to the HttpServletRequest class.

Example:

```
<form action="RequestDemo.jsp">
<input type="text" name="uname" placeholder="Enter Name">
<input type="submit" value="Send">
</form>
<!-- RequestDemo.jsp -->
<%
  String name = request.getParameter("uname");
out.print("Welcome " + name);
%>
```

The response Object:

- Used to send data or redirect the client to another resource.
- Belongs to the HttpServletResponse class.

Example:

```
<form action="ResponseDemo.jsp">
<input type="text" name="query" placeholder="Search term">
<input type="submit" value="Search">
</form>
<!-- ResponseDemo.jsp -->
<%
  String search = request.getParameter("query");
response.sendRedirect("https://www.google.com/search?q=" + search);
%>
```

The out Object:

- Used to send output to the client browser.
- Belongs to the JspWriter class.

Common Methods:

```
out.print(data);
out.println(data);
```

Example:

```
<%
out.println("Hello from JSP!");
%>
```

The session Object:

- Represents a user's session and stores data between multiple requests.
- Belongs to the HttpSession class.

Example:

```
<!-- welcome.jsp -->
<%
  String user = request.getParameter("uname");
session.setAttribute("username", user);
out.print("Welcome " + user);
%>
<a href="next.jsp">Next Page</a>
<!-- next.jsp -->
```

```
<%
  String user = (String) session.getAttribute("username");
out.print("Hello again, " + user);
%>
```

The exception Object:

- Used for error handling.
- Available only in JSP pages declared with isErrorPage="true".

Example:

```
<%@ page errorPage="error.jsp" %>
<%
int a = Integer.parseInt(request.getParameter("x"));
int b = Integer.parseInt(request.getParameter("y"));
int c = a / b;
out.print("Result: " + c);
%>
<!-- error.jsp -->
<%@ page isErrorPage="true" %>
<h2>Sorry! An Exception Occurred.</h2>
<p><b>Details:</b><%= exception %></p>
```

The application Object:

- Represents the entire web application.
- Used to share data among all JSP pages.
- Belongs to ServletContext.

Example:

```
<%
  String college = application.getInitParameter("collegeName");
out.print("Welcome to " + college);
%>
```

15.4 SUMMARY

Java Server Pages (JSP) is a server-side technology built on top of Java Servlets, designed to simplify the creation of dynamic and interactive web pages. Unlike Servlets, which mix HTML and Java code in the same file, JSP enables developers to write HTML content with embedded Java code, making web applications easier to develop, maintain, and modify.

When a JSP page is requested by a client, the JSP engine first translates it into a Servlet, then compiles and executes it within the Java Virtual Machine (JVM). The resulting HTML output is then sent back to the client's web browser. This internal translation process allows JSP to combine the flexibility of Java with the simplicity of web page design.

JSP offers several advantages over traditional Servlets, making web development more efficient and maintainable. It simplifies the process of creating dynamic web content by allowing developers to embed Java code directly within HTML, thereby reducing the complexity of mixing presentation and logic. JSP automatically recompiles pages whenever changes are made, eliminating the need for manual deployment after minor modifications. It also supports tag-based programming, which helps in writing cleaner and more readable code.

In addition, JSP provides a set of predefined implicit objects such as request, response, and session, making common web development tasks easier to perform. Furthermore, built-in exception handling ensures that applications are more reliable and robust, contributing to improved performance and error management in dynamic web applications.

Overall, JSP separates the presentation layer (HTML) from the business logic (Java), promoting cleaner code organization and enabling collaboration between web designers and programmers. This separation of concerns makes JSP a powerful and efficient technology for developing scalable, dynamic, and maintainable web applications.

15.5 KEY TERMS

JSP (Java Server Pages), Servlet, JSP Engine, Scriptlet, Expression Tag, Declaration Tag,

Implicit Objects, MVC (Model-View-Controller), JSTL (JSP Standard Tag Library), Tag-Based Programming

15.6 SELF-ASSESSMENT QUESTIONS

1. Explain the limitations of Servlets and how JSP overcomes them.
2. Describe the architecture of JSP with a suitable diagram.
3. Differentiate between Scriptlet, Expression, and Declaration tags in JSP.
4. Explain the purpose and usage of implicit objects in JSP with examples.
5. Write and explain a JSP program that accepts user input and displays a personalized greeting.
6. How does JSP differ from Servlets?
7. What are the advantages of using JSP over Servlets?
8. List the main components of a JSP page.
9. Define scripting elements in JSP.
10. What is the use of the <%= … %> tag?
11. Name any five implicit objects available in JSP.

15.7 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Editionby Herbert Schildt. McGraw-Hill Education.
2. Beginning Java Programming: The Object-Oriented Approachby Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.
3. Java Programming with Oracle JDBCby Donald Bales. O'Reilly Media.
4. Java EE 8 Application Development *by* David R. Heffelfinger. Packt Publishing.
5. Professional Java for Web Applicationsby Nicholas S. Williams. Wrox/Wiley Publishing.
6. Java 2: Developer's Guide to Web Applications with JDBCby Gregory Brill. Sybex.
7. Beginning JSP, JSF and Tomcat Web Development: From Novice to ProfessionalbyGiulio Zambon& Michael Sekler. Apress.

Dr. U. Surya Kameswari

LESSON-16
JSP PROCESSING AND APPLICATION DESIGN

AIM AND OBJECTIVES:

- Explain the concept and working of JSP as a server-side technology.
- Describe each phase of the JSP lifecycle — translation, compilation, initialization, execution, and destruction.
- Understand how a JSP file is internally converted into a servlet.
- Demonstrate JSP processing through practical examples.
- Identify the advantages of JSP processing in web application development.
- Illustrate how JSP fits into the MVC architecture for separating business logic, control logic, and presentation.
- Develop a simple MVC-based web application using JSP, Servlets, and JavaBeans.

STRUCTURE:

16.1 JSP PROCESSING

JSP (JavaServer Pages) is a server-side technology that allows you to create dynamic web content by embedding Java code within HTML.

However, unlike plain HTML pages, JSPs are compiled and executed on the server before the output (HTML) is sent to the client browser.The server handles the processing of a JSP file in several stages — converting it into a Servlet, compiling it, and executing it to produce HTML output.

16.1.1 JSP Processing Overview

When a client (like a web browser) requests a JSP page, theweb container (e.g., Apache Tomcat) performs the following steps:

1. **Translation Phase:**
   The JSP file is translated into a Java Servlet source file.
2. **Compilation Phase:**
   The generated Servlet source is compiled into a .class file (bytecode).
3. **Loading & Initialization Phase:**
   The Servlet class is loaded into memory and initialized.
4. **Request Processing Phase:**
   The Servlet's service() method executes, processing client requests and generating responses.
5. **Response Phase:**
   The output (HTML) is sent back to the client browser.
6. **Destruction Phase:**
   When the application is stopped, the JSP Servlet is destroyed.

16.1.2 JSP Processing Flow Diagram

The complete JSP processing sequence creates a smooth transition from human-readable .jsp code to machine-executable bytecode, ultimately producing a dynamic web response. This powerful lifecycle serves as the foundation for all dynamic web applications developed using JSP. As shown in below figure.

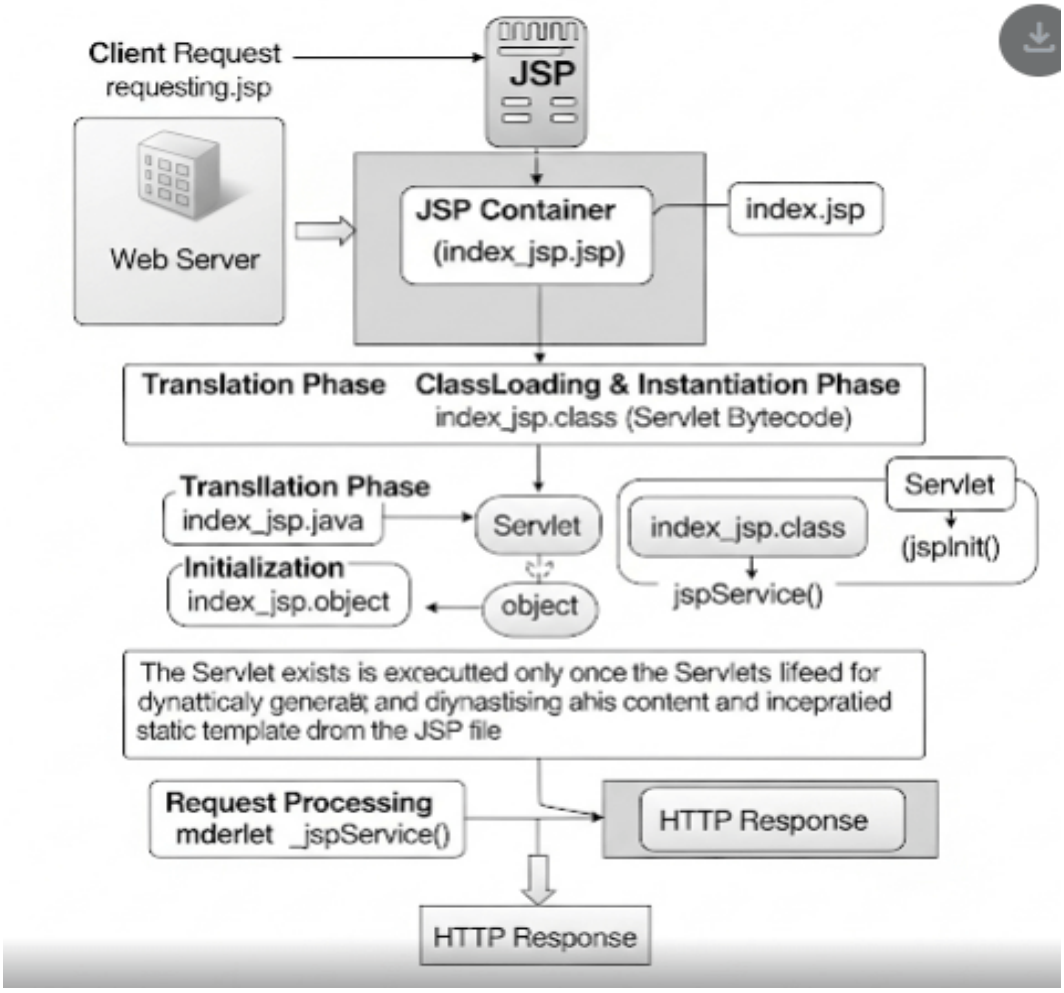


Figure: JSP Processing Flow Diagram

JSP Lifecycle Methods

Internally, a JSP page is converted into a Servlet, and the Servlet follows a defined lifecyclemanaged by the JSP container**.**

JSP Lifecycle Method	Description
jspInit()	Called once when the JSP is initialized.
jspService()	Called for each client request (contains the main logic).
jspDestroy()	Called before the JSP is destroyed.

5. Step-by-Step JSP Processing Example
Let's demonstrate JSP processing using a real example.
Example Files:
File 1:welcome.jsp
File 2:web.xml (deployment descriptor)
File: welcome.jsp

```
<%@ page language="java" contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head><title>JSP Processing Example</title></head>
<body>
<%-- JSP comment: This will not appear in browser --%>
<%!
intvisitCount = 0;
   // This method runs once when JSP is initialized
public void jspInit() {
System.out.println("JSP Initialized...");
   }
   // This method runs when the JSP is destroyed
public void jspDestroy() {
System.out.println("JSP Destroyed...");
   }
%>
<%
   // This part executes for each request
visitCount++;
   String user = request.getParameter("name");
if (user == null || user.equals("")) {
user = "Guest";
   }
%>
<h2>Welcome to JSP Processing!</h2>
<p>Hello, <%= user %>!</p>
<p>You are visitor number: <%= visitCount %></p>
<form action="welcome.jsp" method="post">
   Enter your name: <input type="text" name="name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Step-by-Step JSP Processing Flow:

Step 1 – Client Request

The browser requests:

http://localhost:8080/JSPApp/welcome.jsp

Step 2 – Translation Phase

The server (Tomcat) translates welcome.jsp → welcome_jsp.java

(This is an automatically generated servlet.)

Step 3 – Compilation

welcome_jsp.java → compiled into welcome_jsp.class

Step 4 – Initialization

The container calls:

jspInit();

This runs only once (prints "JSP Initialized..." on the server console).

Step 5 – Request Handling

The container calls:

_jspService(request, response);

This method executes the main JSP code — reading parameters, generating output, etc.

Step 6 – Response Generation

The generated HTML is sent back to the client browser.

Step 7 – Destruction

When the JSP is reloaded or the server stops, the container calls:

jspDestroy();

Example Input and Output

Input 1 (First Visit):

User opens the page without entering a name.

URL:

http://localhost:8080/JSPApp/welcome.jsp

Output 1:

Welcome to JSP Processing!

Hello, Guest!

You are visitor number: 1

[Text box to enter name]

Input 2 (Second Visit with Name):

User enters “Alice” in the text box and submits.

Output 2:

Welcome to JSP Processing!

Hello, Alice!

You are visitor number: 2

[Text box again for new input]

Server Console Output:

JSP Initialized...

JSP Destroyed... (when server stops or reloads)

Generated Servlet Code (Simplified View)

When Tomcat processes your JSP, it internally generates something like this:

```
public final class welcome_jsp extends HttpJspBase {
intvisitCount = 0;
public void jspInit() {
System.out.println("JSP Initialized...");
  }
```

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
throwsIOException, ServletException {
visitCount++;
      String user = request.getParameter("name");
if (user == null) user = "Guest";
JspWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h2>Welcome to JSP Processing!</h2>");
out.println("<p>Hello, " + user + "!</p>");
out.println("<p>You are visitor number: " + visitCount + "</p>");
out.println("</body></html>");
   }
public void jspDestroy() {
System.out.println("JSP Destroyed...");
   }
}
```

This shows how a JSP page internally becomes a servlet that executes Java code to produce HTML output.

16.1.3 JSP Lifecycle:

Java Server Pages (JSP) follow a well-defined life cycle managed by the JSP container. Understanding this life cycle is crucial for developing efficient and reliable JSP applications. The life cycle ensures that JSP pages are translated, compiled, executed, and finally destroyed in a structured manner.

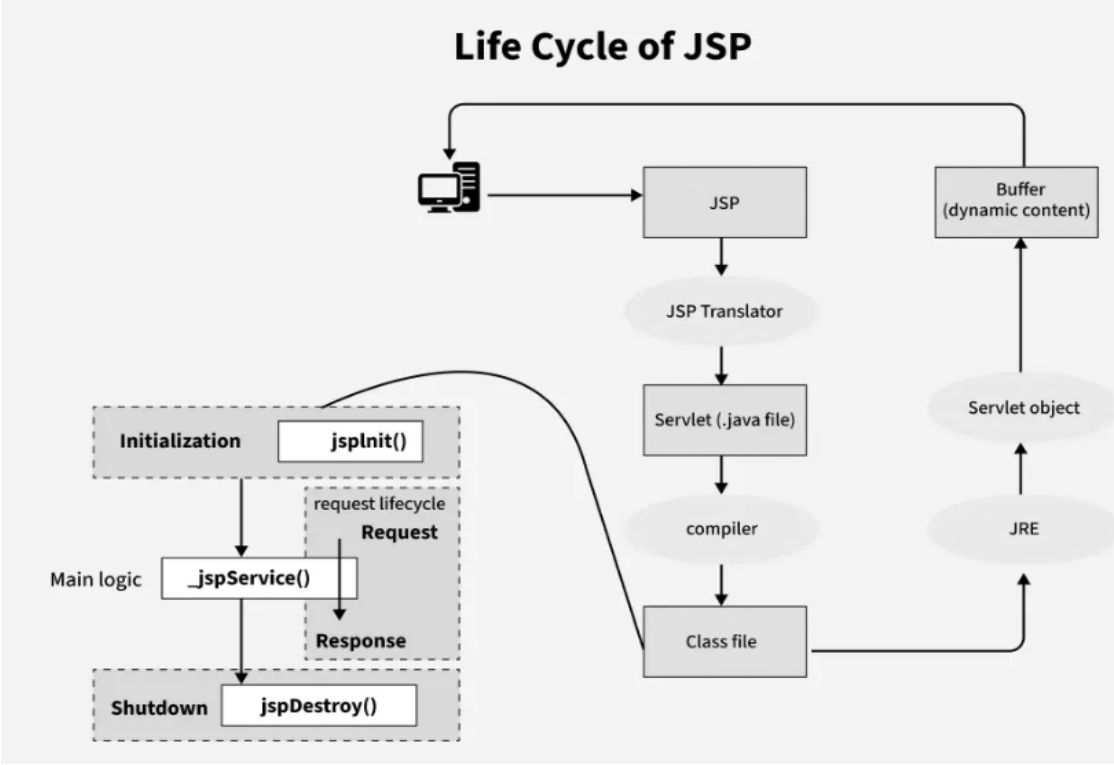


Figure: Life Cycle of JSP

Steps in the JSP Life Cycle:

1. **Translation:** The JSP page is translated into a corresponding servlet.
2. **Compilation:** The translated JSP file (e.g., test.java) is compiled into a class file (test.class).
3. **Class Loading:** The generated servlet class is loaded into the container.
4. **Instantiation:** An instance of the generated servlet class is created.
5. **Initialization:** The container invokes the jspInit() method to initialize the servlet.
6. **Request Processing:** The container calls the _jspService() method to handle client requests and generate responses.
7. **Cleanup:** The jspDestroy() method is invoked by the container to release resources before the JSP is unloaded.

1. Translation of JSP Page

The first phase of the JSP life cycle is the translation phase. In this stage, the JSP container converts the JSP document into an equivalent Java Servlet.

- The generated servlet contains Java code along with HTML/XML markup from the JSP page.
- The container also checks the syntactic correctness of the JSP page during this phase.
- Any errors in the JSP syntax are detected here before compilation.

Example:
A JSP file Test.jsp is translated into a servlet file Test.java by the container.

2. Compilation of JSP Page
After translation, the JSP container compiles the generated Java Servlet code into Java bytecode (class file).

- This compiled class is now ready to be loaded into the container and initialized.
- Compilation ensures that the JSP page can be executed efficiently by the JVM.

3. Class Loading
In this phase, the servlet class generated from the JSP page is loaded into the JSP container by the class loader.

- The container keeps the class in memory, making it ready for instantiation.
- This step is managed automatically by the JSP container.

4. Instantiation
Once the servlet class is loaded, the container creates an instance of the servlet class.

- The container may maintain one or more instances to handle multiple simultaneous requests.
- Instantiation ensures that each JSP page can process requests independently.

5. Initialization
After creating the servlet instance, the JSP container calls the jspInit**()** method.

- jspInit() is invoked only once in the life cycle.
- It is used to perform one-time initialization tasks, such as allocating resources, opening database connections, or setting up configuration parameters.
- This method can be overridden in JSP if custom initialization is required.

6. Request Processing

Once initialized, the JSP page is ready to handle client requests.

- The container invokes the _jspService() method for each client request.
- This method receives the request and response objects as parameters and generates the appropriate dynamic response.
- Important: _jspService() cannot be overridden. It is automatically generated by the JSP container.

7. Destruction (Clean-up)

The final phase of the JSP life cycle is clean-up or destruction.

- When the JSP page is no longer needed or the container shuts down, the jspDestroy() method is called.
- This method allows developers to perform cleanup tasks, such as closing open files, releasing database connections, or freeing resources.
- Unlike _jspService(), jspDestroy() can be overridden to include custom cleanup logic.

Advantages of JSP Processing

- JSP pages are compiled only once, not every time they are requested.
- Subsequent requests are faster since the compiled servlet is reused.
- Clear separation of presentation (HTML) and business logic (Java)**.**
- Automatic servlet generation— developers focus on web design, not boilerplate Java code.

Table:

Phase	Action	Description
Translation	JSP → Servlet	JSP converted into Java source code
Compilation	Servlet → Class	Compiled into bytecode
Initialization	jspInit()	Runs once when JSP loads
Request Handling	_jspService()	Runs for every request
Destruction	jspDestroy()	Runs when JSP unloads

16.2 JSP APPLICATION DESIGN WITH MVC SETTING UP AND JSP ENVIRONMENT

16.2.1 What is MVC?

16.2.2

MVC (Model–View–Controller) is a design pattern used in web development to separate an application into three logical layers:

Component	Description	Technology Used
Model	Contains business logic and data handling.	JavaBeans / POJO classes
View	Handles presentation and user interface.	JSP pages
Controller	Controls the flow of data between Model and View.	Servlets

This separation improves code organization, reusability, and maintenance.

How MVC Works in JSP Applications

1. Client (Browser) sends a request (e.g., submits a form).
2. The Controller (Servlet) receives the request and processes it.
3. The Controller interacts with the Model (Java class/Bean**)** for data or business logic.
4. The result from the Model is sent back to the Controller.
5. The Controller forwards the data to a View (JSP page) for displaying output.
6. The JSP renders the final HTML response to the browser.

16.2.2 MVC Flow Diagram

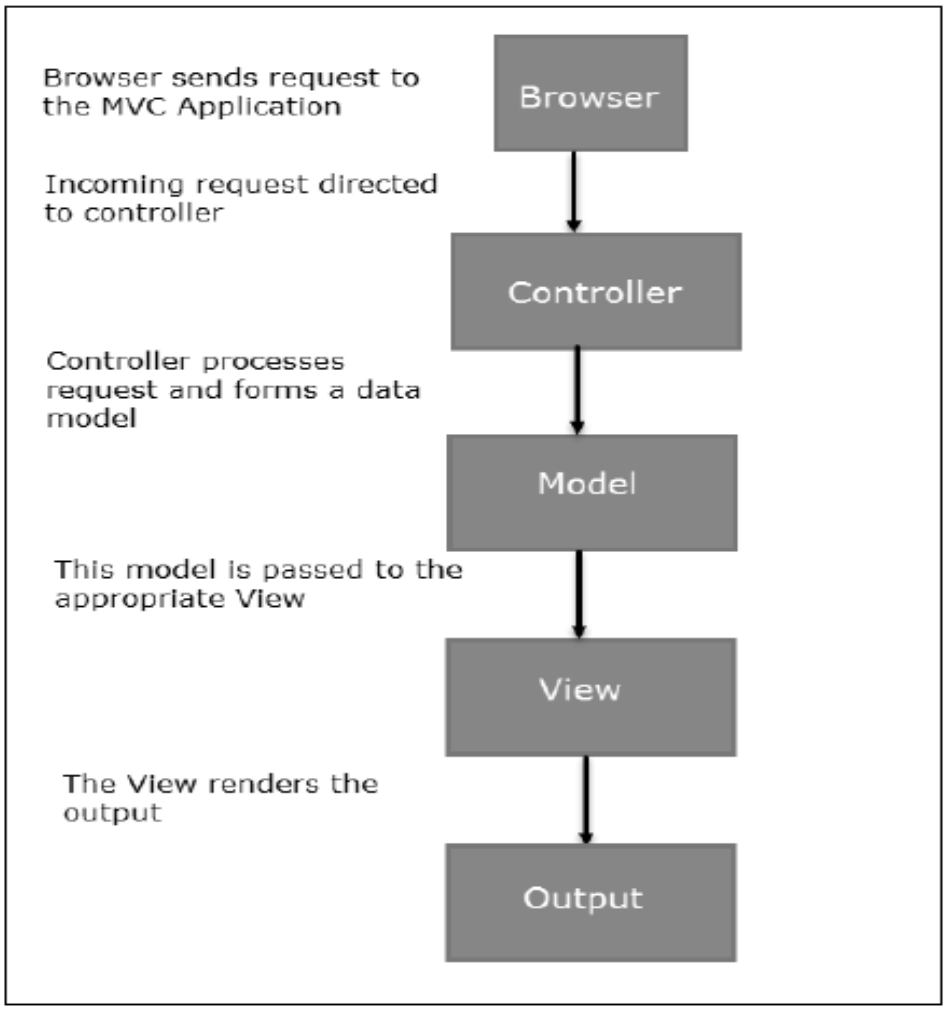


Figure: MVC Flow Diagram.

Example Program: JSP Application using MVC
Let's build a small MVC example where the user enters their name, and the application displays a welcome message.
Step 1 – Model (JavaBean)
File: User.java

```
packagemvcapp;
public class User {
private String name;
public User() {}
public String getName() {
```

```
return name;
   }
public void setName(String name) {
      this.name = name;
   }
public String greetUser() {
return "Welcome, " + name + "! You are learning JSP with MVC.";
   }
}
```

Step 2 – Controller (Servlet)

File: UserServlet.java

```
packagemvcapp;

import java.io.*;
importjavax.servlet.*;
importjavax.servlet.http.*;
public class UserServlet extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throwsServletException, IOException {
      // Get data from the form
      String username = request.getParameter("username");
      // Create model object
      User user = new User();
user.setName(username);
      // Store model data in request scope
request.setAttribute("userData", user);
      // Forward data to the view (JSP)
RequestDispatcherrd = request.getRequestDispatcher("welcome.jsp");
rd.forward(request, response);
   }
}
```

Step 3 – View (JSP Page)

File:welcome.jsp

```
<%@ page import="mvcapp.User" %>
<html>
<head><title>Welcome Page</title></head>
<body>
<%
      User user = (User) request.getAttribute("userData");
   %>
<h2><%= user.greetUser() %></h2>
</body>
</html>
```

Step 4 – HTML Form (Input Page)

File: index.html

```
<html>
<head><title>JSP MVC Example</title></head>
<body>
<h2>Enter Your Name</h2>
<form action="UserServlet" method="post">
```

```
<input type="text" name="username">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Step 5 – Web Deployment Descriptor

File: web.xml

```
<web-app>
<servlet>
<servlet-name>userServlet</servlet-name>
<servlet-class>mvcapp.UserServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>userServlet</servlet-name>
<url-pattern>/UserServlet</url-pattern>
</servlet-mapping>
</web-app>
```

Input / Output

Input (from index.html):

User enters:

John

Output (in Browser – welcome.jsp):

Welcome, John! You are learning JSP with MVC.

Advantages of MVC with JSP

- Clear separation of presentation (JSP), logic (Servlet), and data (Model).
- Easier maintenance and updates.
- Designers and developers can work independently.
- Promotes reusability and scalability.

16.2.3 Setting Up JSP Environment

To execute JSP programs, you need a proper Java web environment with the JDK and Tomcat server.

Step 1: Install Java Development Kit (JDK)

- Download from: https://www.oracle.com/java/technologies/javase-downloads.html
- Install and set environment variables:
- JAVA_HOME = C:\Program Files\Java\jdk-<version>
- PATH = %JAVA_HOME%\bin
- Verify installation:
- java -version

Step 2: Install Apache Tomcat Server

- Download from: https://tomcat.apache.org
- Extract it to: C:\Tomcat
- Set environment variable:
- CATALINA_HOME = C:\Tomcat
- Start Tomcat:
- C:\Tomcat\bin\startup.bat

- Test in browser:
- http://localhost:8080/

You should see the Tomcat welcome page.

Step 3: Deploy JSP Application

1. Inside C:\Tomcat\webapps, create a folder named mvcapp.
2. Inside it, create WEB-INF folder and place:
    - web.xml
    - classes folder (for .class files)
3. Place index.html and welcome.jsp in the mvcapp root folder.
4. Compile Java files (User.java, UserServlet.java) and place .class files inside:
5. C:\Tomcat\webapps\mvcapp\WEB-INF\classes\mvcapp\

Step 4: Test the Application

Open your browser and go to:
http://localhost:8080/mvcapp/index.html

Input:

Name: Alice

Output:

Welcome, Alice! You are learning JSP with MVC.

4. Table

Step	Component	File	Purpose
1	Model	User.java	Holds data & logic
2	Controller	UserServlet.java	Processes requests
3	View	welcome.jsp	Displays output
4	Configuration	web.xml	Maps servlet
5	Deployment	Tomcat Server	Runs JSP & Servlets

16. 3 SUMMARY

The JSP Processing Lifecycle defines how a JavaServer Page is transformed and executed within a web container to generate dynamic web content. When a client requests a JSP file, the server processes it through several systematic stages — starting with translation, where the JSP is converted into a Java servlet source file; followed by compilation, where this source is compiled into bytecode. The compiled servlet is then loaded and initialized, during which the jspInit() method executes once to prepare resources. During each client request, the _jspService() method handles input processing and generates the dynamic HTML response sent back to the browser. Finally, when the JSP is reloaded or the server stops, the jspDestroy() method executes to release resources and clean up. Essentially, every JSP acts as a servlet behind the scenes — such as welcome.jsp being converted into welcome_jsp.java — which ensures efficient execution, reusability, and high performance. This structured lifecycle enables JSPs to deliver robust, maintainable, and dynamic web applications, forming a vital part of Java-based web development.

JSP Application Design with MVC provides a systematic approach to developing dynamic and maintainable web applications by separating the application into three logical

components — **Model, View, and Controller (MVC)**. The Model represents the business logic and data, usually implemented using JavaBeans or POJO classes. The View is responsible for the presentation layer, created using JSP pages to display data to users. The Controller**,** typically implemented as a Servlet, manages the flow of data between the Model and View by handling user requests, invoking business logic, and forwarding results to the appropriate JSP page. This separation of concerns improves code organization, simplifies maintenance, and allows designers and developers to work independently.

16.4 KEY TERMS

JSP (JavaServer Pages), Servlet, Web Container, JSP Lifecycle ,jspInit(),_jspService(), jspDestroy(), Translation Phase, Compilation Phase, JSP Container, MVC Architecture, RequestDispatcher

16.5 SELF-ASSESSMENT QUESTIONS

1. Explain the complete JSP processing flow from request to response.
2. Define JSP and list its advantages over Servlets.
3. Describe the lifecycle methods of JSP with suitable examples.
4. Write and explain the steps to execute a JSP example on Apache Tomcat.
5. Compare JSP and Servlets in terms of processing and purpose.
6. Discuss the MVC architecture in JSP applications with an example.
7. Compare traditional JSP applications with MVC-based JSP applications.
8. What is the role of the Controller in the MVC architecture?

16.6 FURTHER READINGS

1. Java: The Complete Reference, Twelfth Edition *by Herbert Schildt. McGraw-Hill Education.*
2. Beginning Java Programming: The Object-Oriented Approach *by Bart Baesens, Aimee Backiel, and SeppevandenBroucke. Wiley.*
3. Java Programming with Oracle JDBC *by Donald Bales. O'Reilly Media.*
4. Java EE 8 Application Development *by David R. Heffelfinger. Packt Publishing.*
5. Professional Java for Web Applications *by Nicholas S. Williams. Wrox/Wiley Publishing.*
6. Java 2: Developer's Guide to Web Applications with JDBC *by Gregory Brill. Sybex.*
7. Beginning JSP, JSF and Tomcat Web Development: From Novice to Professional *by Giulio Zambon& Michael Sekler. Apress.*

Dr. U. Surya Kameswari