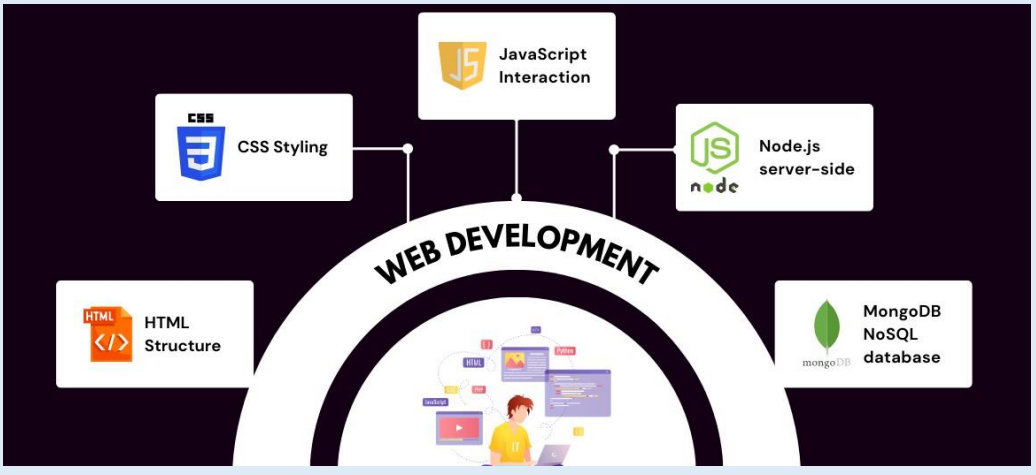
LECTURE NOTES

Subject Name: Full Stack Web Development (24MCA123)

Year / Branch: I MCA II SEMESTER

Regulation: R24

Prepared By: Mr. J. Pavan Sai, Assistant Professor



"Learning to write programs stretches your mind, and helps you think better, creates a way of thinking about things that I think is helpful in all domains." - Bill Gates

1. Syllabus

I MCA – II SEMESTER			
COURSE CODE:	**24MCA123**	**CREDITS:**	**4**
COURSE TITLE:	**FULL STACK WEB DEVELOPMENT**	**L-T-P:**	**4-0-0**

PREREQUISITES: *A course on "Java Programming"*

COUSE EDUCATIONAL OBJECTIVES :

CEO1 *:To understand the fundamentals of web programming and client side scripting.*

CEO2 : *To learn server side development using NodeJS.*

CEO3 : *To understand API development with Express Framework.*

CEO4 :*To learn the advanced client side scripting and ReactJS framework*

CEO5 : *To understand and architect databases using MongoDB and SQL databases*

UNIT- I: OVERVIEW OF HTML AND CSS:	***Lecture Hrs :12***

HTML5: Introduction to HTML5, Browsers and HTML5, Editor's Offline and Online, Tags, Attribute and Elements, Doctype Element, Comments, Headings, Paragraphs, and Formatting Text, Lists and Links, Images and Tables CSS: Introduction CSS, Applying CSS to HTML5, Selectors, Properties and Values, CSS Colors and Backgrounds, CSS Box Model, CSS Margins, Padding, and Borders, CSS Text and Font Properties, CSS General Topics

UNIT- II: OVERVIEW OF JAVASCRIPT	***Lecture Hrs :12***

JAVASCRIPT: Introduction to JavaScript, Applying JavaScript (internal and external), Understanding JS Syntax, Introduction to Document and Window Object, Variables and Operators, Data Types and Num Type Conversion, Math and String Manipulation, Objects and Arrays, Date and Time, Conditional Statements, Switch Case, Looping in JS and Functions.

UNIT- III: REACT JS:	***Lecture Hrs :12***

Introduction, Templating using JSX, Components, State and Props, Lifecycle of Components, Rendering List and Portals, Error Handling, Routers, Redux and Redux Saga, Immutable.js, Service Side Rendering, Unit Testing, Webpack.

UNIT- IV: NODE JS	***Lecture Hrs :12***

NodeJS: NodeJS Overview, NodeJS - Basics and Setup, NodeJS Console, NodeJS Command Utilities, NodeJS Modules, NodeJS Concepts, NodeJS Events, NodeJS with ExpressJS, NodeJS Database Access.

UNIT- V: MONGO DB	***Lecture Hrs :12***

MongoDB: SQL and NoSQL Concepts Create and Manage MongoDB, Migration of Data into MongoDB, MongoDB with ReactJS, MongoDB with NodeJS, Services Offered by MongoDB.

COURSE OUTCOMES: *On successful completion of this course, students will be able to:*		POs related to COs
CO1	Demonstrate knowledge on HTML5, CSS to design look and feel of the web applications.	PO1,PO3
CO2	Design and build the web applications using JavaScript Programming.	PO1, PO2,PO3,PO4
CO3	Apply ReactJS Framework to design and develop user Interfaces and rapid frontend web applications for society as well as Enterprise	PO1,PO2,PO3
CO4	Analyze NodeJS tool Utilities and modules to implement back-end applications.	PO1,PO3
CO5	Apply MongoDB to design and organize the database for development of Web Applications.	PO1,PO2, PO3, PO4,PO6,PO8

TEXT BOOKS:

1. *Vasan Subramanian, "Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node," Apress, 2nd Edition, 2019.*
2. *Kogent Learning Solutions Inc., "Web Technologies Black Book," Dreamtech Press, 2011.*

REFERENCE BOOKS:

1. *Brad Dayley, "Node.js, MongoDB, and AngularJS Web Development," Second Edition, 2018.*
2. *Uttam K Roy, "Web Technologies," Oxford University Press, 2010.*

ONLINE LEARNING RESOURCES:

https://www.w3schools.com/html/default.asp
https://www.w3schools.com/css/default.asp
https://www.w3schools.com/js/default.asp
https://www.w3schools.com/react/default.asp
https://www.w3schools.com/nodejs/default.asp
https://www.w3schools.com/mongodb/index.php
https://react.dev/learn
https://nodejs.org/en
https://www.mongodb.com/docs/

CO-PO MAPPING(DETAILED; HIGH:3; MEDIUM:2; LOW:1)

Course	POs / COs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8
C203: Full Stack Web Development	C203.1	3	-	2	-	-	-	-	-
	C203.2	3	2	3	2	-	-	-	-
	C203.3	3	3	3	-	-	-	-	-
	C203.4	3	-	3	-	-	-	-	-
	C203.5	3	3	3	3	-	2	-	2
	C203	3	2.	2.8	2.5	-	2	-	2

FULL STACK WEB DEVELOPMENT

UNIT –I

OVERVIEW OF HTML AND CSS

Design is not just what it looks like and feels like. Design is how it works.

— Steve Jobs

UNIT –I: OVERVIEW OF HTML AND CSS

1. Unit Overview

This unit introduces the fundamentals of **HTML5 and CSS**, explaining how web pages are structured and styled. Topics include HTML5 tags, attributes, elements, headings, paragraphs, lists, links, images, tables, and formatting text, as well as CSS basics, selectors, properties, colors, backgrounds, box model, margins, padding, borders, text, and font styling.

2. Objectives of the Unit

By the end of this unit, students should be able to:

- Understand the **structure and syntax of HTML5**.
- Create well-structured web pages using **HTML tags, lists, links, images, and tables**.
- Apply **CSS** to style web pages effectively.
- Use **CSS selectors, properties, and the box model** for layout and design.
- Format text and apply styling to create visually appealing content.

3. Learning Outcomes

After completing this unit, students will be able to:

- Build **basic web pages** with HTML5.
- Incorporate **lists, links, images, tables, and multimedia content**.
- Apply **CSS styling rules** to control colors, fonts, spacing, and layout.
- Understand the role of **CSS box model, margins, padding, and borders** in design.
- Develop a foundation for **advanced front-end web development**.

4. Importance of Studying this Unit

- HTML5 and CSS are the **backbone of web development**; mastering them is essential for creating interactive and visually appealing websites.
- Provides the **foundation for learning JavaScript, React, and other front-end technologies**.
- Helps students understand **web standards and responsive design principles**.
- Enables **professional web design and development skills** for real-world applications.

5. Key Concepts

- **HTML5 Basics:** Structure of web pages, tags, attributes, elements, comments, doctype.
- **Text Formatting:** Headings, paragraphs, bold, italics, and other formatting tags.
- **Lists & Links:** Ordered/unordered lists, hyperlinks.
- **Images & Tables:** Adding images, tables, captions, and styling table content.
- **CSS Fundamentals:** Inline, internal, external CSS; selectors and properties.
- **CSS Styling:** Colors, backgrounds, fonts, text properties.
- **Box Model:** Margins, padding, borders, and content layout.
- **Page Layout & Design:** General styling principles, responsive design basics.

1) Introduction to HTML5

HTML5 is the fifth and current major version of HTML (HyperText Markup Language), the standard markup language for creating web pages and web applications. It was finalized and published as a W3C Recommendation in October 2014, succeeding HTML4 which was standardized in 1997.

Key Features of HTML5

1. **Semantic Elements**: HTML5 introduced new semantic tags that better describe the content:

   <header>, <footer>, <nav>, <article>, <section>, <aside>, <main>, <figure>, <figcaption>

2. **Multimedia Support**: Native support for audio and video without plugins:

   ```
   <video controls>
     <source src="movie.mp4" type="video/mp4">
   </video>


   <audio controls>
     <source src="audio.mp3" type="audio/mpeg">
   </audio>
   ```

3. **Canvas Element**: For drawing graphics via JavaScript:

   ```
   <canvas id="myCanvas" width="200" height="100"></canvas>
   ```

4. **Form Enhancements**: New input types and attributes:

   ```
   <input type="email" required>
   <input type="date">
   <input type="range" min="1" max="10">
   ```

5. **Local Storage**: Web Storage API for client-side data storage:

   ```
   localStorage.setItem('key', 'value');
   ```

6. **Geolocation API**: Access to user's geographical location.
7. **Responsive Design Support**: Better support for mobile devices and responsive layouts.

2) Browsers and HTML5

Browser Support for HTML5

Modern web browsers have excellent support for HTML5 features, though some differences exist between browsers. Here's how major browsers handle HTML5:

Current Browser Support (2026)

1. **Google Chrome**: Excellent HTML5 support, frequentlyupdated
2. **Mozilla Firefox**: Strong support, implements new HTML5 features quickly
3. **Apple Safari**: Good support (especiallyon macOS and iOS)
4. **Microsoft Edge**: Excellent support (Chromium-based version)
5. **Opera**: Good support (also Chromium-based)

3) HTML Editors: Offline and Online

HTML editors are tools used to write, edit, and manage HTML, CSS, and JavaScript code. They can be **offline (desktop-based)** or **online (web-based)**. Below is a comparison of popular options in both categories.

Offline HTML Editors (Desktop Applications)

These are installed on your computer and do not require an internet connection to work.

1. Visual Studio Code (VS Code)

- **Platform**: Windows, macOS, Linux
- **Features**:
  - Free and open-source
  - IntelliSense (smart code completion)
  - Built-in Git support

- Extensions for HTML5, CSS3, JavaScript
- Live Server extension for real-time preview

- **Best for**: Professional developers, beginners

2. Sublime Text

- **Platform**: Windows, macOS, Linux
- **Features**:
  - Lightweight and fast
  - Multiple cursors for quick editing
  - Customizable with plugins
  - Syntax highlighting for HTML5
- **Best for**: Fast coding, lightweight editing

2. Adobe Dreamweaver

- **Platform**: Windows, macOS
- **Features**:
  - WYSIWYG (What You See Is What You Get) editor
  - Live preview (design + code view)
  - Built-in FTP for publishing
  - Supports HTML5, CSS3, JavaScript
- **Best for**: Designers, beginners (paid software)

2. Notepad++

- **Platform**: Windows
- **Features**:

- Free and lightweight
- Syntax highlighting for HTML
- Supports multiple tabs

- **Best for**: Quick edits, beginners

5. Brackets (Discontinued but Still Usable)

- **Platform**: Windows, macOS, Linux
- **Features**:
  - Open-source (by Adobe)
  - Live Preview feature
  - Preprocessor support (LESS, SCSS)
- **Best for**: Front-end developers

4) **Online HTML Editors (Web-Based)**

These run in a browser and require an internet connection.

1. CodePen

- **Website**: https://codepen.io
- **Features**:
  - Live preview
  - Supports HTML5, CSS3, JavaScript
  - Community templates ("Pens")
  - Collaboration features
- **Best for**: Sharing code snippets, experimenting

1. JSFiddle

- **Website**: https://jsfiddle.net
- **Features**:
  - Simple interface
  - Supports frameworks (React, Vue, jQuery

5) What is the DOCTYPE?

The <!DOCTYPE> declaration is the very first thing in an HTML document, before the <html> tag. It's not an HTML element or tag - it's an instruction to the web browser about what version of HTML the page is written in.

HTML5 DOCTYPE

For HTML5, the declaration is very simple:

html

<!DOCTYPE html>

This is the shortest possible doctype and is case-insensitive (though lowercase is typically used).

Purpose of DOCTYPE

The DOCTYPE serves two main purposes:

1. **Triggers standards mode** in browsers (as opposed to quirks mode)
2. **Identifies the HTML version** being used

   Historical DOCTYPE Declarations

   Before HTML5, DOCTYPEs were much more complex:

   HTML 4.01 Strict

   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">

   XHTML 1.0 Transitional

   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

Key Features of HTML5 DOCTYPE

- Case-insensitive (<!doctype html> works the same)
- No reference to a DTD (Document Type Definition)
- Backwards compatible
- Supported by all modern browsers

What Happens Without a DOCTYPE?

If you omit the DOCTYPE:

- Browsers may render the page in "quirks mode"
- Layout and styling may appear inconsistent
- The page may not render as intended

Best Practices

1. Always include <!DOCTYPE html> as the very first line
2. Don't include any content before it (not even comments or whitespace)
3. Use the HTML5 version for all new projects

6) HTML Comments

HTML comments are used to add explanatorynotes or to temporarily disable code without deleting it. Comments are ignored by browsers and are not displayed on the webpage.

Basic Syntax

html

<!-- This is an HTML comment -->

Anything between <!-- and --> is treated as a comment.

Multi-line Comments

```
<!--
  This is a multi-line
  HTMLcomment that spans
  several lines
-->
```

HEADINGS

- ✓ Headings also helps to structure the document.
- ✓ Html offers six levels of headings which use the elements <h1>, <h2>, <h3>, <h4>, <h5>, <h6>.
- ✓ While browsers can display headings differently, they tend to display the <h1> element as the largest of the size and <h6> as the smallest
- ✓ General form:

<h1 [align=”left”/”right”/”center”]> </h1>

- ✓ Eg:

```
<html>
    <head></head>
    <body>
        <h1>Aidan’s Airplanes (h1)</h1>
        <h2>The best is used Airplanes (h2)</h2>
        <h3>we’ve got them by the hangar ful (h3)</h3>
        <h4>we’re the guys to see for a good used airplane (h4)</h4>
        <h5>we offer great prices on great planes(h5)</h5>
        <h6> Noreturns, no guarantees, no refunds, all sales are final(h6)</h6>
    </body>
</html>
```

PARAGRAPHS

- ✓ Paragraph element offers a way to structure the text in the web page.
- ✓ Each paragraph of text should go in between an opening and closing tag.
- ✓ General form of paragraph tag

  **<p [align="left"/"right"/"center"]> **
- ✓ when a browser displays a paragraph it usually inserts a new line before the next paragraph and adds a little bit of extra virtual space.
- ✓ Each paragraph can be aligned on the screen either to the left, the right or centered.
- ✓ Html processors ignore all white spaces in source documents except for spacing words.
- ✓ That is tabs, newlines, paragraphs are not formatted as we expect. In fact all these that are encountered in the source code get converted into a single space characters.
- ✓ Any space that are placed between the words will get converted into a single space in the displayed document.
- ✓ Escape sequences are used to display more than one space.

FORMATTING TEXT

- ✓ HTML text formatting refers to the use of specific HTML tags to modify the appearance and structure of text on a webpage.
- ✓ It allows you to style text in different ways, such as making it bold, italic, underlined, highlighted, or struck-through.
- ✓ Below are listed the formatting tags available in the latest version of HTML.

- `<b>` tag - Bold Text
- `<i>` tag - Italic Text
- `<u>` tag - Underlined Text
- `<strong>` tag - Strong Text
- `<em>` tag - Emphasized Text
- `<mark>` tag - Highlighted Text
- `<sup>` tag - Superscript Text
- `<sub>` tag - Subscript Text
- `<del>` tag - Deleted Text
- `<ins>` tag - Inserted Text
- `<big>` tag - Big Text
- `<small>` tag - Small Text

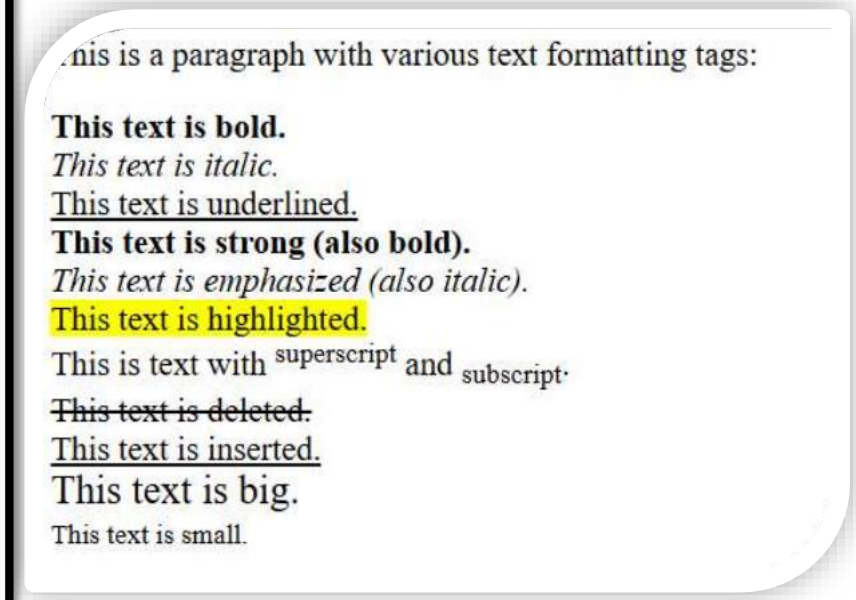
FORMATTING TEXT

```
<!DOCTYPE html>
<html>
<head>
<title>HTML Text Formatting Tags</title>
</head>
<body>

  <p>This is a paragraph with various text formatting tags:</p>

  <p>
    <b>This text is bold.</b><br>
    <i>This text is italic.</i><br>
    <u>This text is underlined.</u><br>
    <strong>This text is strong (also bold).</strong><br>
    <em>This text is emphasized (also italic).</em><br>
    <mark>This text is highlighted.</mark><br>
    This is text with <sup>superscript</sup> and <sub>subscript</sub>.<br>
    <del>This text is deleted.</del><br>
    <ins>This text is inserted.</ins><br>
    <big>This text is big.</big><br>
    <small>This text is small.</small>
  </p>

</body>
</html>
```



LISTS- UNORDERED LIST

- ✓ One of the most effective way of structuring a website or its contents is to use as List.
- ✓ In addition to the headings and paragraphs Lists help to organize and present information for easy reading.
- ✓ Itemized lists helps to highlight important points.
- ✓ HTML provides 3 types of lists:
  1) Unordered or Bulleted list
  2) Ordered or Numbered list
  3) Definition List
- ✓ The ordered and unordered lists are each made up of sets of list item

1) Unordered List:

- ✓ The basic unordered list has a bullet in front of each list item.
- ✓ Every thing between the tags must be encapsulated within <li> tags.
- ✓ Each item in the list must encoded between --

UNORDERED LIST

**Syntax: <ul [type= “disc”/”square”/”circle”][compact]> ..... **

- ✓ Type attribute sets the style of a list item.
- ✓ The bullet can be ***disc or square or circle***
  - ❑ Disc: causes each bullet to appear as a solid round disc.
  - ❑ Square: causes each bullet to appear as a solid square.
  - ❑ Circle: causes each bullet to appear as an empty circle.
- ✓ Compact attribute is used to minimize the amount of space that a list uses.
- ✓ Disc is the default bullet on some browsers and circle on some browsers.
- ✓ Type attribute will not work if it is written in upper case letters.
- ✓ Unordered lists may be nested inside same or other type of lists.
- ✓ A line space automatically is inserted before and after an unordered list except for a list nested written another list.

ORDERED LIST

2. Ordered List:

- ✓ An ordered list has a number instead of a bullet in front of each list item.
- ✓ Ordered lists are used to define a list whose items are in sequential, numerical order.
- ✓ General form:

**Syntax: <ol [type= “1”/”i”/”a”/”A”] [start=”n”][compact]> ……
**

- ❑ Type attribute sets the style of a list item ;
- ❑ “I” causes the items to be numbered as I, II, III.....
- ❑ “A” causes the items to be numbered as A, B, C.....
- ❑ “a” causes the items to be numbered as a, b, c.....
- ❑ “1” (default) causes the items to be numbered as 1, 2, 3.....
- ❑ “i” causes the items to be numbered as i, ii, iii.....
- ❑ Compact attribute to minimize the amount of space that a list uses.
- ❑ Start attribute specifies the starting number of the first item in an ordered

ORDERED AND UNORDERED LIST EXAMPLES

Unordered List

```
<html>
        <head></head>
        <body>
                <h1>List of PG Courses</h1>
                        <ul type = “disc”>
                                <li>MCA</li>
                                <li>MBA</li>
                                <li>M.Tech</li>
                        </ul>
        </body>
</html>
```

List of PG Courses

- MCA
- MBA
- M.Tech

Ordered List

```
<html>
        <head></head>
        <body>
                <h1>list of UG courses</h1>
                        <ol type = “1”>
                                <li>B.Tech(CSE)</li>
                                <li>B.Tech(ECE)</li>
                                <li>B.Tech(EEE)</li>
                                <li>B.Tech(civil)</li>
                                <li>B.Tech(Mech)</li>
                        </ol>
        </body>
</html>
```

list of UG courses

1. B.Tech(CSE)
2. B.Tech(ECE)
3. B.Tech(EEE)
4. B.Tech(civil)
5. B.Tech(Mech)

DEFINITION LIST

<dl> , <dt>, <dd> are the tags used to list definitions .
DL is used to provide a list of items with associated definitions.
Each item should be placed in a DT and its definition goes into DD directly following it.

<dl> tag:

Elements within a definition lists are either items being defined or their definitions.
General form

<dl [compact]> </dl>

<dt> tag:

Definition terms mark items whose definition will be provided by the next data definition.
General form

<dt> </dt>

<dd> tag:

Definitions of terms are enclosed within these tags .
The definition can include any text or block formatting text.
General form

<dd> …</dd>

DEFINITION LIST -EXAMPLE

Definition List

```
<html>
        <head>
                <title>definition list</title>
        </head>
        <body bgcolor="violet" text="maroon">
                <dl>
                        <dt><i>html</i></dt>
                                <dd>hyper text mark up language is language to design static web page</dd>
                        <dt><i>xml</i></dt>
                                <dd>extensible mark up language is a tool to define data for web applications </dd>
                        <dt><i>beans</i></dt>
                                <dd>it is a reusable software component that can be visually manipulated by any builder tool.</dd>
                        <dt><i>servlet</i></dt>
                                <dd> it is a server side technology</dd>
                        <dt><i>jsp</i></dt>
                                <dd>its also a server side technology</dd>
                </dl>
        </body>
</html>
```

html
: hyper text mark up language is language to design static web page

xml
: extensible mark up language is a tool to define data for web applications

beans
: it is a reusable software component that can be visually manipulated by any builder tool.

servlet
: it is a server side technology

jsp
: its also a server side technology

LINKS

- ✓ **HTML Links**, also known as **hyperlinks**, are defined by the **<a>** tag in HTML, which stands for "anchor."
- ✓ These links are essential for navigating between web pages and directing users to different sites, documents, or sections within the same page.
- ✓ The basic attributes of the <a> tag include **href**, **title**, and **target**, among others.
- ✓ **Basic Syntax of an HTML Link:**

```
<a href="https://www.example.com">Visit Example</a>
```

- ✓ **A Simple HTML link example**

```
<html>
        <head>
            <title>HTML Links</title>
        </head>
        <body>
            <p>Click on the following link</p>
            <a href="https://www.geeksforgeeks.org">GeeksforGeeks</a>
        </body>
</html>
```

LINKS

- ✓ By default, links will appear as follows in all browsers:
  - ❑ An unvisited link is underlined and blue.
  - ❑ A visited link is underlined and purple.
  - ❑ An active link is underlined and red.

HTML Links – Target Attribute

- ✓ The target attribute in the <a> tag specifies where to open the linked document. It controls whether the link opens in the same window, a new window, or a specific frame.

Attribute	Description
`_blank`	Opens the linked document in a new window or tab.
`_self`	Opens the linked document in the same frame or window as the link. (Default behavior)
`_parent`	Opens the linked document in the parent frame.
`_top`	Opens the linked document in the full body of the window.
`framename`	Opens the linked document in a specified frame. The frame's name is specified in the attribute.

LINKS

```
<html>

<head>
    <title>Target Attribute Example</title>
</head>

<body>
    <h3>
        Various options available in the
        Target Attribute
    </h3>

    <p>
        If you set the target attribute to
        "_blank", the link will open in a new
        browser window or tab.
    </p>
    <a href="https://www.geeksforgeeks.org"
       target="_blank">
        GeeksforGeeks
    </a>

    <p>
        If you set the target attribute to
        "_self", the link will open in the
        same window or tab.
    </p>
    <a href="https://www.geeksforgeeks.org"
       target="_self">
        GeeksforGeeks
    </a>

    <p>
        If you set the target attribute to
        "_top", the link will open in the full
        body of the window.
    </p>
    <a href="https://www.geeksforgeeks.org"
       target="_top">
        GeeksforGeeks
    </a>

    <p>
        If you set the target attribute to
        "_parent", the link will open in the
        parent frame.
    </p>
    <a href="https://www.geeksforgeeks.org"
       target="_parent">
        GeeksforGeeks
    </a>
</body>

</html>
```

Various options available in the Target Attribute

If you set the target attribute to "_blank", the link will open in a new browser window or tab.

GeeksforGeeks

If you set the target attribute to "_self", the link will open in the same window or tab.

GeeksforGeeks

If you set the target attribute to "_top", the link will open in the full body of the window.

GeeksforGeeks

If you set the target attribute to "_parent", the link will open in the parent frame.

GeeksforGeeks

IMAGES

- ✓ The **HTML <img> tag** is used to embed an image in web pages by linking them. It creates a placeholder for the image, defined by attributes like src, width, height, and alt, and does not require a closing tag.
- ✓ There are **two ways** to insert the images into a webpage:
  - ❑ By providing a full path or address (URL) to access an internet file.
  - ❑ By providing the file path relative to the location of the current web page file.

Attribute	Description
src	Specifies the path to the image file.
alt	Provides alternate text for the image, useful for accessibility and when the image cannot be displayed.
crossorigin	Allows importing images from third-party sites with cross-origin access, typically used with canvas.
height	Specifies the height of the image.
width	Specifies the width of the image.

IMAGES

```
<html>
    <body>
        <img src="https://media.geeksforgeeks.org/wp-content/cdn-uploads/20190710102234/download3.png" alt="GeeksforGeeks logo" width="300" height="300" />
    </body>
</html>
```

TABLES

- ✓ **HTML Tables** allow you to arrange data into rows and columns on a web page, making it easy to display information like schedules, statistics, or other structured data in a clear format.
- ✓ **What is an HTML Table?**
  - ❑ An HTML table is created using the <table> tag. Inside this tag, you use
    - ❖ ***<tr>*** to define table rows,
    - ❖ ***<th>*** for table headers, and
    - ❖ ***<td>*** for table data cells
  - ❑ Each <tr> represents a row, and within each row, <th> or <td> tags represent the cells in that row, which can contain text, images, lists, or even another table.

TABLES

✓ **Syntax of <table> tag**:

<table [align=”center”/”right”/”left”] [border=”n”]

[cellpadding=”n”] [width=”nn%”] [cellspacing=”n”]

[bgcolor=”colorname”]>

……

</table>

Attribute name	Description
align	Specifies alignment of a table
border	Specifies the table border
cellpadding	Specifies the space between the cell and the cell content
cellspacing	Specifies the space between cells
Width	Specifies the width of the tables

TABLES

<tr> tag: <tr> used to create row in the html

Syntax:

<tr [align="center"/"right"/"left"]
[valign="top"/"center"/"bottom"]
[bgcolor="colorname"]>
</tr>

Attribute name	Description
Align	aligns the content in a table row.
valign	Vertical aligns the content in a table row.
Bgcolor	Specifies a background color for a table row

TABLES

<th> tag: To create table header.

Syntax:

```
<th [align="center"/"right"/"left"]
    [valign="top"/"center"/"bottom"]
    [now rap][colspan="n"][rowspan="n"]
    [bgcolor="colorname"]>
</th>
```

Attribute name	Description
Align	aligns the content alignement in a header cell.
Valign	Vertical aligns the content in a header cell.
Bgcolor	Specifies a background color for a table row
Nowrap	Specifies that the content inside a header cell should not wrap
Colspan	Specifies the no. Of columns a header cell should span
Rowspan	Specifies the no. Of rows a header cell should span

TABLES

<td> tag: To create table cell

Syntax:

```
<td [align="center"/"right"/"left"]
    [colspan="n"][rowspan="n"]
    [bgcolor="colorname"]>
</td>
```

- The text in <th> elements are BOLD and centered by default.
- The text in <td> elements are Regular and left-aligned by default.

TABLES

Example:
Design following format using Table tags.

SNO	SNAME	WP_MARKS
Mca53	Rohith	90
Mca54	Gowthami	80

```
<html>
        <head></head>
        <body>
            <table border="2"      width="50%" align="center"
              Cellpadding="2"     cellspacing="4" bgcolor="pink">
            <tr>
                    <th>SNO</th>
                    <th>SNAME</th>
                    <th>WP_MARKS</th>
            </tr>
            <tr>
                    <th>Mca53</th>
                    <th>Rohith</th>
                    <th>90</th>
            </tr>
            <tr>
                    <th>Mca54</th>
                    <th>Gowthami</th>
                    <th>80</th>
            </tr>
```

TABLES

```
<html>
        <body>
                <table border=4 width="40%">
                <tr>
                        <thcolspan=4>maruthi</th>
                </tr>
                <tr>
                        <td>omnivan</td>
                        <td>200000</td>
                </tr>
                <tr>
                        <td>maruthi 800</td>
                        <td>242000</td>
                </tr>
                <tr>
                        <td>maruthi 1000</td>
                        <td>310000</td>
                </tr>
                <tr>
                        <td>maruthizen</td>
                        <td>390000</td>
                </tr>
                <tr>
                        <thcolspan=2>tata</th>
                </tr>
                <tr>
                        <td>sumo</td>
                        <td>475000</td>
                </tr>
                <tr>
                        <td>estate</td>
                        <td>462000</td>
                </tr>
                <tr>
                        <thcolspan=2>ambassador</th>
                </tr>
                <tr>
                        <td>petrol</td>
                        <td>324000</td>
                </tr>
                <tr>
                        <td>diesel</td>
                        <td>387000</td>
                </tr>
        </body>
</html>
```

maruthi	
omnivan	200000
maruthi 800	242000
maruthi 1000	310000
maruthizen	390000
tata	
sumo	475000
sierra	447000
estate	462000
ambassador	
petrol	324000
diesel	387000

INTRODUCTION TO CSS

- ✓ CSS stands for Cascading Style Sheets.
- ✓ It is a stylesheet language used to describe the presentation of a document written in a markup language like HTML or XML.
- ✓ As HTML is used to structure the content of a webpage, and CSS as the styling that makes it visually appealing.
- ✓ The Stylesheets of the elements can be cascaded (overwritten) in 3 ways
    1) Inline
    2) Internal and
    3) External

INTRODUCTION TO CSS

1) Inline CSS:

- ✓ This involves applying styles directly to individual HTML elements using the style attribute.
- ✓ The CSS rules are written within the opening tag of the element.
- ✓ Example:

```
<p style="color: blue; font-size: 16px;">
  This line is an inline-styled paragraph.
</p>
```

INTRODUCTION TO CSS

2) Internal/Embedded CSS:

- ✓ This involves embedding CSS rules directly within the <head> section of your HTML document using the <style> tag.
- ✓ The CSS rules apply to all the HTML elements within that specific document.
- ✓ Example:

```
<html>
<head>
    <style>
        h1 {
            color: green;
            text-align: center;
        }
        p {
            font-family: Arial, sans-serif;
        }
    </style>
</head>
<body>
    <h1>This is a green heading.</h1>
    <p>This paragraph will use the Arial font.</p>
</body>
</html>
```

INTRODUCTION TO CSS

3) External CSS

- ✓ Writing CSS rules in one or more separate External files with a .css extension.
- ✓ These CSS files are then linked to your HTML documents using the <link> tag within the <head> section.
- ✓ Example:

```
<!-- HTML file -->
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <p>This line is styled using external CSS.</p>
</body>
```

```
/* styles.css */
p { color: red; font-size: 20px; }
```

COMPONENTS TO CSS

- ✓ **CSS works by associating rules with HTML elements. A CSS rule contains two main parts:**
  - **a selector which specifies the HTML element(s) to style.**
  - **a declaration block which contains one or more declarations separated by semicolons.**
- ✓ **Each declaration includes a property name and a value, specifying the aspect of the element's presentation to control.**

COMPONENTS TO CSS

```
<head>
    <title>CSS Tutorial</title>
    <style>
        h1 {
            color: #36CFFF;
        }

        p {
            font-size: 1.5em;
            color: white;
        }

        div {
            border: 5px inset gold;
            background-color: black;
            width: 300px;
            text-align: center;
        }
    </style>
</head>
<body>
    <div>
        <h1>Hello World!</h1>
        <p>This is a sample CSS code.</p>
    </div>
</body>
```

In the above CSS snippet:

- **h1, p,** and **div** are the selectors that target the **<h1>, <p>**, and ** ** elements.
- **color, font-size, border, background-color, width,** and **text-align** are the properties.
- **#36CFFF, 1.5em, white, 5px inset gold, black, 300px,** and **center** are the corresponding values passed to these properties.

CSS COLORS AND BACKGROUND

CSS BCOLOR

1) color: Sets the color of an element.

color : default | color | hexvalue | rgbvalue

transparent (default):

- ✓ The background is transparent, showing whatever is behind the element.

<color>:

- ✓ CSS provides a set of predefined color names (keywords) that you can use directly
- ✓ Any valid CSS color value, such as: Named colors: red, blue, green, black, white, etc.

CSS COLOR

Hexadecimal values:

- ✓ Hexadecimal color codes are a way to represent colors using a six-digit hexadecimal number (base-16).
- ✓ The format is #RRGGBB, where RR represents the red component, GG represents the green component, and BB represents the blue component. Each component is 1 a two-digit hexadecimal value ranging from 00 (lowest intensity) to FF (highest intensity).
- ✓ #ff0000 (red), #00ff00 (green), #0000ff (blue), etc.

RGB values:

- ✓ RGB (Red, Green, Blue) colors are defined using the rgb() function, which takes three integer values (0-255) representing the intensity of red, green, and blue light, respectively.
- ✓ rgb(255, 0, 0), rgb(0, 255, 0), rgb(0, 0, 255), etc.

CSS COLOR

```
<html>
        <head>
                <style>
                        h1
                        {
                                color:rgb(255,0,0);
                        }
                        p
                        {
                                color:#00FF00
                        }
                </style>
        </head>
        <body>
                <h1> To illustrate the font color of the element</h1>
                <p> This Example is to illustrate the CSS color properties and its values </p>
        </body>
</html>
```

CSS BACKGROUND-COLOR

1) Background-color: Sets the background color of an element.

Background-color : default | color | hexvalue | rgbvalue

transparent (default):

- ✓ The background is transparent, showing whatever is behind the element.

<color>:

- ✓ CSS provides a set of predefined color names (keywords) that you can use directly
- ✓ Any valid CSS color value, such as: Named colors: red, blue, green, black, white, etc.

CSS BACKGROUND-COLOR

Hexadecimal values:

- ✓ Hexadecimal color codes are a way to represent colors using a six-digit hexadecimal number (base-16).
- ✓ The format is #RRGGBB, where RR represents the red component, GG represents the green component, and BB represents the blue component. Each component is 1 a two-digit hexadecimal value ranging from 00 (lowest intensity) to FF (highest intensity).
- ✓ #ff0000 (red), #00ff00 (green), #0000ff (blue), etc.

RGB values:

- ✓ RGB (Red, Green, Blue) colors are defined using the rgb() function, which takes three integer values (0-255) representing the intensity of red, green, and blue light, respectively.
- ✓ rgb(255, 0, 0), rgb(0, 255, 0), rgb(0, 0, 255), etc.

CSS BACKGROUND-COLOR

```
<html>
<head>

    <style>
        h1 {
            background-color: blue;
        }
    </style>

</head>
<body>
    <h1>Geeksforgeeks</h1>
</body>
</html>
```

CSS BACKGROUND-IMAGE AND BACKGROUND-REPEAT

2) background-image: Sets one or more background images for an element.

background-image : none | url

- ✓ **none (default): No background image is displayed.**
- ✓ **url("path/to/image.jpg"): Specifies the path to an image file. The path can be absolute or relative.**

3) background-repeat: controls how a background image can be controlled .

background-repeat : repeat | no-repeat | repeat-x | repeat-y

- ✓ **repeat (default): The image is repeated both horizontally and vertically to cover the entire background.**
- ✓ **repeat-x: The image is repeated only horizontally.**
- ✓ **repeat-y: The image is repeated only vertically.**
- ✓ **no-repeat: The image is not repeated; it is displayed only once.**

CSS BACKGROUND-IMAGE AND BACKGROUND-REPEAT

```
<html>
<head>

    <style>
        body {
            background-image:
                url(
            "https://media.geeksforgeeks.org/wp-content/cdn-uploads/20190417124305/250.png");
            background-repeat: repeat-x;
        }
    </style>

</head>
<body>
    <h1>"Hello world"</h1>
</body>
</html>
```

CSS TEXT AND FONT PROPERTIES

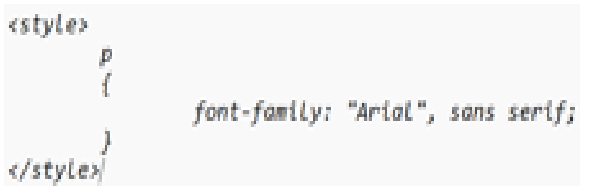
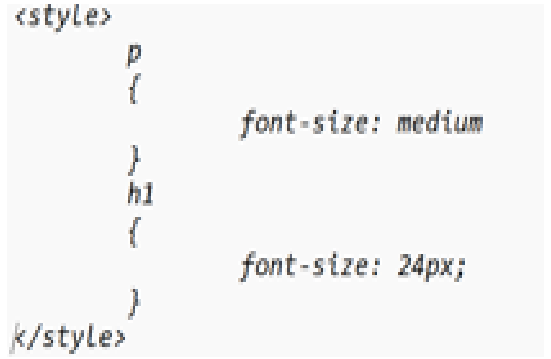
CSS TEXT

CSS Text : CSS text properties are used to style and format the text content of HTML elements. They control various aspects of the text's appearance, such as color, alignment, decoration, spacing, and transformation.			
text-align	Specifies the horizontal alignment of text within an element	text-align : **left|right|center|justify**	
text-decoration-line	it specifies the type of the line the text decoration will have	text-decoration-line: **none|underline|overline|line-through**	**text-decoration-style: solid** This line has a solid underline. This line has a solid line-through. This line has a solid overline.
text-decoration-style	it specifies the type of line drawn as a text-decoration such as solid, dashed,dotted or double	text-decoration-style: **solid|double|dashed|dotted**	**text-decoration-style: double** This line has a double underline. This line has a double line-through. This line has a double overline.
text-decoration-color	it sets the color of the text-decoration line	text-decoration-color: **color**	**text-decoration-color: color;** underline line-through overline Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks! Hello Geeks!
text-transform	is used to change the case of a text	text-transform : **none|capitalize|uppercase|lowercase**	**text-transform: uppercase:** GEEKSFORGEEKS IT IS A COMPUTER SCIENCE PORTAL FOR GEEKS.

CSS TEXT

```
<html>
        <head>
                <style>
                        p
                        {
                                text-decoration-style:double;
                                text-decoration-line:overline;
                                text-decoration-color:blue;
                                text-align:center;
                                text-transform:uppercase;
                        }
                </style>
        </head>
        <body>
                <p> This Example is to illustrate the CSS Text properties and its values </p>
        </body>
</html>
```

CSS FONT

CSS font : CSS fonts control how text appears on a webpage.			
font-family	is used to set the font face of the text	font-family: **family-name | generic-family**	
font-size	adjusts the size of the text on the webpage	font-size: **predefined keyword|length** **Predined keywords are small|medium|largre**	
font-style	defines the style of the font, typically italic or normal.	font-style : **normal|italic**	
font-weight	Sets the boldness or thickness of the font.	font-weight : **normal|bold|bolder|lghter**	

CSS FONT

```
<html>
        <head>
                <style>
                        p
                        {
                                font-family :"arial",sansserif;
                                font-size:16pt;
                                font-style:italic;
                                font-weight:bold
                        }
                </style>
        </head>
        <body>
                <p> This Example is to illustrate the CSS font properties and its values </p>
        </body>
</html>
```

This Example is to illustrate the CSS font properties and its values

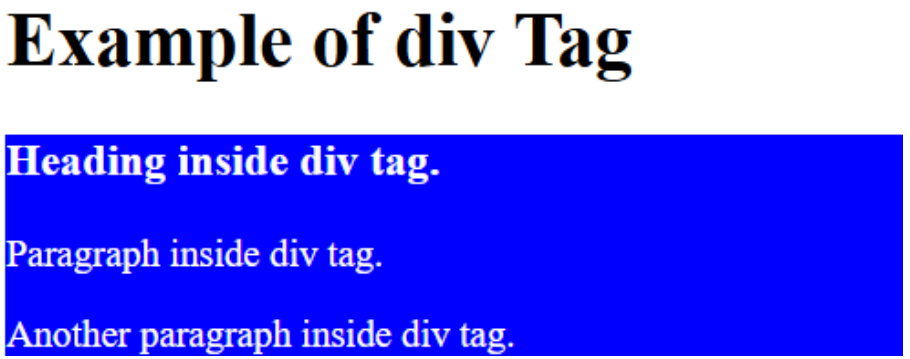
CSS BOX MODEL

DIV TAG

- ✓ **The HTML tag is used to define sections in web pages.**
- ✓ **Developers use this tag to group HTML elements, allowing them to apply CSS styles to multiple elements continuously.**
- ✓ `Syntax : <div> ..... </div>`
- ✓ **This HTML code shows grouping a heading and two paragraphs within a section of a webpage using the tag.**

```
<html>
        <head>
                <style>
                        div
                        {
                                color:white;
                                background-color:blue;

                        }
                </style>
        </head>
        <body>
                <h1>Example of div Tag</h1>
                <div>
                      <h3>Heading inside div tag.</h3>
                      <p>Paragraph inside div tag.</p>
                      <p>Another paragraph inside div tag.</p>
                </div>
        </body>
</html>
```



```html
<html lang="en">
<head>
    <title>HTML div tag</title>
    <style>
        .first {
            width: 100px;
            height: 100px;
            background-color: rgb(4, 109, 109);
            text-align: center;
            display: grid;
            place-items: center;
            float: left;
        }
        .second {
            width: 100px;
            height: 100px;
            background-color: rgb(17, 92, 222);
            text-align: center;
            display: grid;
            place-items: center;
            float: left;
        }
        .third {
            width: 100px;
            height: 100px;
            background-color: rgb(82, 40, 180);
            text-align: center;
            display: grid;
            place-items: center;
            float: left;
        }
        .fourth {
            width: 100px;
            height: 100px;
            background-color: rgb(157, 17, 222);
            text-align: center;
            display: grid;
            place-items: center;
            float: left;
        }

        div {
            border-radius: 10px;
            margin: 10px 10px;
        }

        div p {
            color: white;
        }
    </style>
</head>
<body>
    <h3>HTML div Tag Example</h3>
    <!-- Using HTML div tag -->
    <div class="first">
        <p>First</p>
    </div>
    <div class="second">
        <p>Second</p>
    </div>
    <div class="third">
        <p>Third</p>
    </div>
    <div class="fourth">
        <p>Fourth</p>
    </div>
</body>
</html>
```

HTML div Tag Example

SPAN TAG

- ✓ **The HTML tag is used to define css to an inline element or apply style th part of the text**
- ✓ Syntax : .....

```
<html>
        <head>
                <style>
                        span
                        {
                                background-color:green;
                                color:yellow
                        }
                </style>
        </head>
        <body>
                <p> this is about span tag <span> an inline element</span></p>
        </body>
</html>
```

this is about span tag an inline element

BLOCK LEVEL VS INLINE LEVEL

1. Block-Level Elements

- ✓ **Block-level elements take up the full width of the container, starting on a new line. These elements are primarily used to structure large sections of content.**

Examples:

- ✓ ** : A container for HTML elements that doesn't affect the content's appearance.**
- ✓ **<p>: Defines a paragraph of text.**
- ✓ **<h1> to <h6>: Used for headings.**
- ✓ **<section>, <article>, <footer>, etc.: Semantic tags used to organize content.**

Characteristics:

- ✓ **Always starts on a new line.**
- ✓ **Occupies the full width available (by default).**
- ✓ **Can contain other block-level and inline elements.**

BLOCK LEVEL VS INLINE LEVEL

2. Inline Elements

- ✓ **Inline elements only take up as much width as their content and stay within the flow of the surrounding text.**
- ✓ **These are great for styling parts of text or embedding small elements within blocks of content.**

Examples:

- ✓ **: A generic container for inline elements or text.**
- ✓ **<a>: Defines a hyperlink.**
- ✓ **<img>: Embeds an image.**
- ✓ **, ,
, etc.: Used for formatting text.**

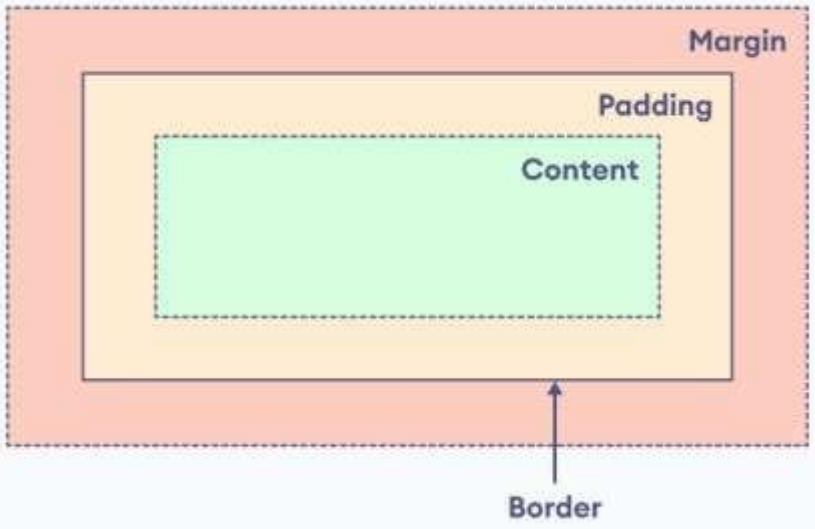
Characteristics:

- ✓ **Does not start on a new line.**
- ✓ **Takes up only as much width as needed.**
- ✓ **Can only contain other inline elements or text, not block-level elements.**

```
<p>This is a paragraph with an <a href="#">inline link</a> to another page.</p>
```

CSS BOX MODEL

- ✓ **The box model treats every HTML element as a rectangular box consisting of content, padding, border, and margin.**
- ✓ **The box model defines the layout of an HTML element in the following way:**



The components of the box model are:

- `content` : actual text or image that is displayed in the element
- `padding` : transparent space between the content and the border of an element
- `border` : line that surrounds the padding and content within the element
- `margin` : transparent area added outside the border

The primary purpose of the box model is to explain how the dimensions and spacing of elements are calculated and how they relate to each other.

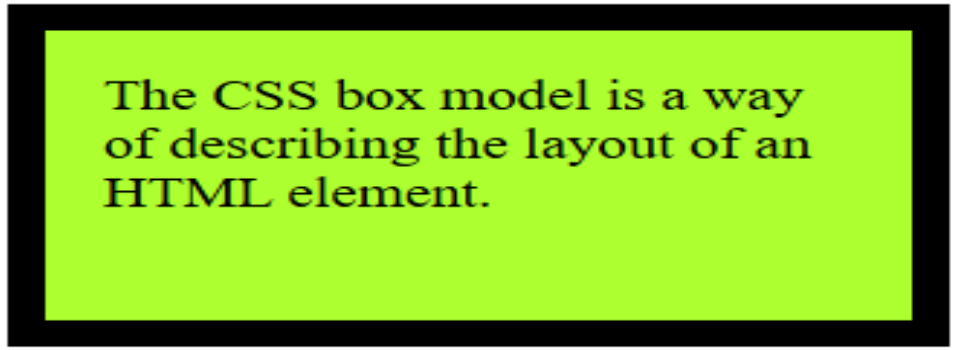
CSS BOX MODEL

- ✓ **Box Model can be applied to both Block level and inline elements .**
- ✓ **While applying for block level element, we can specify height, width, top and bottom , left, right**
- ✓ **Where as with inline , even we specify, it will ignore**
- ✓ **To make it work ,** the display property should be changed to either inline-block or block**.**

CSS BOX MODEL

```
<html>
        <head>
                <style>
                        div
                        {
                            width: 200px;
                            height: 80px;
                            border: 10px solid black;
                            padding: 15px;
                            background-color: greenyellow;
                        }
                </style>
        </head>
        <body>
                <div>
                    The CSS box model is a way of describing the layout of an HTML element.
                </div>
        </body>

</html>
```



CSS BOX MODEL

```
<html>
        <head>
                <style>
                        span
                        {
                            width: 10px;
                            height: 10px;
                            border: 5px solid black;
                            padding: 5px;
                            background-color: greenyellow;
                        }
                </style>
        </head>
        <body>
                <p>The CSS box model is a way of describing the layout of an <span>HTML element</span></p>
        </body>

</html>
```

The CSS box model is a way of describing the layout of an HTML element

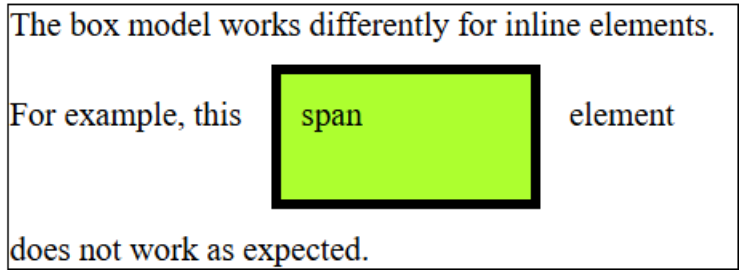
CSS BOX MODEL

- ✓ In the above example,
  - The height and width are ignored.
  - The top and bottom margins are also ignored, and only the left and right margins work.
  - The padding works for all four sides of the elements.
- ✓ Hence, we cannot calculate the actual dimensions of the inline element.
- ✓ In order to apply the box model in the inline elements, the display property should be changed to either inline-block or block.
- ✓ Let's see an example with inline-block,

CSS BOX MODEL

```
<html>
        <head>
                <style>
                        p
                        {
                            width: 350px;
                            border: 1px solid black;
                        }
                        span
                        {
                            display: inline-block;
                            width: 100px;
                            height: 40px;
                            border: 5px solid black;
                            padding: 10px;
                            margin: 10px;
                            background-color: greenyellow;
                        }
                </style>
        </head>
        <body>
        <p>
            The box model works differently for inline elements. For example,
            this <span>span</span> element does not work as expected.
        </p>
    </body>

</html>
```



- ✓ CSS stands for **Cascading Style Sheets**.
- ✓ It is a stylesheet language used to describe the presentation of a document written in a markup language like HTML or XML.
- ✓ As HTML is used to structure the content of a webpage, and CSS as the styling that makes it visually appealing.

Types of CSS

1) Inline CSS:
   - ✓ This involves applying styles directly to individual HTML elements using the style attribute.
   - ✓ The CSS rules are written within the opening tag of the element.
   - ✓ Example:

```
<p style="color: blue; font-size: 16px;">
  This line is an inline-styled paragraph.
</p>
```

2) Internal/Embedded CSS:
   - ✓ This involves embedding CSS rules directly within the <head> section of your HTML document using the <style> tag.
   - ✓ The CSS rules apply to all the HTML elements within that specific document.
   - ✓ **Example:**

```html
<html>
<head>
    <style>
        h1 {
            color: green;
            text-align: center;
        }
        p {
            font-family: Arial, sans-serif;
        }
    </style>
</head>
<body>
    <h1>This is a green heading.</h1>
    <p>This paragraph will use the Arial font.</p>
</body>
</html>
```

3) External CSS

- ✓ Writing CSS rules in one or more separate External files with a `.css` extension.
- ✓ These CSS files are then linked to your HTML documents using the `<link>` tag within the `<head>` section.
- ✓ **Example:**

```html
<!-- HTML file -->
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <p>This line is styled using external CSS.</p>
</body>
```

```css
/* styles.css */
p { color: red; font-size: 20px; }
```

Components of CSS

- ✓ **CSS works by associating rules with HTML elements. A CSS rule contains two main parts:**
  - **a selector which specifies the HTML element(s) to style.**
  - **a declaration block which contains one or more declarations separated by semicolons.**
- ✓ **Each declaration includes a property name and a value, specifying the aspect of the element's presentation to control.**

```
<html>
<head>
   <title>CSS Tutorial</title>
   <style>
      h1 {
         color: #36CFFF;
      }

      p {
         font-size: 1.5em;
         color: white;
      }

      div {
         border: 5px inset gold;
         background-color: black;
         width: 300px;
         text-align: center;
      }
   </style>
</head>
<body>
   <div>
      <h1>Hello World!</h1>
      <p>This is a sample CSS code.</p>
   </div>
</body>
</html>
```

In the above CSS snippet:

- **h1, p,** and **div** are the selectors that target the **<h1>, <p>**, and ** ** elements.
- **color, font-size, border, background-color, width,** and **text-align** are the properties.
- **#36CFFF, 1.5em, white, 5px inset gold, black, 300px,** and **center** are the corresponding values passed to these properties.

1. background Properties and Their Values:

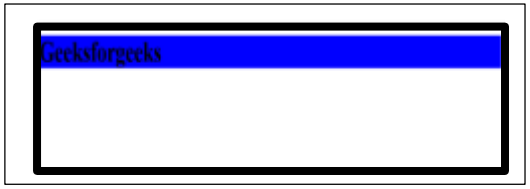
1.1. background-color: Sets the background color of an element.

- ✓ **Values:**
  - transparent (default): The background is transparent, showing whatever is behind the element.
  - <color>: Any valid CSS color value, such as: Named colors: red, blue, green, black, white, etc.
  - Hexadecimal values: #ff0000 (red), #00ff00 (green), #0000ff (blue), etc.
  - RGB values: rgb(255, 0, 0), rgb(0, 255, 0), rgb(0, 0, 255), etc.

```
<head>

    <style>
        h1 {
            background-color: blue;
        }
    </style>

</head>
<body>
    <h1>Geeksforgeeks</h1>
</body>
</html>
```



2) background-image: Sets one or more background images for an element.

- ✓ **Values:**
  - none (default): No background image is displayed.
  - url("path/to/image.jpg"): Specifies the path to an image file. The path can be absolute or relative.

3) background-repeat: Controls how a background image will be repeated.

- ✓ **Values:**
  - **repeat (default):** The image is repeated both horizontally and vertically to cover the entire background.
  - **repeat-x:** The image is repeated only horizontally.
  - **repeat-y:** The image is repeated only vertically.
  - **no-repeat:** The image is not repeated; it is displayed only once.

```
<html>
<head>

    <style>
        body {
            background-image:
                url(
            "https://media.geeksforgeeks.org/wp-content/cdn-uploads/20190417124305/250.png");
            background-repeat: repeat-x;
        }
    </style>

*
</head>
<body>
    <h1>"Hello world"</h1>
</body>
</html>
```

2. Text Properties and Their Values:

CSS text properties are used to style and format the text content of HTML elements. They control various aspects of the text's appearance, such as color, alignment, decoration, spacing, and transformation.

text-decoration-line	It specifies the type of line the text decoration will have.	`text-decoration-line: none | underline | overline | line-through`
text-decoration-style	It specifies the type of line drawn as a text decoration such as solid, wavy, dotted, dashed, or double.	`text-decoration-style: solid|double|dotted|dashed|wavy`
text-decoration-color	It sets the color of the text-decoration line.	`text-decoration-color:color`

```html
<html>

<head>
    <title>
        CSS text-decoration-style property
    </title>

    <!-- CSS style -->
    <style>
        p {
            text-decoration-style: solid;
        }

        .GFG1 {
            text-decoration-line: underline;
        }

        .GFG2 {
            text-decoration-line: line-through;
        }

        .GFG3 {
            text-decoration-line: overline;
        }
    </style>
</head>

<body>
    <h1 style="color: green">
        GeeksforGeeks
    </h1>

    <b>text-decoration-style: solid</b>

    <p class="GFG1">
        This line has a solid underline.
    </p>
    <p class="GFG2">
        This line has a solid line-through.
    </p>
    <p class="GFG3">
        This line has a solid overline.
    </p>
</body>

</html>
```

CSS text-transform property is used to change the case of a text. For example,

```
p {
    text-transform: capitalize;
}
```

CSS Text Transform Syntax

The syntax of the text-transform property is,

text-transform: none | capitalize | uppercase | lowercase

Here,

- **none - doesn't change the text, default value**
- **capitalize - sets the first character of each word to uppercase**
- **uppercase - sets all characters in the uppercase**
- **lowercase - sets all characters in the lowercase**

CSS fonts control how text appears on a webpage. With CSS, you can specify various properties like font family, size, weight, style, and line height to create visually appealing and readable typography

- ✓ Key Properties of CSS Fonts
- ✓ To customize fonts effectively in web design, it’s crucial to understand the main CSS font properties:
    - font-family: Specifies the font type.
    - font-size: Determines the size of the text.
    - font-weight: Adjusts the thickness of the text.
    - font-style: Controls the slant of the text (e.g., italic).
    - line-height: Sets the space between lines of text.

- letter-spacing: Modifies the space between characters.
- text-transform: Controls the capitalization of text.

CSS font-family property is used to set the font face of the text on the webpage

CSS Font Family Types

Font families are divided into two types:

- **Generic family: Refers to the category of the fonts that share similar design characteristics. For example, Serif, sans-serif, Cursive, etc.**
- **Font family: Refers to the specific font family name like Helvetica, Arial, Georgia, etc.**

- The syntax of the `font-family` property is as follows:

```
font-family: family-name | generic-family
```

Here,

- **family-name: refers to the specific font family like Arial, Helvetica, etc**
- **generic-family: refers to the broader category of font families with similar design characteristics like serif, sans-serif, etc**

- Let's see an example,

```
h1   {
    font-family: "Source Sans Pro", "Arial", sans-serif;
}
```

- In the above example, the browser will first try to render `Source Sans Pro`. If it is not available, the browser will try to render `Arial`. And if it is also not available, the browser will finally use a font from the `sans-serif` family.

CSS font-size property adjusts the size of the text on the webpage.

The font-size property has the following syntax,

font-size: predefined keyword|length|initial|inherit;

Here,

- **predefined keyword: Refers to the keyword that has predetermined font-size like small, medium, large, etc.**
- **length: Refers to the font-size using a specific length unit like px, em or points.like 24px, 2em, etc.**

CSS font-weight is used to adjust the lightness or boldness of the text in a webpage.

The font-weight property has the following syntax,

font-weight: normal|bold|bolder|lighter|number|initial|inherit;

The fundamental difference between block-level and inline elements in HTML lies in how they behave in terms of line breaks and width:

Block-Level Elements:

- **Start on a new line: They always begin on a new line in the document flow.**
- **Occupy the full width available: By default, they expand to fill the entire width of their parent container.**
- **Create "blocks" of content: They essentially create visual blocks on the page.**
- **Can contain both block-level and inline elements: This allows for hierarchical structuring of content.**
- **Respect width and height properties: You can explicitly set the width and height of block-level elements using CSS.**

- **Respect margin and padding on all four sides: You can control the spacing around and within block-level elements using margin-top, margin-right, margin-bottom, margin-left, and padding-top, padding-right, padding-bottom, padding-left.**

Common Block-Level Elements:

- ** **
- **<p>**
- **<h1> to <h6> (headings)**
- **<ul>, <ol>, <li> (lists)**
- **<form>**
- **<header>**
- **<footer>**
- **<section>**
- **<article>**
- **<nav>**

Inline Elements:

- **Do not start on a new line: They flow along with the surrounding content on the same line.**
- **Only take up the width necessary: They only occupy the space required to display their content.**
- **Do not force line breaks: Elements before and after an inline element will stay on the same line.**
- **Can only contain other inline elements and text: Placing block-level elements inside inline elements is generally considered semantically incorrect.**
- **width and height properties do not apply: You cannot directly set the width and height of inline elements using CSS. Their dimensions are determined by their content.**

- **Only horizontal margin and padding are respected: margin-left and margin-right and padding-left and padding-right will work, but margin-top, margin-bottom, padding-top, and padding-bottom may not affect the flow of surrounding inline elements as expected.**

Common Inline Elements:

- ****
- **<a> (anchor/link)**
- **<img> (image)**
- ** (strong emphasis)**
- ** (emphasis)**
- **
 (line break)**
- **<input>**
- **<textarea>**
- **<button>**
- **<code>**

Analogy:

Think of block-level elements like paragraphs in a book – each starts on a new line and occupies the full width of the page. Inline elements are like individual words or phrases within a sentence – they flow together without forcing new lines.

Key Differences Summarized:

Feature	**Block-Level Elements**	**Inline Elements**
Line Breaks	**Start on a new line**	**Do not start on a new line**
Width	**Occupy full width**	**Only take necessary width**

Height	**Respect height property**	**height property does not apply**
Width	**Respect width property**	**width property does not apply**
Vertical Margin/Padding	**Respected on all sides**	**Not fully respected (can cause unexpected behavior)**
Content	**Can contain block and inline elements**	**Should only contain inline elements/text**

CASE STUDY 1: College Website Homepage

Scenario:

A college wants a simple homepage with:

- College name as heading
- About paragraph
- Courses list
- Link to contact page
- College image
- Table showing departments

Questions:

1. Create HTML structure using proper **tags, elements, and attributes**
2. Add **lists for courses**
3. Insert **image and links**
4. Create a **table for departments**
5. Apply CSS:
    - Background color
    - Font styles
    - Box model (margin, padding, border)

CASE STUDY 2: Online Shopping Product Page

Scenario:

Design a product page showing:

- Product name
- Product image
- Description
- Price
- “Buy Now” link

Questions:

1. Use **headings, paragraphs, and formatting tags**
2. Insert image using <img>
3. Create a link using <a>
4. Apply CSS:
    - Text color and font
    - Background styling
    - Add border using **box model**

CASE STUDY 3: Student Registration Form Page

Scenario:

Create a webpage for student details display:

- Name, Age, Course
- List of hobbies
- Table of marks

Questions:

1. Use **lists (ordered/unordered)**
2. Create **table with rows and columns**
3. Add comments in HTML
4. Apply CSS:
    - Padding and margins
    - Background color
    - Font styling

CASE STUDY 4: News Article Page

Scenario:

Design a news article webpage:

- Title
- Article content
- Highlight important text
- Add related links

Questions:

1. Use:
    - Headings (<h1>)
    - Paragraphs (<p>)
    - Formatting tags (<b>, <i>, etc.)
2. Add links
3. Apply CSS:
    - Text alignment
    - Font properties
    - Background colors

CASE STUDY 5: Personal Profile Page

Scenario:

Create your personal profile webpage:

- Name
- Photo
- Skills (list)
- Education (table)

Questions:

1. Structure HTML using proper **doctype and elements**
2. Add **image and lists**
3. Create **table for education**
4. Apply CSS:
    - Colors
    - Fonts
    - Borders
    - Margins & padding

UNIT- I: OVERVIEW OF HTML AND CSS:

HTML5: Introduction to HTML5, Browsers and HTML5, Editor's Offline and Online, Tags, Attribute and Elements, Doctype Element, Comments, Headings, Paragraphs, and Formatting Text, Lists and Links, Images and Tables CSS: Introduction CSS, Applying CSS to HTML5, Selectors, Properties and Values, CSS Colors and Backgrounds, CSS Box Model, CSS Margins, Padding, and Borders, CSS Text and Font Properties, CSS General Topics

PART -A

Q. No.	Questions	Blooms Taxonomy Level
1	What is HTML5?	L1
2	List any two new features of HTML5.	L1
3	What is the purpose of the <!DOCTYPE> declaration?	L1
4	Define HTML elements with an example.	L1
5	Differentiate between HTML tags and attributes.	L2
6	What is the role of the <head> tag in HTML?	L1
7	Write syntax to insert an image in HTML5.	L3
8	Define CSS and its use in web development.	L1
9	What is the box model in CSS?	L2
10	Write the syntax to apply an internal CSS.	L3

PART –B

1	Explain the structure of an HTML5 document with example.	L2
2	Describe various types of HTML lists and how to create them.	L2
3	Write an HTML code to create a table with row and column spans.	L3
4	Discuss different types of CSS with examples.	L2
5	Explain the CSS box model with a neat diagram.	L2
6	How are margins, borders, and padding applied in CSS? Explain with code.	L3
7	Write HTML and CSS code to design a simple web page.	L3
8	Explain the different types of CSS selectors with examples.	L2
9.	What are the advantages of using CSS in web development?	L2
10.	Differentiate between inline, internal, and external CSS with examples.	L2

FULL STACK WEB DEVELOPMENT

UNIT –II

OVERVIEW OF JAVASCRIPT

JavaScript is the duct tape of the Internet.

— Charlie Campbell

Unit -II: OVERVIEW OF JAVASCRIPT

1. Unit Overview

This unit introduces **JavaScript**, the core scripting language for web development, covering syntax, variables, data types, operators, objects, arrays, functions, control structures, and interacting with the **Document Object Model (DOM)** and **window object**. Students learn to add dynamic behavior to web pages using both internal and external JavaScript.

2. Objectives of the Unit

By the end of this unit, students should be able to:

- Understand the **basics of JavaScript syntax** and structure.
- Apply JavaScript code to HTML pages using **internal and external scripts**.
- Work with **variables, operators, data types, and type conversion**.
- Manipulate **strings, numbers, dates, and arrays** effectively.
- Implement **conditional statements, loops, switch cases, and functions**.
- Access and manipulate web page content using **Document and Window objects**.

3. Learning Outcomes

After completing this unit, students will be able to:

- Write **interactive and dynamic web pages** using JavaScript.
- Use JavaScript to **perform calculations, string operations, and date manipulations**.
- Control program flow with **if-else, switch, and loops**.
- Develop and use **functions** to modularize code.
- Access and manipulate web page elements and browser properties using the **DOM and Window object**.

4. Importance of Studying this Unit

- JavaScript is **essential for client-side web development**; it adds interactivity and dynamic behavior to static HTML/CSS pages.
- Understanding JavaScript enables students to **develop web applications and front-end frameworks** (like ReactJS).
- Provides a foundation for **server-side programming with Node.js**.
- Essential for **modern web technologies, web apps, and interactive UI design**.

5. Key Concepts

- **Introduction to JavaScript:** History, role in web development, client-side scripting.
- **Applying JavaScript:** Internal (within <script> tags) and external (.js file) scripts.
- **JS Syntax:** Statements, semicolons, comments, identifiers.
- **Document & Window Object:** DOM manipulation, browser window control, accessing elements.
- **Variables and Operators:** var, let, const, arithmetic, logical, comparison operators.

- **Data Types & Type Conversion:** Numbers, strings, booleans, null, undefined; type casting.
- **Math and String Manipulation:** Built-in Math object, string methods.
- **Objects & Arrays:** Creating, accessing, and modifying objects and arrays.
- **Date and Time:** Date object methods, time manipulation.
- **Conditional Statements:** if, else if, else, ternary operator.
- **Switch Case:** Decision making using switch.
- **Looping in JS:** for, while, do...while, for...in, for...of.
- **Functions:** Defining, calling, parameters, return values, scope.

INTRODUCTION TO JAVASCRIPT

1.INTRODUCTION TO JAVA SCRIPT

- ✓ JavaScript is a scripting language designed primarily for adding interactivity to webpage.
- ✓ JavaScript developed by Net Scape Navigator in 1995 as a method for validating forms and provide interactive content to website.
- ✓ JavaScript is a scripting language used to make web pages interactive.
- ✓ JavaScript code can be embedded in HTML pages and interpreted by the web browser (or client)

APPLYING JAVASCRIPT(INTERNAL AND EXTERNAL)

2.APPLYING JAVASCRIPT (INTERNAL AND EXTERNAL)

- ✓ Java Script code can be written in 2 ways
  1) Internal Java Script
  2) External Java Script
  3) When to Use Internal and External JavaScript Code?

1) Internal Java Script

Placing JavaScript code directly into <script> tags within an HTML file is known as internal JavaScript

Eg:

```
<html>
        <body>
                <script>
                        var a=parseInt(prompt("enter a value"))
                        var b=parseInt(prompt("enter b value"))
                        var c=a+b
                        document.writeln(c)
                </script>
        </body>
</html>
```

APPLYING JAVASCRIPT(INTERNAL AND EXTERNAL)

2) External Java Script

External JavaScript refers to the practice of storing your JavaScript code in a separate file with a .js extension and then linking that file to your HTML document.

Eg :

first.js

```
a=parseInt(prompt("enter a value"))
b=parseInt(prompt("enter b value"))
c=a+b
document.writeln(c)
```

Sample.html

```
<html>
        <body>
                <script src =".\first.js"></script>
        </body>
</html>
```

APPLYING JAVASCRIPT(INTERNAL AND EXTERNAL)

3) When to Use Internal and External JavaScript Code?

If you have only a few lines of code that is specific to a particular webpage, then it is better to keep your JavaScript code internally within your HTML document.

On the other hand, if your JavaScript code is used in many web pages, then you should consider keeping your code in a separate file. In that case, if you wish to make some changes to your code, you just have to change only one file which makes code maintenance easy. If your code is too long, then also it is better to keep it in a separate file. This helps in easy debugging.

INTRODUCTION TO DOCUMENT AND WINDOW OBJECT

4.INTRODUCTION TO DOCUMENT AND WINDOW OBJECT,

- ✓ A javascript object is a complex variable with properties and methods.
- ✓ In javascript, almost everything is an object. Even primitive datatypes can be treated as objects.
- ✓ Most of the objects are prebuilt i.e it comes with the browser.
- ✓ Some of the built-in objects of javascript are:

4.1. Document object

4.2 Window object

4.3 Form object

4.4 Date object

4.5 Browser object

INTRODUCTION TO DOCUMENT AND WINDOW OBJECT

4.1 Document Object

- ✓ The document object represents a web page that is loaded in the browser.
- ✓ By accessing the document object, we can access the element in the HTML page. With the help of document objects, we can add dynamic content to our web page.
- ✓ The document object can be accessed with a window. document or just document.
- ✓ Some of the document object properties and methods are:

1. body: It returns the contents of the body element.
2. anchors: It returns all <a> elements that have a name attribute.
3. forms: It returns all the elements of the form.
4. links: It returns all <area> and <a> elements that have a href attribute.
5. Close, write , write ln etc., are methods

INTRODUCTION TO DOCUMENT AND WINDOW OBJECT

```
<html>
<head>
        <title>document.anchors Example</title>
</head>
<body bgcolor = "pink' color = "blue">

        <a name="First Hyperlink" href = "first.html">Section 1</a><br><br>
        <a name="Second Hyperlink" href = "second.html">Section 2</a><br><br>
        <a name="Third Hyperlink" href = "third.html">Section 3</a><br><br>
        <script>
                document.body.style.backgroundColor = "lightblue";
                document.body.style.color = "red";

                document.writeln("</br>"+document.anchors[0].name);
                document.writeln("</br>"+document.anchors[1].name);
                document.writeln("</br></br>"+document.anchors[2].name);

                document.writeln("</br>"+document.links[0].href);
                document.writeln("</br>"+document.links[1].href);
                document.writeln("</br>"+document.links[2].href);

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

INTRODUCTION TO DOCUMENT AND WINDOW OBJECT

4.2 Window Object

- ✓ Window object to create and manipulate a window on webpage.
- ✓ Two important methods of window object are
  1. open( )
  2. close( )

open: method to open a new window. General form of open( ) is:

open("url", "name", "attribute");

- ✓ url is the path of the file to be opened in new window.
- ✓ name is the name of the new window.
- ✓ attribute to control the look of the opened window with.

Window attribute	Description
Full screen	Specifies if window should be opened in full screen mode.
Height	Specifies the height of the window in pixels
Left	Specifies the x co-ordinate of the window in pixels
Menu bar[1/0]	Specifies if the menu bar of the window should be included.
Resizable [1/0]	Specifies if window should be resizable
Scrollbars[1/0]	Specifies if window should contain scrollbars
Status[1/0]	Specifies if the status bar of the window should be included
Toolbar[1/0]	Specifies if the toolbar of the window should be included.
Width	Specifies the width of the window.

VARIABLES AND OPERATORS

5.VARIABLES AND OPEARATORS

5.1 Variable

- ✓ Variable names
- ✓ Creating variables

Variable names:

A variable is a named location in memory that is used to hold a value that can be modified by the program.

There are strict rules governing how to name variables in javascript

- ✓ Names must begin with a letter, digit or underscore.
- ✓ You cannot use spaces in names.
- ✓ Names are case-sensitive.
- ✓ You cannot use a reserved word as a variable name

3.4) OPERATORS

In java script Operators are symbols that are used to perform some operations on the operands.

Combination of operands and operators are known as **Expressions.**

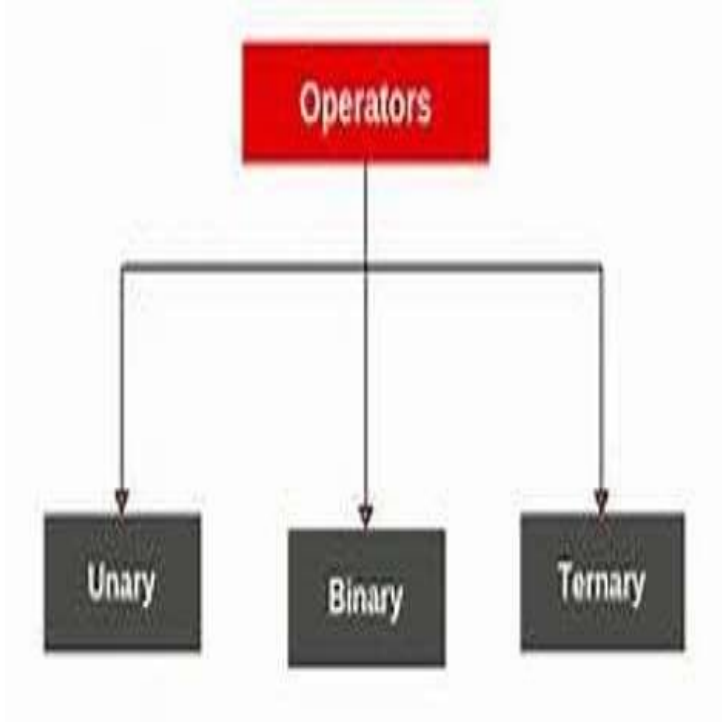
Java provides, a rich set of operators to manipulate the variables. **There are three types of operators in java**.

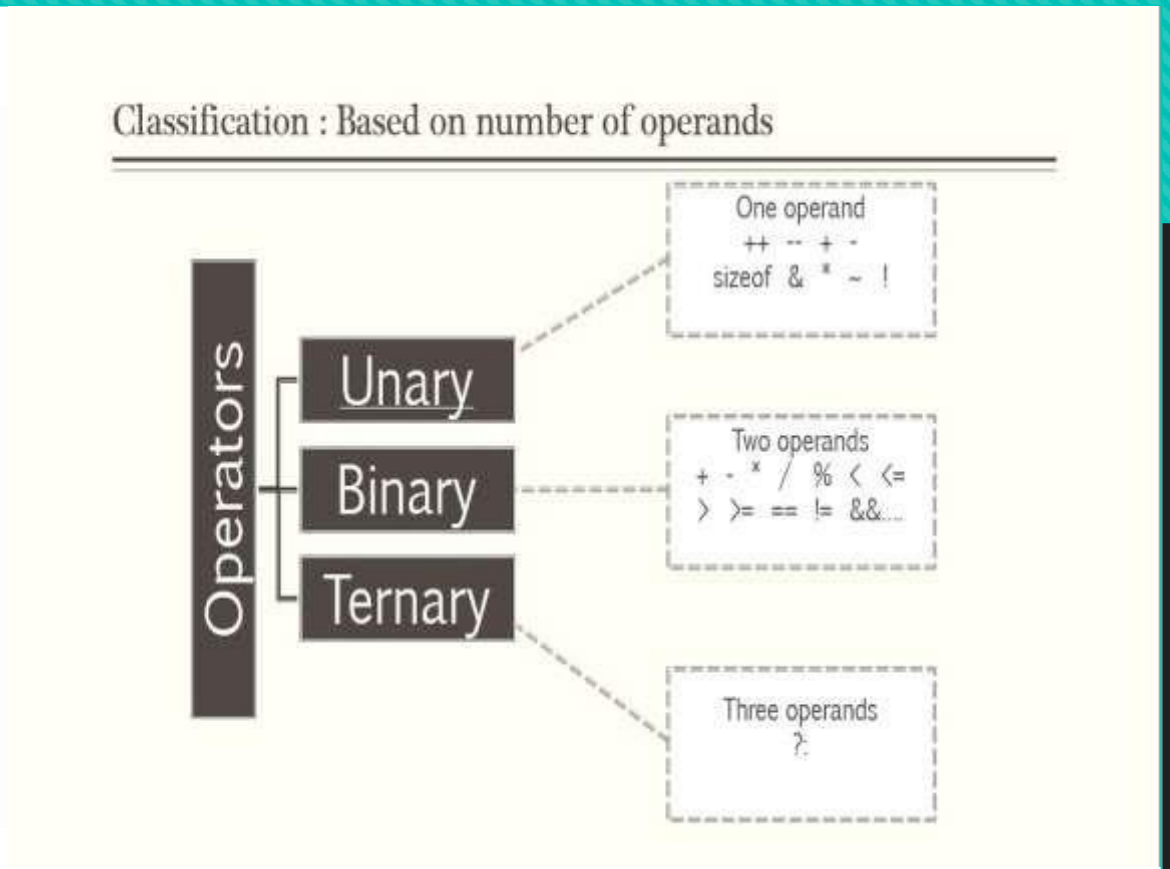
1. **Unary operators**

2. **Binary operators**

3. **Ternary operators**

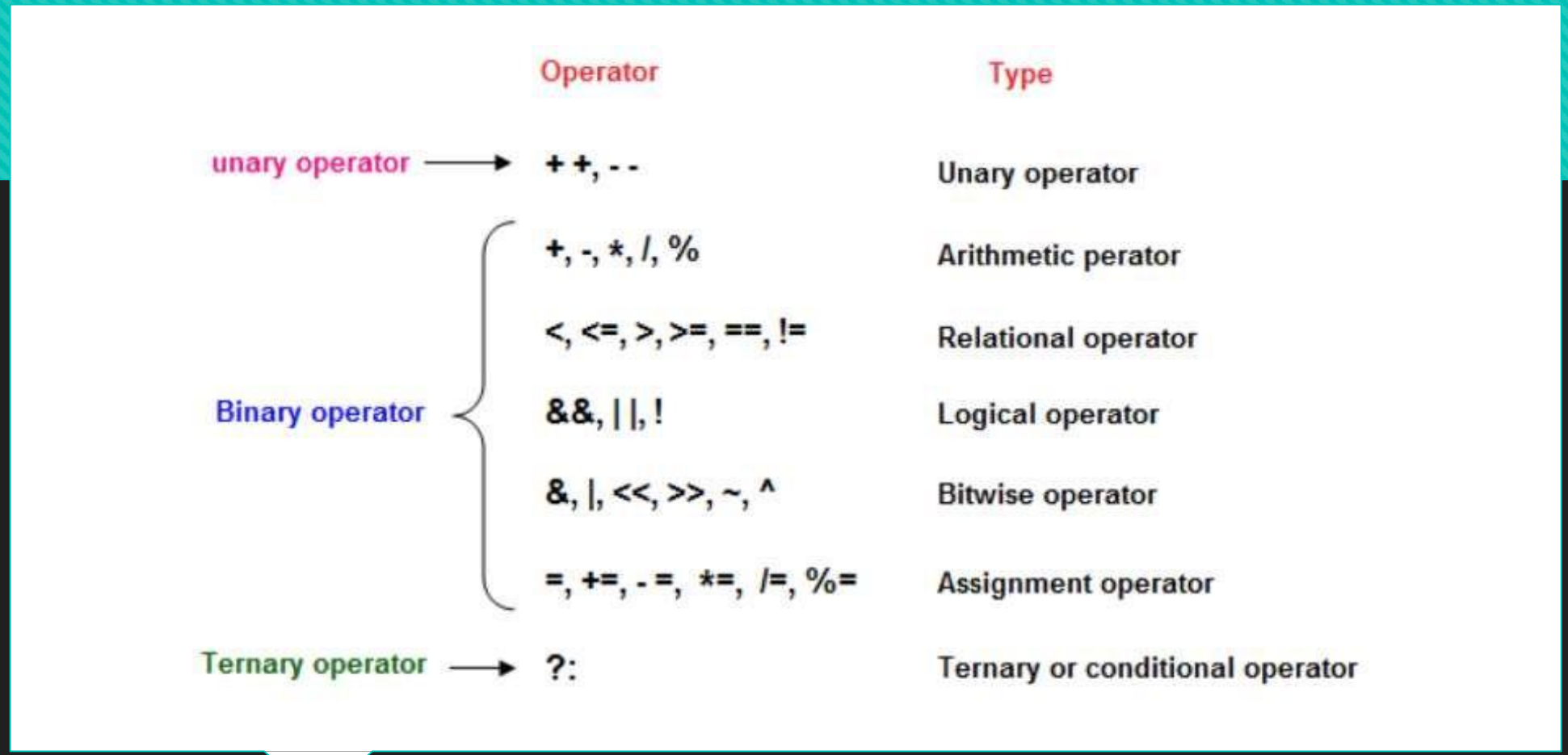
OPERAORS





Types of Operators

OPERATORS

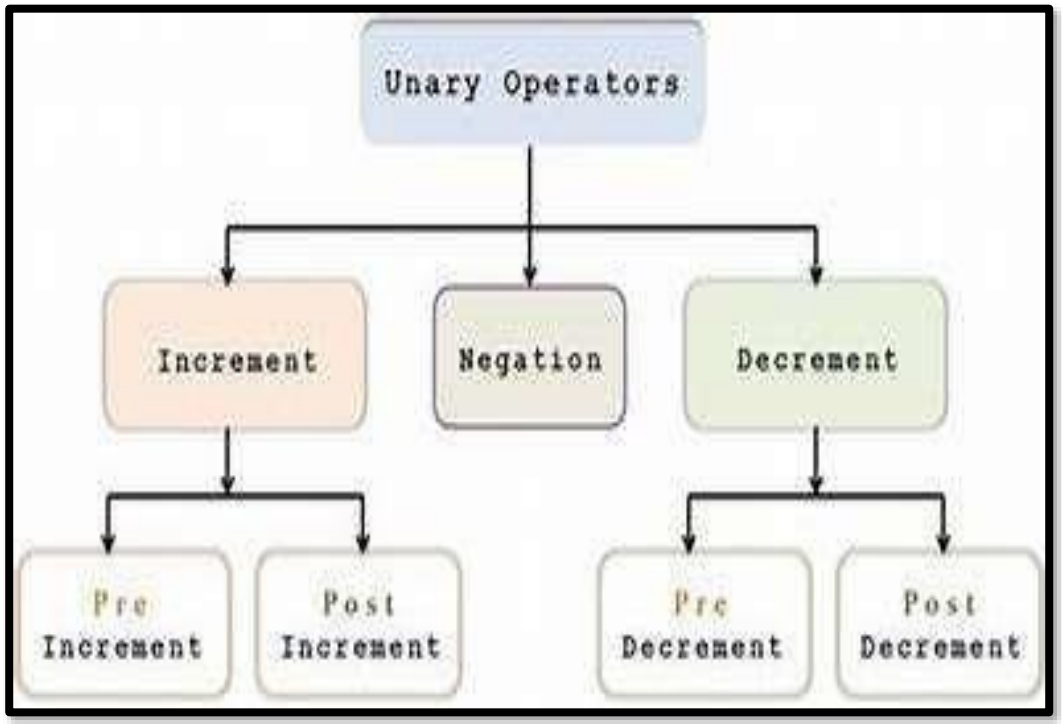

Operator
Type
unary operator
+ +, - -
Unary operator
Binary operator
+, -, *, /, %
Arithmetic perator
<, <=, >, >=, ==, !=
Relational operator
&&, | |, !
Logical operator
&, |, <<, >>, ~, ^
Bitwise operator
=, +=, - =, *=, /=, %=
Assignment operator
Ternary operator
?:
Ternary or conditional operator

UNARY OPERATOR

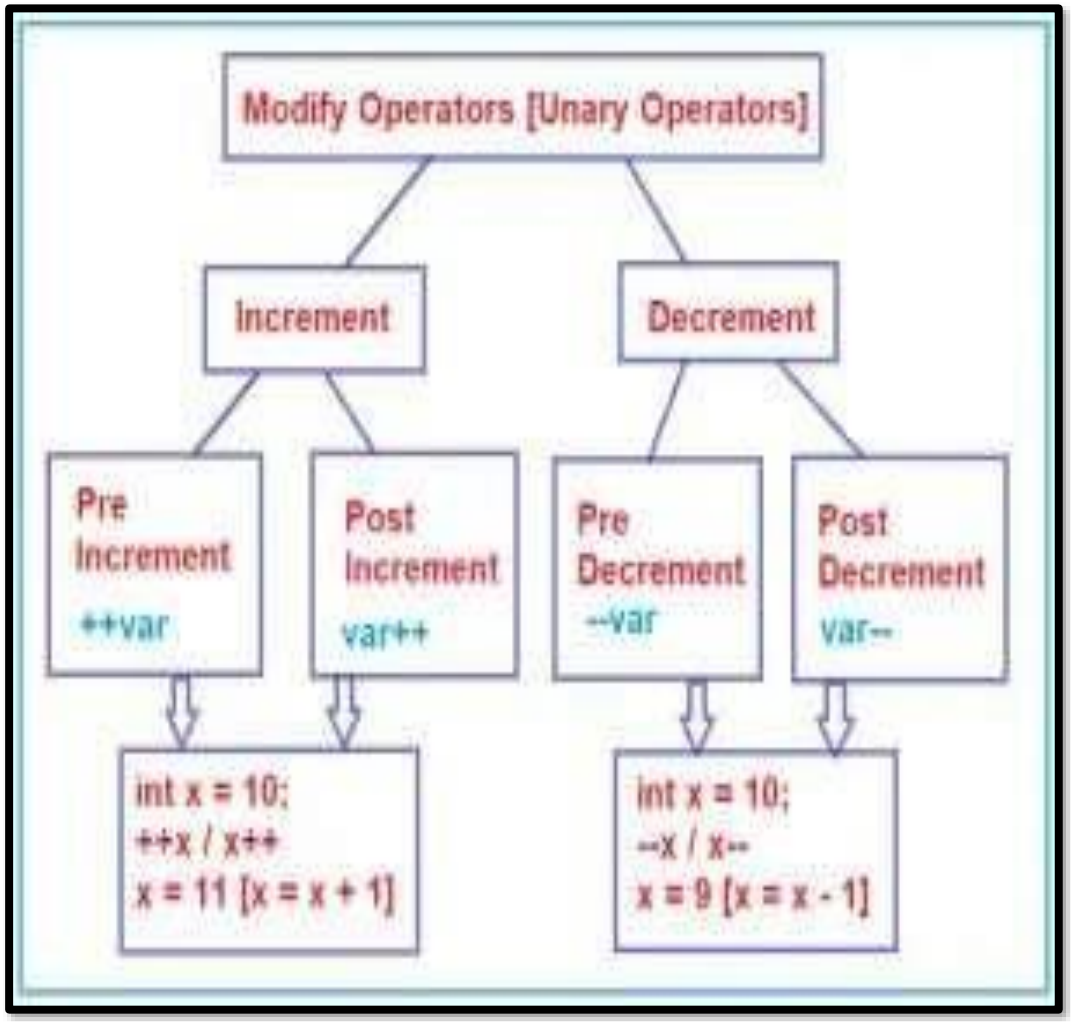
Operator that operates on single operand is called Unary Operator

Java supports 2 Unary Operators

1) **Increment Operator(++)**
2) **Decrement Operator (--)**



UNARY OPERATORS



- Pre Increment and Post Increment increases the variable value by 1
- Pre Decrement and Post Decrement decreases the variable value by 1
- Both pre and post increment does nt show difference if it is not used in an Expression.
- It shows difference only if it is used in expression

UNARY OPERATORS

- In the Pre-Increment, value is first incremented and then used inside the expression.
- In the Post-Increment, value is first used in an expression and then incremented.

Values before operations	Expression	Values after operations
a = 1	b = a++;	b = 1, a = 2
a = 1	b = ++a;	b = 2, a = 2
a = 1	b = a--;	b = 1, a = 0
a = 1	b = --a;	b = 0, a = 0
a = 1	b = 8 - ++a;	b = 6, a = 2
a = 1, c = 5	b = a++ + --c;	b = 5, a = 2, c = 4
a = 1, c = 5	b = ++a - c--	b = -3, a = 2, c = 4

BINARY OPERATORS – ARITHMETIC OPERATORS

- In Java, arithmetic operators are used to perform various mathematical operations on numerical data types, such as integers and floating-point numbers.
- These operators allow you to perform addition, subtraction, multiplication, division, and more

Arithmetic Operators

Operator	Meaning	Example
+	Addition	4 + 7 ⟶ 11
-	Subtraction	12 - 5 ⟶ 7
*	Multiplication	6 * 6 ⟶ 36
/	Division	30 / 5 ⟶ 6
%	Modulus	10 % 4 ⟶ 2
//	Quotient	18 // 5 ⟶ 3
**	Exponent	3 ** 5 ⟶ 243

BINARY OPERATORS – RELATIONAL OPERATORS

- The Relational Operators are used to determine whether 2 numbers are equal, or if one is greater or less than the other.
- Every Relational Statement evaluates to either True or False

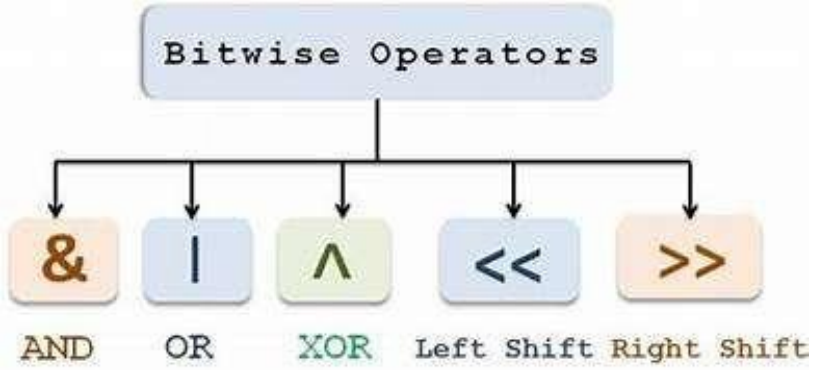
Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

BINARY OPERATORS – LOGICAL OPERATORS

- The Java Logical Operators work on the Boolean operand.
- It's also called Boolean logical operators.
- It operates on two Boolean values, which return Boolean values as a result.

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0



binary Operators – bitwise operators

BINARY OPERATORS – BITWISE OPERATORS

Bitwise AND Operator (&)

- Bitwise binary AND operator is denoted by the symbol
- It returns 1 if and only if both bits are 1, else returns 0.

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR Operator (|)

- Bitwise binary OR operator is denoted by the symbol |.
- It returns 1 if either of the bit is 1, else returns 0.

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise XOR Operator (^)

- Bitwise binary XOR operator is denoted by the symbol
- It returns 0 if both bits are the same, else returns 1..

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

BINARY OPERATORS – BITWISE OPERATORS

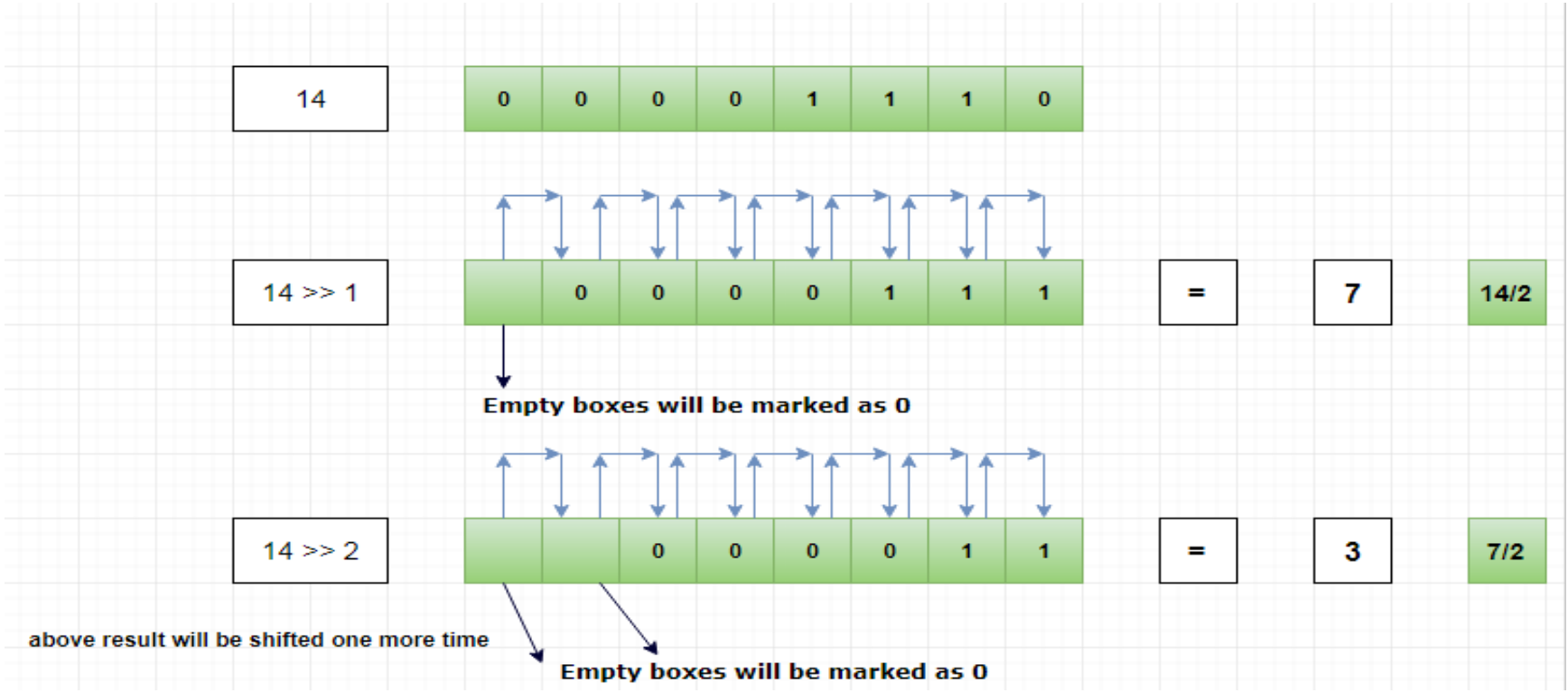
Bitwise NOT Operator (~)

X	~x
0	1
1	0

- Bitwise binary NOT operator is denoted by the symbol ~.
- It returns the inverse or complement of the bit. It makes every 0 a 1 and every

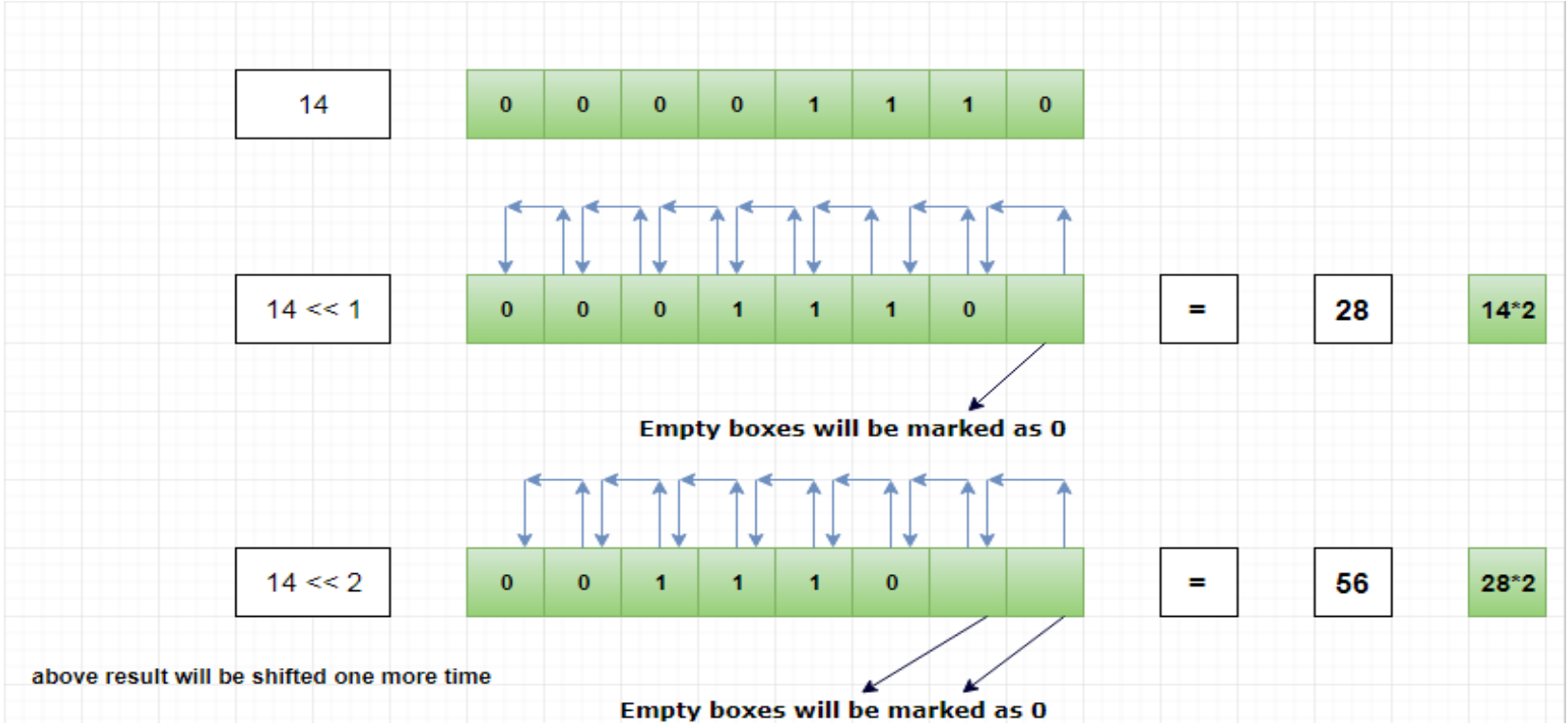
Operator	Symbol	Description
Signed Left Shift	<<	Shifts all the bits to the left by a specified number. For example, num<<2, will shift the bits of num to the left by two positions.
Signed Right Shift	>>	Shifts all the bits to the right by a specified number. For example, num>>2, will shift the bits of num to the right by two positions.
Unsigned Right Shift	>>>	Same as Signed Right Shift except that it fills the vacant leftmost positions with 0's instead of sign bit

BINARY OPERATORS – BITWISE OPERATORS – RIGHT SHIFT OPERATOR



Each Right Shift is equivalent to division by 2

BINARY OPERATORS – BITWISE OPERATORS – LEFT SHIFT OPERATOR



Each left Shift is equivalent to Multiply by 2

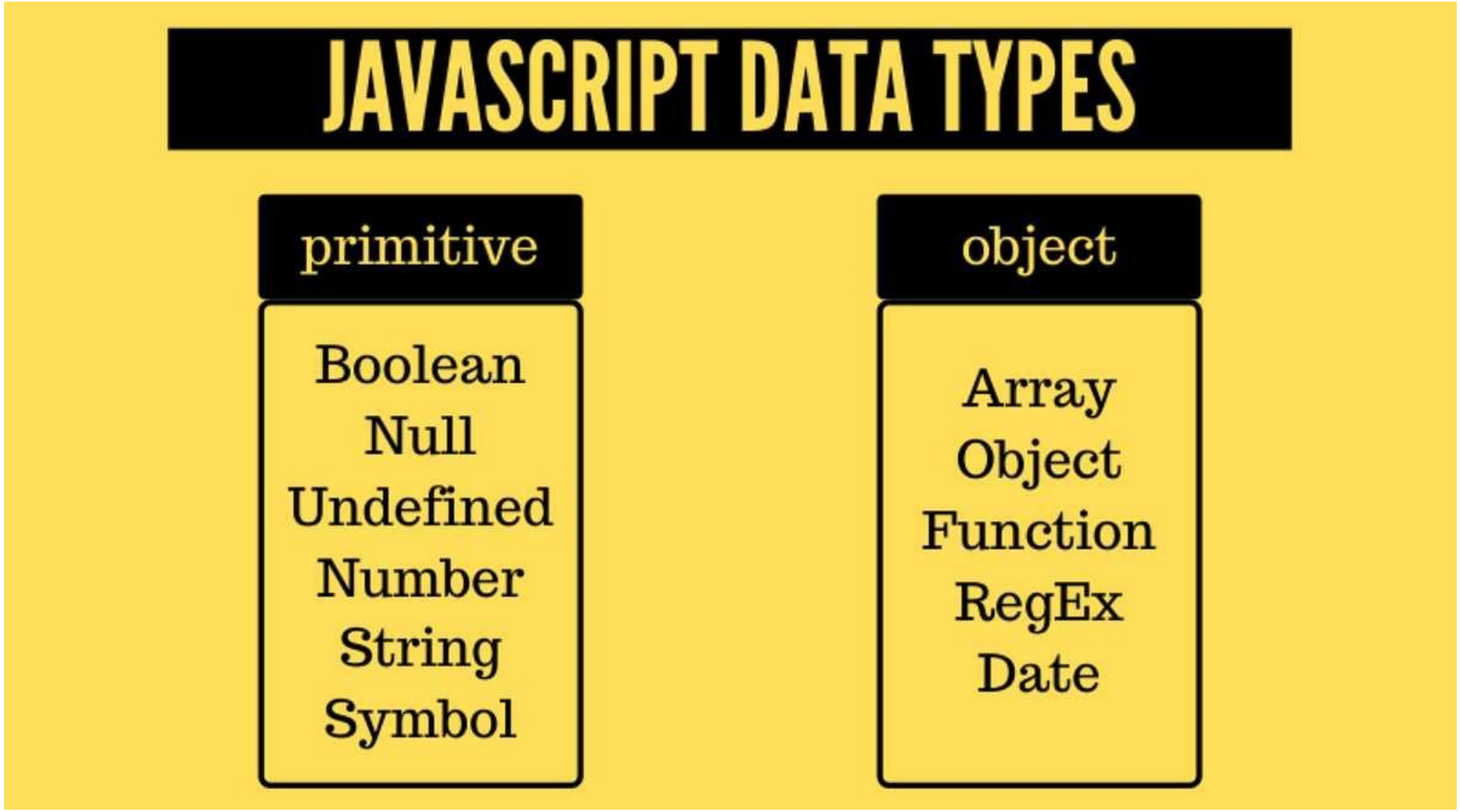
BINARY OPERATORS – ASSIGNMENT /COMPOUND ASSIGNMENT OPERATORS

Assignment operator	Sample expression	Explanation	Assigns
กำหนด: int a = 3, b = 5, c = 4, d = 6, e = 12			
+=	a += 7	a = a + 7	10 ให้ a
-=	b -= 4	b = b - 4	1 ให้ b
*=	c *= 5	c = c * 5	20 ให้ c
/=	d /= 3	d = d / 3	2 ให้ d
%=	e %= 9	e = e % 9	3 ให้ e

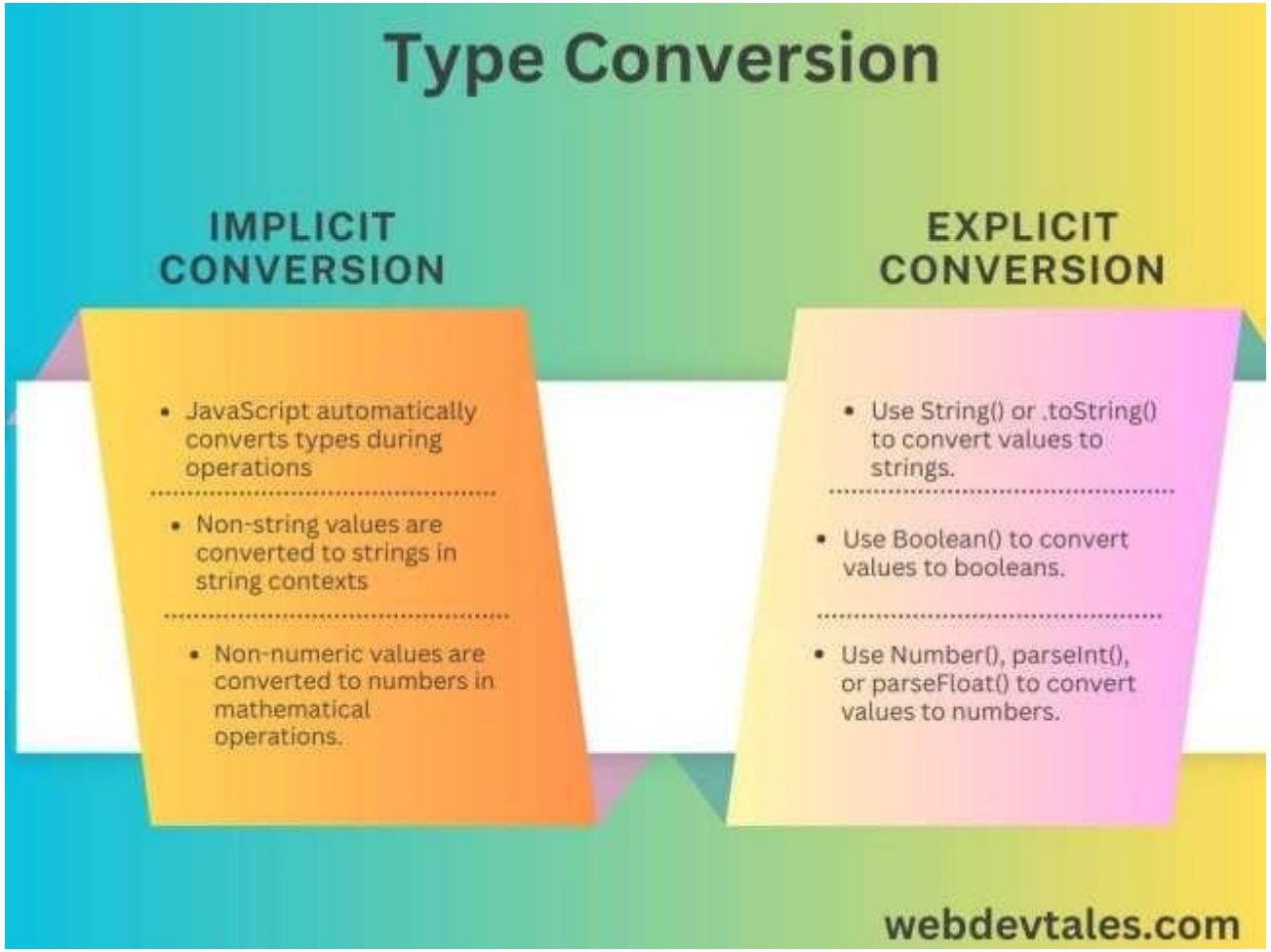
OPERATOR PRECEDENCE

Precedence	Operator	Operand type	Description
1	++,	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

DATA TYPES


JAVASCRIPT DATA TYPES
primitive
Boolean
Null
Undefined
Number
String
Symbol
object
Array
Object
Function
RegEx
Date

NUMBER TYPE CONVERSIONS



MATH FUNCTIONS

- Mathematical functions and values are part of a built in javascript object called Math.
- JavaScript provides following mathematical functions.

abs (value) :

Returns the absolute value of the number passed into it.

Ex: abs(-3); //prints 3

ceil(value):

Returns the smallest integer which is greater than, or equal to, the value passed in.

Ex: ceil (22.8) ,ceil(22.4); //prints 23,23

floor (value):

Returns the largest integer which is smaller than, or equal to, the number passed in.

Ex: floor (22.8),floor(22.8) ;//prints 22,22

min(value1, value2):

Returns the smaller of its argument.

Ex: min(3,4); // prints 3

max(value1, value2):

Returns the largest of its argument.

Ex: max(3,4); // prints 4

pow(value1, value 2):

Returns the result of raising value to power.

Ex: pow(2,4); // prints 16

round(value):

Returns the result of rounding its argument to the nearest integer.

Ex: round (5.3);// prints 5

sqrt(value):

Returns the square root of the value

Ex: sqrt(144);//Prints 12(sqrt of 144 is 12)

STRING FUNCTIONS

charAt(index):

This function returns the character which is at position index in the string.

```
Ex:  var str = "JavaScript";
     var res = str.charAt(4);
     document.writeln(res);          // prints " s "
```

concat("string" [, "string"[.."string"]])

This function is used to add two strings

```
Ex: var name = prompt("enter your name");
    var msg = "welcome";
    var res = msg.concat(name);
    document.writeln(res);          //prints welcome smith
```

indeOf("search"[,offset]):

- The string is searched for the string in the first parameter.
- If the search is successful, the index of the start of the target string is returned.
- If the search is unsuccessful the operation returns -1
- By default the indexOf ( ) functions starts index at 0 (zero).
- An optional offset may be specified so that the search starts part way along the string.

```
Ex:  var str = "JavaScript";
     var res = str.indexOf("S");
     document.writeln(res);      // prints " 4 "
```

lastIndexOf("search" [,offset]):

- This function does exactly the same thing as indexOf () but works its way backwards along the string.
- The offset works in exactly the same way as for indexOf ().

```
Ex:  var str = "JavaScript";
     var res = str.lastIndexOf("a");
     document.writeln(res);      // prints " 3 "
```

length:

Returns number of characters in the string.

```
Ex:  var str = "JavaScript";
     var res = str.length;
     document.writeln(res);      // prints " 10 "
```

STRING FUNCTIONS

split(separator [,limit]):

- This function breaks the string a part whenever it encounters the character passed in as the first parameter.
- The pieces of the string are stored in an array.
- Split() has an optional second parameter which is an integer value indicates how many of the pieces are to be stored in the array.

Ex:

```
var str="JavaScript is a scripting language";
var res= str.split(" ");
for(var i=0;i<res.length;i++)
        document.writeln(res[i] + "< /br>"); // prints " JavaScript
                                                        is
                                                        scripting
                                                        language"
```

subStr(index[,length]):

- Returns a substring which starts at the position specified by index.
- The substring continues either to end of the string or for the no of characters indicated by the length parameter.

Ex:

```
var str = "JavaScript";
var res = str.substr(4,3);
document.writeln(res);      // prints "scr"
```

STRING FUNCTIONS

subStr(index[,length]):

- Returns a substring which starts at the position specified by index.
- The substring continues either to end of the string or for the no of characters indicated by the length parameter.

```
Ex:     var str = "JavaScript";
        var res = str.substr(4,3);
        document.writeln(res);      // prints "scr"
```

substring(index1,[index2]):

Returns the set of characters which starts at index1 and continues up to, but does not include, the character at index2

```
Ex:     var str = "JavaScript";

        var res = str.subString(4,6);
        document.writeln(res);      // prints "sc "
```

toLowerCase( ):

Converts all characters in the string to lowercase.

```
Ex:     var str = "JavaScript";
        var res = str.toLowerCase();
        document.writeln(res);      // prints "javascript"
```

toUpperCase( ) :

Converts all characters in the string to uppercase.

```
Ex:     var str = "javascript";
        var res = str.toUpperCase();
        document.writeln(res);      // prints " JAVASCRIPT  "
```

ARRAYS AND ITS FUNCTIONS

- An array is an ordered set of date elements which can be accessed through a single variable name.
- In many programming language arrays are contiguous areas of memory which mean that the first array element is physically located next to the second, and so on.
- In JavaScript, an array is slightly different because it is a special type of object and has functionality which is not normally available in other languages.

Basic Array Functions:

- In javascript, arrays are special type of objects.
- The basic operations that can be performed on traditional arrays are
  1. Creation
  2. Addition of elements
  3. Accessing individual elements
  4. Removing element
  5. Searching element

1. Creating arrays :

Three ways of creating javascript arrays are

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday"];
var days =new Array ("Monday", "Tuesday", "Wednesday", "Thursday");
var days =new Array(4);
```

ARRAYS AND ITS FUNCTIONS

2. **Addition elements to an array:**
   - Array elements are added by their index. The index denotes the position of the element in the array and these start form 0(zero).
   - Adding an element uses the square bracket syntax
     Ex: var days[3]="Monday";

   What happens if we want to add an item to an array which is already full, the javascript interpreter simply extends the array and inserts the new line.
3. **Accessing Array:**

   The elements in the array are accessed through their index.

```
Ex: var days=["mon", "tue", "wed"]
    for(var  i=0;i<days.length;i++)
        document.writeln(days[i]);
```

4. **Searching an array:**

   To search an array, simply read each element and compare it with search element.

```
Ex: for(var  i=0;i<length;i++)
    {
      if (data[i]== "Tuesday") {
        document.writeln(data[i]+", ");
        break;       }
    }
```

ARRAYS AND ITS FUNCTIONS

5. **Removing array members:**

JavaScript does not provide a built in function for removing an element from an array. So the following procedure is used to remove an element.

- Read each element in the array
- If the element is not the one you want to delete, copy it into a temporary array
- If you want to delete the element then do nothing
- Increment the loop counter
- Repeat the process.

Ex:

```
<script>
        var days = ["Monday", "Tuesday","Wednesday","Thursday"];
        for(var i=0; i< days.length;i++)
            document.writeln(days[i]);
        var rem = prompt ("enter element to remove");
        var temp=new Array(days.length -1);
        var count=0;
        for(var i=0;i<leng;i++)
        {
                if(days[i]==rem)
                {
                }
                else
                {
                        temp[i]=days[i];
                        count++;
                }
        }
</script>
```

ARRAYS AND ITS FUNCTIONS

Object-based Array function:

Since array is an object in Java script the following Array functions helps to manipulate Array elements.

1. concat(array2[, array3 [,array n]]):

A list of arrays is concatenated onto the end of the array and a new array returned.The original arrays are all unaltered by this process.

```
Ex: var first=[1,2,3];
    var second=[4,5];
    var third=[6,7,8];
    var res=first.concat(second, third);
    document.writeln(res);      // prints 1,2,3,4,5,6,7,8
```

2. join(string):

- This functio  s all the elements of array as a string.
- In the resulting string, the elements are separated using the optional string parameter.
- If this is omitted the elements will be separated using a comma(,)

```
Ex:   var a =[1,2,3,4];
      var res= a.join(" ; ");
      document.writeln(res);                    //prints  1;2;3;4
```

3. pop( ):

Removes the last element from the array.

```
Ex:   var a=[4,5,6,7];
      a.pop( );
      document.writeln(res);                    //prints    4,5,6
```

4. push (element1 [, element 2 [, element n]]):

Adds a list of items into the end of the array.

```
Ex:   var a=[1,2,3];
      a.push(4,5,6);
      var res1=a.join(" ; ");
      document.writeln(res);                    //prints  1;2;3;4;5;6
```

5. reverse( ):

Swaps all the elements in the array so that which was first is last, and viceversa.

```
Ex:   var a=[1,2,3];
      a.reverse();
      var res1=a.join(" ; ");
      document.writeln(res1);                   //prints 3;2;1
```

ARRAYS AND ITS FUNCTIONS

6. **shift( ):**
   removes the first element of the array

```
Ex:   var a=[1,2,3];
      a.shift();
      var res1=a.join( " ; ");
      document.writeln(res1);          //prints 2;3
```

7. **slice(start [,finished]):**
   To extract a range of elements from an array

```
Ex:   var a=[1,2,3,4,5,6,7];
      a.slice(1,3);
      var res1=a.join( " ; ");
      document.writeln(res1);          //prints 2;3
```

8. **sort():**
   Helps to sort array elements into lexicographic, dictionaryorder.

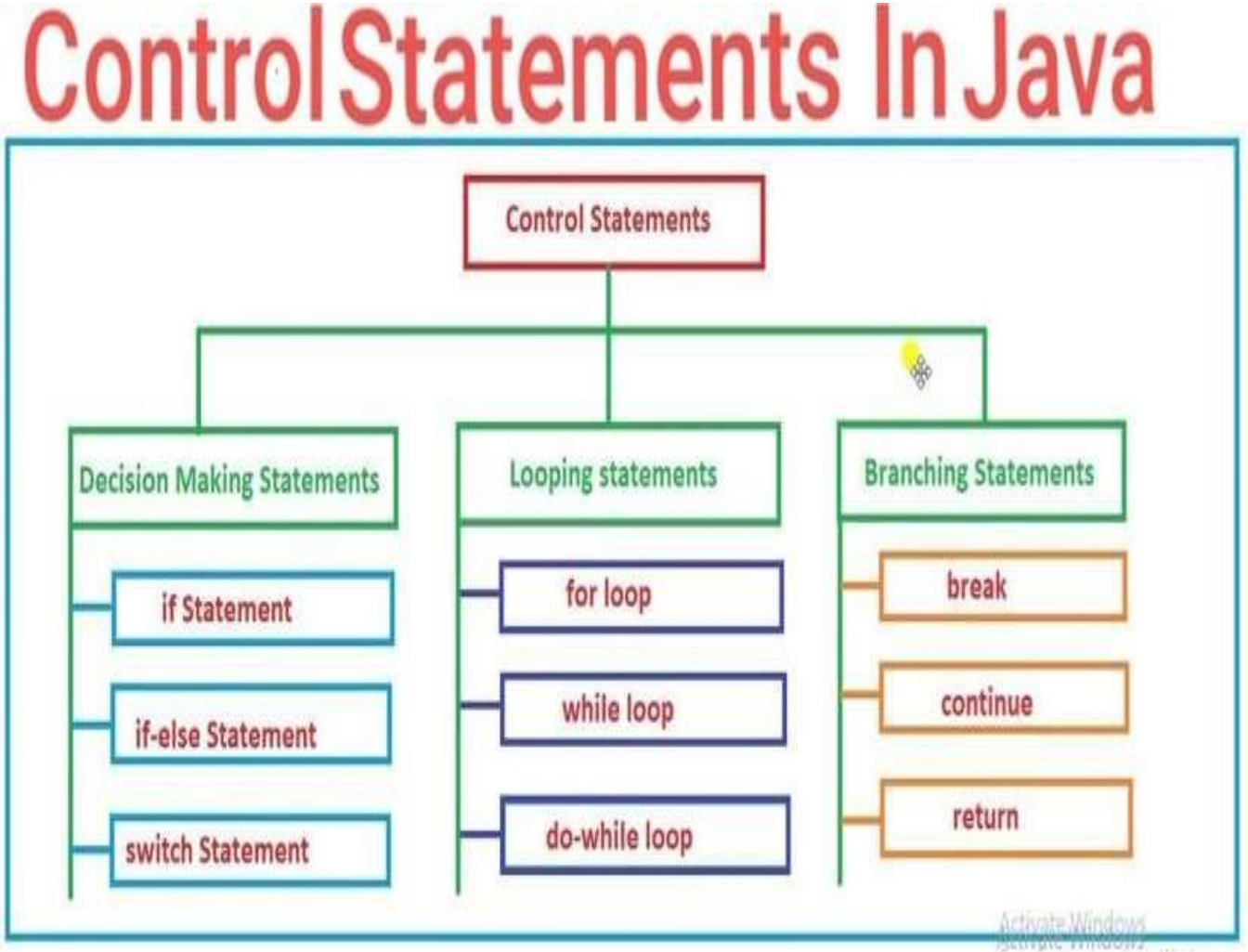
```
Ex:   var a=[9,8,6];
      a.sort( );
      var res1=a.join( " ; ");
      Rdocument.writeln(res1);         //prints  6;8;9
```

ARRAYS AND ITS FUNCTIONS

9. **splice(number [,element1][, element2[element 3]]]):**
   - splice is used to alter an array by removing some elements and at the same elements and at the sametime adding in new ones.
   - The first parameter indicates the position in the array at which the new element will start.
   - The second parameter indicates how many elements will be deleted from the original array.
   - If you don't want to delete element then set this to o
   - Finally there is a list of elements which are to be inserted into the array.

```
Ex:     var a=[1,2,3,6,7];
        var res=a.splice(3,0,4,5);
        var res1=a.join(" ; ");
        document.writeln(res1);                 //prints 1; 2;3;4;5;6;7
```

CONTROL STATEMENTS



Control Statements

- Statements that control the flow of program is called Control Statements
- In Java script, control statements can be divided into the following three categories:

  Selection Statements

  Iteration Statements

  Jump Statements

Selection Statements

Selection Statements

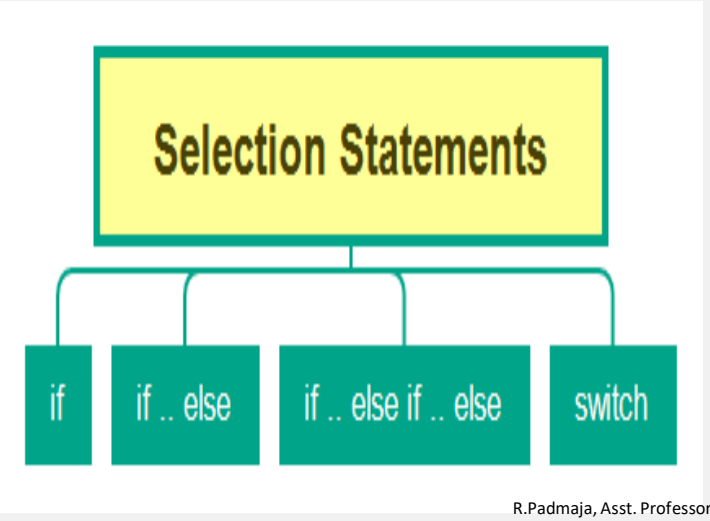
- Selection statements allow you to control the flow of program execution on the basis of the outcome of an expression or state of a variable known during runtime
- Selection statements can be divided into the following categories:

  The Simple if

  if-else statements

  The if-else-if statements

  The switch statements



Simple If

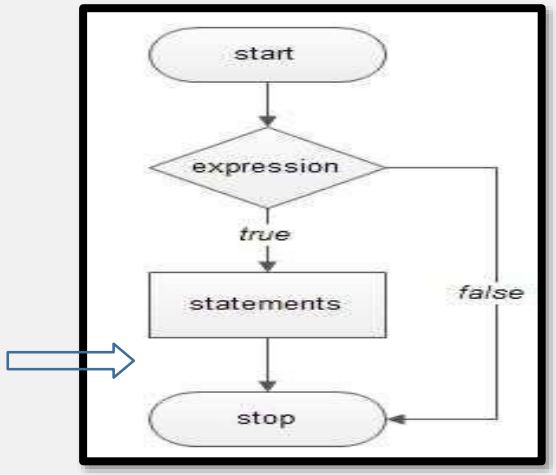
Simple If

- Simple If construct of Java consist of one if condition with one block of statements.
- When condition becomes true then executes the block given below it.

Syntax:

```
if ( condition):
        ……..
```

Flowchart



If.. Else

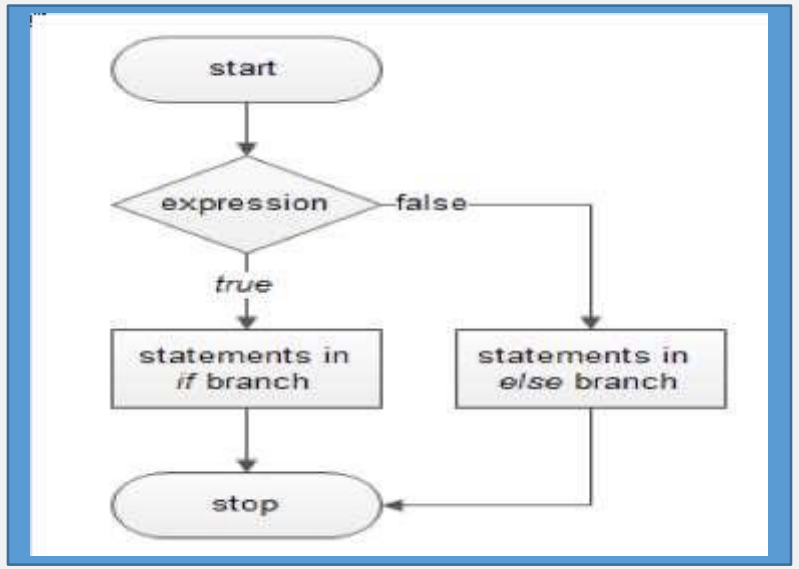
if - else statements

- This construct of Java program consist of one if condition with two blocks.
- When condition becomes true then executes the block given below it.
- If condition evaluates result as false, it will executes the block given below else.

Syntax:

```
if ( condition):
        ……..
e lse:
        ……..
```

Flowchart

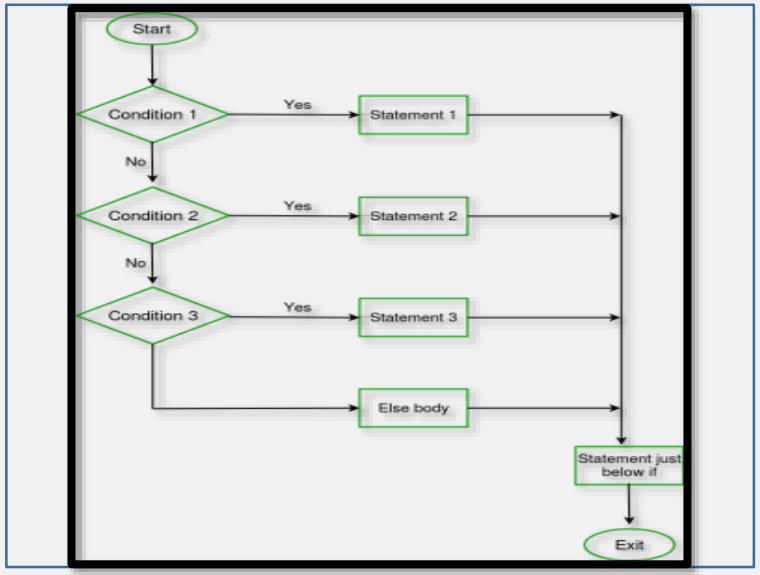


If.. Else if .. else

Java Ladder if else statements (if-elif-else)

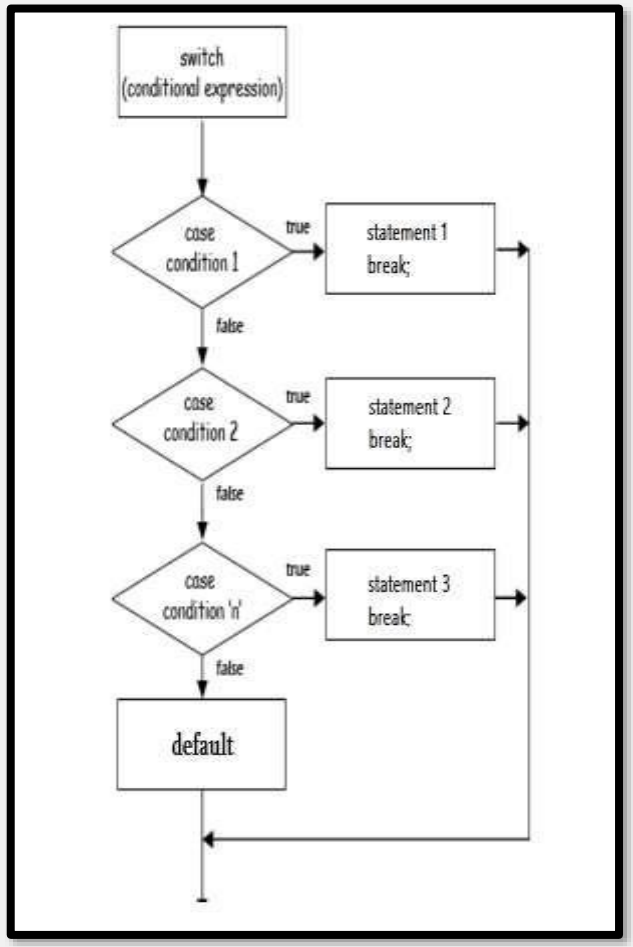
- This construct of Java program consist of more than one if condition.
- When first condition evaluates result as true then executes the block given below it.
- If condition evaluates result as false, it transfer the control at else part to test another condition.
- So, it is multi-decision making construct.

```
if ( condition-1):
        ……..
        ……..
 elif (condition-2):
        ……..
        ……..
elif (condition-3):
        ……..
        ……..
 else:
        ……..
        ……..
```



Switch Statement

- Java n Switch Case is a selection control statement.
- **The switch expression is evaluated once. *The value of the expression is compared with the values of each case; if there is a match, the associated block of code is executed.***
- **Then it makes a decision based on whether the condition is true or not.**
- **If the condition is true, it evaluates the indented expression; however, if the condition is false, the indented expression under else will be evaluated.**
- **When we need to run several conditions, you can place as many elif conditions as necessary between the *if* condition and the *else* condition.**
- **Switch case statements are a substitute for long if statements that compare a variable to several integral values**
- **The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.**



Iterative/Looping Statements

- The flow of the programs written in any programming language is **sequential by default**.
- The first statement of the program is executed first, followed by the second, and so on.
- There may be a situation when the programmer needs **to execute a block of code several times**.
- For this purpose, The programming languages provide **various kinds of loops** that are able to repeat some particular code numerous numbers of times.
- **Looping simplifies complicated problems into smooth ones.**
- **It allows programmers to modify the flow of the program so that rather than writing the same code, again and again, programmers are able to repeat the code a finite number of times.**
- In Python, there are three different types of loops:

  For Loop

  While Loop and

  Nested Loop

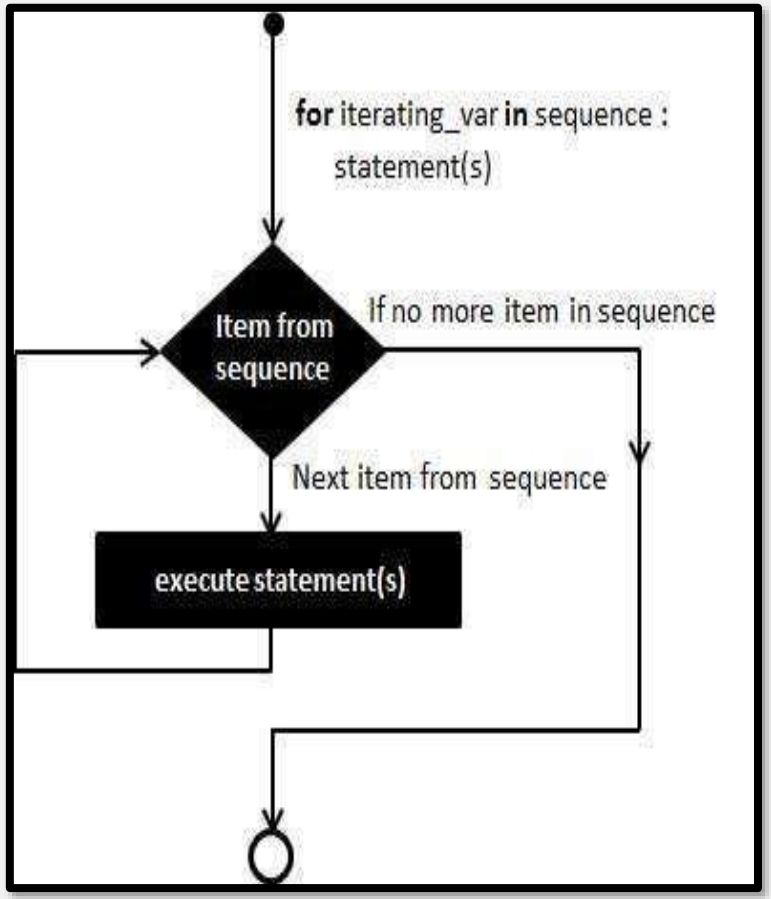
For Statement

For Loop

- The for loop is used in the case where a programmer needs to execute a part of the code until the given condition is satisfied.
- The for loop is also called a pre-tested loop.
- It is best to use for loop if the number of iterations is known in advance.

```
for (statement 1; statement 2; statement 3)
{ // code block to be executed }
```

- **Statement 1** is executed (one time) before the execution of the code block.
- **Statement 2** defines the condition for executing the code block.
- **Statement 3** is executed (every time) after the code block has been executed.

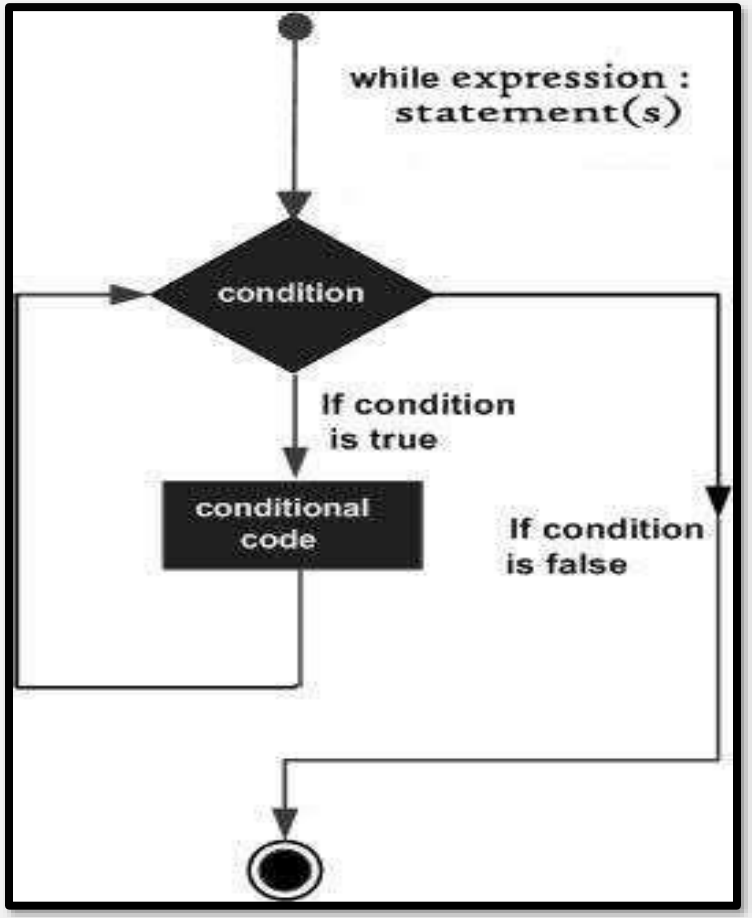


While Statement

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- The syntax of a **while** loop in Python programming language is

```
while expression:
   statement(s)
```

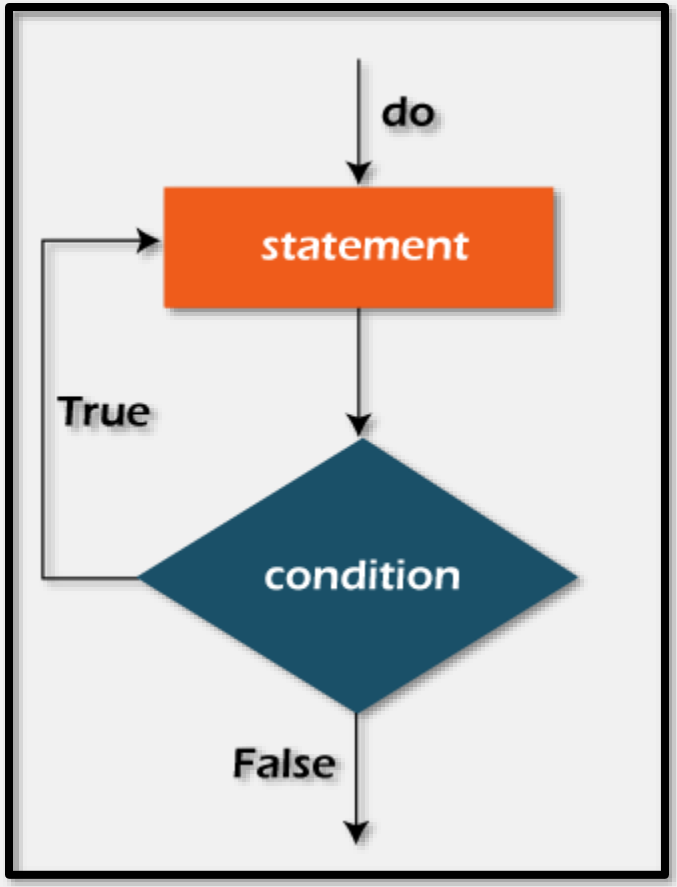
- Here, **statement(s)** may be a single statement or a block of statements. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code.
- Python uses indentation as its method of grouping statements.



Do..While Statement

- The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```



FUNCTIONS

- A function is a piece of code that performs a specific task.
- Functions are defined using the function keyword.

```
function Name(parameter)
{

}
```

- The function name can be any combination of digits, letters and underscores but cannot contain a space.
- That's the same rule as for variable meaning.
- To find factorial of a number using function is:

```
<html>
    <head>
        <script>
            function factorial ( k )
            {
                var fact=1;
                for(var i=1;i<=k;i++)
                    fact=fact*i;
                    return fact;
             }
        </script>
    </head>
    <body>
        <script>
            var n=prompt("enter n value:");
            var res=factorial ( n );
            document.writeln(res);
        </script>
    </body>
</html>
```

CASE STUDY 1: Student Marks Calculator

Scenario:

A webpage allows a user to enter marks of 5 subjects and calculates:

- Total
- Average
- Grade

Questions:

1. Declare variables for marks
2. Use **operators** to calculate total & average
3. Use **conditional statements (if-else)** to assign grade
4. Display result using **document object**

CASE STUDY 2: Simple Calculator

Scenario:

Create a calculator that performs:

- Addition
- Subtraction
- Multiplication
- Division

Questions:

1. Take input values
2. Use **functions** for each operation

3. Use **switch case** for operation selection
4. Display output

CASE STUDY 3: Login Validation System

Scenario:

User enters username and password:

- If correct → "Login Successful"
- Else → "Invalid Login"

Questions:

1. Use **variables and data types**
2. Apply **conditional statements**
3. Show result using **alert() or document.write()**

CASE STUDY 4: Number Analyzer

Scenario:

User enters a number, and the system checks:

- Positive / Negative
- Even / Odd

Questions:

1. Use **operators (% for modulus)**
2. Apply **if-else conditions**
3. Display results

CASE STUDY 5: Student Data using Arrays & Objects

Scenario:

Store details of students:

- Name
- Age
- Marks

Questions:

1. Create **array of objects**
2. Access and display data
3. Use **looping (for loop)** to print all students

CASE STUDY 6: Digital Clock

Scenario:

Display current date and time on webpage

Questions:

1. Use **Date object**
2. Extract hours, minutes, seconds
3. Display time dynamically

CASE STUDY 7: String Manipulation Tool

Scenario:

User enters a string, and program:

- Converts to uppercase
- Finds length
- Reverses string

Questions:

1. Use **string functions**
2. Apply **loops** for reversing
3. Display results

CASE STUDY 8: Multiplication Table Generator

Scenario:

User enters a number → display multiplication table

Questions:

1. Use **loop (for/while)**
2. Use **document object** to print table

CASE STUDY 9: Form Input Validation

Scenario:

Validate user input:

- Name should not be empty
- Age must be number

Questions:

1. Use **data type checking**
2. Apply **conditions**
3. Show error messages

CASE STUDY 10: Shopping Cart Total

Scenario:

User selects items → calculate total price

Questions:

1. Store prices in **array**
2. Use **loop to calculate total**
3. Apply **functions**

UNIT- II: OVERVIEW OF JAVASCRIPT
JAVASCRIPT: Introduction to JavaScript, Applying JavaScript (internal and external), Understanding JS Syntax, Introduction to Document and Window Object, Variables and Operators, Data Types and Num Type Conversion, Math and String Manipulation, Objects and Arrays, Date and Time, Conditional Statements, Switch Case, Looping in JS and Functions.

PART -A		
1	What is JavaScript?	L1
2	Write the syntax for declaring a variable in JS.	L3
3	What are the different data types in JavaScript?	L1
4	Write the use of the Math object in JavaScript.	L2
5	What is the difference between == and === in JS?	L2
6	What is an array in JavaScript?	L1
7	Define a function in JavaScript.	L1
8	What is the purpose of the Date object in JavaScript?	L2
9	Write a for loop syntax in JavaScript.	L3
10	List any two string manipulation methods in JavaScript.	L1
PART –B		
1	Explain how to apply JavaScript internally and externally with examples.	L2
2	Describe the various conditional statements in JavaScript with syntax and examples.	L2
3	Write a program to demonstrate use of arrays and objects in JavaScript.	L3
4	Explain Math and String manipulation methods in JavaScript with examples.	L2
5	Write a program to check if a number is even or odd using JavaScript.	L3
6	Explain loops in JavaScript with suitable examples.	L2
7	Describe functions in JavaScript with types and examples.	L2
8	Explain Document and Window objects in JavaScript.	L2
9	Write a JavaScript program to calculate the factorial of a number.	L3
10	Discuss type conversion and its types in JavaScript.	L2

FULL STACK WEB DEVELOPMENT

UNIT –III

REACT JS

Simple things should be simple, complex things should be possible.

— Alan Kay

Unit -III: REACT JS

1. Unit Overview

This unit introduces **React JS**, a popular JavaScript library for building dynamic and interactive user interfaces. Topics include JSX templating, components, state and props management, component lifecycle, lists and portals, routing, error handling, state management with Redux and Redux Saga, Immutable.js, server-side rendering, unit testing, and module bundling with Webpack.

2. Objectives of the Unit

By the end of this unit, students should be able to:

- Understand the **core concepts of React JS** and its component-based architecture.
- Use **JSX** to create dynamic templates.
- Manage component **state and props** effectively.
- Understand the **component lifecycle** and implement hooks or lifecycle methods.
- Render lists, portals, and handle **errors** in React applications.
- Implement **routing** using React Router.
- Use **Redux and Redux Saga** for state management.
- Work with **Immutable.js** for immutable data structures.
- Implement **server-side rendering (SSR)** and perform **unit testing**.
- Configure and use **Webpack** for bundling React applications.

3. Learning Outcomes

After completing this unit, students will be able to:

- Build **dynamic and interactive web applications** using React JS.
- Create **reusable components** with JSX templates.
- Manage and update **state and props** in components.
- Handle events, errors, and conditional rendering efficiently.
- Implement **client-side routing** and navigation using React Router.
- Apply **Redux & Redux Saga** for scalable state management.
- Use **Immutable.js** to improve data reliability and performance.
- Perform **server-side rendering** and write **unit tests** for React components.
- Bundle and optimize applications using **Webpack**.

4. Importance of Studying this Unit

- React JS is one of the **most widely used front-end libraries** in modern web development.
- Mastering React enables students to build **scalable, maintainable, and high-performance web applications**.
- Knowledge of React and related tools (Redux, Webpack, SSR) is essential for **professional web development roles**.
- Provides a strong foundation for **full-stack development** when paired with Node.js and MongoDB.

5. Key Concepts

- **Introduction to React:** Overview, benefits, and use cases.
- **JSX Templating:** Syntax, expressions, and embedding in components.
- **Components:** Functional and class-based components, component structure.
- **State and Props:** Managing data within and across components.
- **Lifecycle of Components:** Mounting, updating, and unmounting phases.
- **Rendering Lists & Portals:** Rendering arrays of data, using React portals.
- **Error Handling:** Error boundaries and exception management.
- **Routing:** React Router for navigation between views.
- **Redux & Redux Saga:** Global state management and asynchronous operations.
- **Immutable.js:** Creating immutable data structures.
- **Server-Side Rendering (SSR):** Rendering React on the server for performance and SEO.
- **Unit Testing:** Testing React components with Jest or React Testing Library.
- **Webpack:** Module bundler for building and optimizing React apps.

UNIT- III

REACT JS

INTRODUCTION

React JS is a JavaScript library (not a full framework) that's mainly used for building user interfaces — especially for **single-page applications** where the page doesn't reload when you interact with it.

In simple words:

React helps you build dynamic websites that can update parts of the page without refreshing the whole page.

Some key points about React JS:

- **Developed by Facebook** (now Meta).
- **Component-based** — You build small pieces (called components) like buttons, forms, navbars, etc., and combine them to make a full app.
- **Very fast** — Uses something called a **Virtual DOM** to make updates super quick.
- **Reusable code** — Once you create a component, you can reuse it wherever you want.
- **Works with HTML & JavaScript together** (something called **JSX**).

Example: Without React → Every time you click a button, the whole page might reload.

With React → Only that button or part of the page updates instantly without reloading.

When people say **"React is a library, not a full framework,"** they mean:

- **React** only **handles the "view" layer** (UI part) — basically updating and rendering your components efficiently.

View	User Interface (UI)	What the user sees (buttons, forms, pages)

UNIT- III

REACT JS

What is a Single-Page Application (SPA)?

A **Single-Page Application (SPA)** is a web app **that loads only one HTML page** at the start, and **dynamically updates** the page **without reloading the entire page** from the server.

Example:

Traditional Website (Multi-Page)	Single-Page Application (SPA)
Every time you click a link, the whole page reloads from server	Page never reloads. Only parts of page change
Slower because full HTML loads again and again	Faster because only small parts update
Example: Old websites (PHP, WordPress)	Example: Gmail, Facebook, Instagram (React, Vue, Angular apps)

What is Virtual DOM?

✓**Virtual DOM** is a **copy of the real DOM** (the web page structure) but **inside memory**.
✓React uses this **Virtual DOM** to make the web page updates **faster and smoother**.

In Technical Words:

- When something changes (like clicking a button),
- React **does NOT update the real webpage directly**.
- It **updates the Virtual DOM first** (which is very fast □).
- Then it **compares** (diffs) the old Virtual DOM with the new one.
- React **finds only the parts that changed**.
- React then **updates ONLY those parts** in the Real DOM!

UNIT- III

REACT JS

✅**No full page reload**

✅**Only tiny updates**

✅**Super fast performance**

In One Line:

Virtual DOM = A smart memory copy of the real page to make changes faster and efficient.

Templating using JSX

What is JSX?

JSX stands for **JavaScript XML**.

It **looks like HTML**, but it's actually **JavaScript** under the hood. You use JSX to **design how your UI should look** in React.

✅JSX lets you **write HTML inside JavaScript** easily.

Why use JSX?

- Easier to **visualize** your UI while coding.
- Cleaner and **less confusing** than building UI using pure JavaScript.
- It **compiles** into simple React.createElement() behind the scenes.

Basic JSX Example

const element = <h1>Hello, world!</h1>;

- <h1>Hello, world!</h1> looks like HTML, but it's **JSX**.
- element is a normal **JavaScript variable** holding that JSX.

UNIT- III

REACT JS

JSX Full Details

1. Multiple Elements Must Be Wrapped

You **cannot return two sibling elements directly** in JSX.

✗This will cause an error:

```
return <h1>Hello</h1><p>World</p>;
```

✓You must wrap them inside a **single parent**, like a or a **React Fragment**:

```
return (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
);
```

Or using **Fragment**:

```
import React from 'react';
return (
  <>
    <h1>Hello</h1>
    <p>World</p>
  </>
);
```

2. Embedding JavaScript inside JSX

Use **curly braces {}** to run JavaScript inside JSX.

Example:

```
const name = 'John';
return <h1>Hello, {name}!</h1>;
```

It will show: Hello, John!

You can even do calculations inside:

```
return <p>2 + 2 = {2 + 2}</p>;
```

Output: 2 + 2 = 4

3. JSX Attributes

Attributes in JSX **look like HTML**, but some are **camelCase**.

HTML Attribute	JSX Attribute
class	className
for	htmlFor

Example:

```
return <h1 className="title">Hello World</h1>;
```

4. Inline Styling in JSX

To apply CSS **directly** using JSX, you pass a **JavaScript object** inside {}.

Example:

```
const style = {
  color: 'blue',
  backgroundColor: 'yellow',
```

```
  fontSize: '20px'
};

return <h1 style={style}>Styled Text</h1>;
```

or directly inside:

```
return <h1 style={{ color: 'red', fontWeight: 'bold' }}>Hello</h1>;
```

5. Conditional Rendering (if/else in JSX)

JSX doesn't allow if/else directly.
Instead, you use **ternary operators** or **short-circuiting**.

Example using ternary:

```
const isLoggedIn = true;
return (
  <div>
    {isLoggedIn ? <p>Welcome back!</p> : <p>Please login.</p>}
  </div>
);
```

Example using &&:

```
const isAdmin = true;
return (
  <div>
    {isAdmin && <p>Admin Panel</p>}
  </div>
);
```

UNIT- III

REACT JS

6. Lists in JSX (Looping)

You can use .map() to render lists.

Example:

```
const users = ['John', 'Jane', 'Alice'];
return (
  <ul>
    {users.map((user, index) => (
      <li key={index}>{user}</li>
    ))}
  </ul>
);
```

- Always add a **unique key prop** when rendering lists!

JSX Final Notes:

Feature	Description
Curly Braces {}	To embed JS
className	Instead of class
style Object	Inline CSS styling
Ternary & &&	Conditional rendering
map()	To render lists
Must wrap in parent	div or Fragment

In Short:

JSX = **Write HTML in JavaScript** + **Add Power of JavaScript in HTML**

UNIT- III

REACT JS

Example Program

Profile Card

```
import React from 'react';

function ProfileCard() {

const user = {

    name: 'John Doe',

    age: 24,

    bio: 'A passionate developer ',

    profilePicture: 'https://randomuser.me/api/portraits/men/75.jpg'

  };

  const cardStyle = {

    border: '1px solid #ccc',

    borderRadius: '10px',

    width: '300px',

    padding: '20px',

    textAlign: 'center',

    boxShadow: '0px 0px 10px rgba(0,0,0,0.1)'

  };

  const imageStyle = {

    width: '100px',

    height: '100px',

    borderRadius: '50%',
```

```
    objectFit: 'cover',
    marginBottom: '10px'
  };
  return (
    <div style={cardStyle}>
      <img src={user.profilePicture} alt="Profile" style={imageStyle} />
      <h2>{user.name}</h2>
      <p>Age: {user.age}</p>
      <p>{user.bio}</p>
    </div>
  );
}
export default ProfileCard;
```

What this JSX is doing:

Part	What's Happening
{user.name}	Embeds JavaScript value into the HTML
{} inside style	Inline CSS Styling
	Dynamic Image
Whole layout inside	Wrapped in a single parent

How to Render it:

In your index.js:

```
import React from 'react';
```

UNIT- III

REACT JS

```
import ReactDOM from 'react-dom';

import ProfileCard from './ProfileCard';

ReactDOM.render(

  <ProfileCard />,

  document.getElementById('root')

);
```

How it will look visually:

[Profile Picture]

John Doe

Age: 24

A passionate developer

ReactJS Components

In **ReactJS**, components are the **building blocks** that help you create **interactive UIs** by dividing your app into smaller, **reusable pieces of code**. Understanding how **components** work is essential for efficiently developing **React applications**.

What are ReactJS Components?

Components in React are JavaScript functions or classes that return a piece of UI. These components allow developers to build complex UIs from small, isolated, and reusable pieces. React components are the core building blocks for any React application and can manage their state, handle user inputs, and render dynamic content.

Types of React Components

There are two main types of components in React:

- **Functional Components**
- **Class Components**

UNIT- III

REACT JS

Functional Components

A Functional Component is a simpler and more concise way of writing components in React using JavaScript functions. These components receive props (properties) as an argument and return JSX (JavaScript XML) to define the UI structure.

```
import React, { useState } from 'react';
const Welcome = () => {
   const [message, setMessage] = useState("Hello, World!");
   return (
      <div>
         <h1>{message}</h1>
         <button onClick={() => setMessage("Hello, React!")}>
            Change Message
         </button>
      </div>
   );
};
export default Welcome;
```

Output

Functional Components

In this code

- useState is used to manage the message state, initially set to "Hello, World!".

- The button click triggers setMessage, which updates the message state to "Hello, React!".
- The component displays the message in an <h1> element and updates it when the button is clicked.

Class Components

Class components in React are ES6 classes that extend the React.Component class. They are used for creating components that need to have their own state or lifecycle methods. While functional components are now the go-to choice for many developers (especially with the introduction of hooks like useState and useEffect), class components still have their place and provide a more traditional way of handling component logic in React.

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };
  decrement = () => {
    this.setState({ count: this.state.count - 1 });
  };
  render() {
    return (
      <div>
        <h1>Counter: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
        <button onClick={this.decrement}>Decrement</button>
      </div>
```

```
      );
    }
}

export default Counter;
```

Output

Class Components in React

In this code

- **state:** The counter (count) is initialized to 0 in the constructor.
- **increment and decrement:** These methods update the count state by 1 when the respective buttons are clicked.
- **render:** Displays the current count and two buttons to increment or decrement the counter.

Difference between Class component and Functional component

Feature	Functional Components	Class Components
Definition	A function that returns JSX.	A class that extends React.Component and has a render() method.
State	Can use state via hooks like useState.	Can manage state using this.state and this.setState().

UNIT- III

REACT JS

Feature	Functional Components	Class Components
Lifecycle Methods	No lifecycle methods (use useEffect hook).	Uses lifecycle methods like componentDidMount, componentDidUpdate, componentWillUnmount.
Syntax	Simpler syntax, function-based.	More verbose, class-based syntax.
Performance	More lightweight and performant.	Slightly heavier due to the class structure and additional methods.
Use of Hooks	Can use hooks like useState, useEffect, etc.	Cannot use hooks directly.
Rendering	Directly returns JSX.	Must define a render() method to return JSX.
Context	Easier to integrate with React hooks like useContext.	Uses Context.Consumer for context integration.

Functional Component (with Hooks)

```
import React, { useState, useEffect, useRef, useContext, createContext } from 'react';

// Creating Context for global state

const CountContext = createContext();

const Counter = () => {
```

UNIT- III

REACT JS

```
// useState Hook: to manage the count state

const [count, setCount] = useState(0);

// useRef Hook: to focus on the input element

const inputRef = useRef(null);

// useEffect Hook: to run code when the component mounts

useEffect(() => {

  console.log('Component Mounted or Updated');

  // Focusing on the input element after render

  inputRef.current.focus();

  return () => {

    console.log('Component Unmounted');

  };

}, [count]); // Dependency array ensures it runs when count changes

// useContext Hook: Accessing the value from the context

const globalCount = useContext(CountContext);

return (

  <div>

    <h1>Functional Component - Counter</h1>
```

```
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
      {/* Using useRef to focus the input */}
      <input ref={inputRef} type="text" placeholder="Focus me" />
      {/* Displaying context value */}
      <p>Global Count from Context: {globalCount}</p>
    </div>
  );
};
// App Component with Context.Provider
const App = () => {
  const [globalCount, setGlobalCount] = useState(10);
  return (
    <CountContext.Provider value={globalCount}>
      <Counter />
      <button onClick={() => setGlobalCount(globalCount + 1)}>Increase Global Count</button>
    </CountContext.Provider>
```

```
  );

};

export default App;
```

Explanation of Functional Component with Hooks:

1. **useState**: Manages the local state count. It returns the current state and a function to update it.
2. **useEffect**: Runs side effects like logging messages or focusing the input field. Here, it runs on every render because count is part of the dependency array.
3. **useRef**: Creates a reference to the input field, and after each render, it focuses on the input element.
4. **useContext**: Allows us to consume the value from a CountContext provider. The globalCount value is managed at the App level but accessed inside Counter via useContext.

Class Component (with Lifecycle Methods)

```
import React, { Component } from 'react';

// Creating Context for global state

const CountContext = React.createContext();

class Counter extends Component {

constructor(props) {

    super(props);

    // Initializing state
```

```
  this.state = {
    count: 0,
  };
  this.inputRef = React.createRef(); // For input focus
}
// Lifecycle method: componentDidMount (similar to useEffect with empty dependency array)
componentDidMount() {
  console.log('Component Mounted');
  // Focus on the input element after mounting
  this.inputRef.current.focus();
}
// Lifecycle method: componentDidUpdate (similar to useEffect with count as dependency)
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    console.log('Count has been updated');
  }
}
// Lifecycle method: componentWillUnmount (similar to cleanup in useEffect)
```

```
componentWillUnmount() {
  console.log('Component will Unmount');
}
render() {
  return (
    <div>
      <h1>Class Component - Counter</h1>
      <p>Current Count: {this.state.count}</p>
      <button onClick={() => this.setState({ count: this.state.count + 1 })}>Increment</button>
      <button onClick={() => this.setState({ count: this.state.count - 1 })}>Decrement</button>
      {/* Using createRef to focus the input */}
      <input ref={this.inputRef} type="text" placeholder="Focus me" />
      {/* Consuming Context value */}
      <CountContext.Consumer>
        {(globalCount) => <p>Global Count from Context: {globalCount}</p>}
      </CountContext.Consumer>
    </div>
  );
```

```
  }

}

// App Component with Context.Provider

class App extends Component {

constructor(props) {

    super(props);

    this.state = {

    globalCount: 10,

    };

  }

  render() {

    return (

      <CountContext.Provider value={this.state.globalCount}>

        <Counter />

        <button onClick={() => this.setState({ globalCount: this.state.globalCount + 1 })}>Increase
Global Count</button>

      </CountContext.Provider>

    );

  }
```

```
}

export default App;
```

Explanation of Class Component with Lifecycle Methods:

1. **State**: Managed through this.state, and updated using this.setState().
2. **Lifecycle Methods**:
    - **componentDidMount()**: Equivalent to useEffect() with an empty dependency array. This is called after the component mounts.
    - **componentDidUpdate()**: Similar to useEffect() that runs when a specific value (count) changes.
    - **componentWillUnmount()**: Equivalent to the cleanup function in useEffect(), called before the component is removed.
3. **createRef**: A reference is created to focus the input element, similar to useRef.
4. **Context**: Consumed via CountContext.Consumer to access the global globalCount.

Comparison of Functional and Class Component:

Feature	Functional Component (with Hooks)	Class Component (with Lifecycle Methods)
State Management	useState()	this.state, this.setState()
Side Effects	useEffect()	componentDidMount(), componentDidUpdate(), componentWillUnmount()
Refs	useRef()	React.createRef()
Context Consumption	useContext()	CountContext.Consumer
Focus Management	useRef() (for focusing input)	React.createRef() (for focusing input)

UNIT- III

REACT JS

Feature	Functional Component (with Hooks)	Class Component (with Lifecycle Methods)
Clean Up	useEffect() clean-up function	componentWillUnmount()

Feature Comparison: Functional Components (with Hooks) vs Class Components (with Lifecycle Methods)

Explanation of Features:

1. State Management:

- **Functional Components**: Use the useState() hook to manage local component state. useState() returns an array where the first value is the state variable, and the second value is a function to update the state.

  ```
  const [count, setCount] = useState(0);
  ```

- **Class Components**: Use this.state to define state and this.setState() to update the state. State is usually defined in the constructor, and this.setState() triggers a re-render.

  ```
  constructor(props) {
     super(props);
     this.state = { count: 0 };
  }
  this.setState({ count: 1 });
  ```

2. Side Effects:

- **Functional Components**: useEffect() hook is used to handle side effects like data fetching, subscriptions, or manually changing the DOM. It can also act like multiple lifecycle methods (componentDidMount, componentDidUpdate, and componentWillUnmount) depending on how dependencies are set.

```
useEffect(() => {
  // Side effect code (e.g., fetching data)
}, []); // Empty dependency array means it runs once (componentDidMount)
```

- **Class Components**: The equivalent lifecycle methods are:
  - componentDidMount() — runs once after the component is mounted (similar to useEffect with an empty dependency array).
  - componentDidUpdate() — runs whenever the component updates (similar to useEffect with dependencies).
  - componentWillUnmount() — called before the component is removed from the DOM for cleanup (similar to useEffect cleanup).

3. Refs:

- **Functional Components**: useRef() is used to persist references to DOM elements or mutable values across renders without causing re-renders. It's also used for focus management.

```
const inputRef = useRef(null);
inputRef.current.focus(); // Focus input
```

- **Class Components**: React.createRef() is used to create a reference and attach it to a DOM element.

```
this.inputRef = React.createRef();
this.inputRef.current.focus(); // Focus input
```

4. Context Consumption:

- **Functional Components**: useContext() allows you to consume context values directly without needing to wrap the component in a Context.Consumer.

```
const value = useContext(MyContext);
```

- **Class Components**: You use CountContext.Consumer to consume values from the context. This requires a render prop pattern.

```
<CountContext.Consumer>
  {value => <div>{value}</div>}
</CountContext.Consumer>
```

5. Focus Management:

- **Functional Components**: You can use useRef() to store a reference to an input element and then focus it when necessary.

```
const inputRef = useRef();
inputRef.current.focus();
```

- **Class Components**: Use React.createRef() to manage focus in class components.

```
this.inputRef = React.createRef();
this.inputRef.current.focus();
```

6. Clean Up:

- **Functional Components**: You can return a clean-up function inside useEffect() to handle component cleanup, such as unsubscribing from a service or clearing timers.

```
useEffect(() => {
   const timer = setTimeout(() => console.log('Timer!'), 1000);
   return () => clearTimeout(timer); // Clean up
}, []);
```

- **Class Components**: componentWillUnmount() is used for cleanup, like clearing timers or canceling network requests.

```
componentWillUnmount() {
```

```
    clearTimeout(this.timer);
  }
```

Summary:

- **Functional Components** with **Hooks** provide a more concise, cleaner, and modular way to handle state, side effects, and other features.
- **Class Components** use lifecycle methods and internal state to manage component logic, but they are more verbose and less flexible compared to functional components with hooks.

State and Props in React

In React, **state** and **props** are two essential concepts that help manage and pass data between components. They are used to control how the user interface (UI) looks and behaves. Here's an overview of both:

1. State

State is an object that holds data or information about the component's behavior and can change over time. When the state of a component changes, React automatically re-renders that component to reflect the new state

- **State is local**: Each component can have its own state.
- **State is mutable**: It can be updated during the lifecycle of the component.

In **Functional Components**, you manage state using the useState() hook, while in **Class Components**, you use this.state and this.setState().

Example of State in Functional Component (using useState):

```
import React, { useState } from 'react';
const Counter = () => {
  // Declare state using useState hook
```

```
    const [count, setCount] = useState(0);
    // Function to increment the count
    const increment = () => {
        setCount(count + 1);
    };
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={increment}>Increment</button>
        </div>
    );
};
export default Counter;
```

In this example:

- The useState(0) hook initializes the count state to 0.
- setCount is the function used to update the state (count).

Example of State in Class Component:

```
import React, { Component } from 'react';
class Counter extends Component {
    constructor(props) {
        super(props);
        // Initialize state in the constructor
        this.state = { count: 0 };
    }
    // Function to increment the count
    increment = () => {
        this.setState({ count: this.state.count + 1 });
    };
```

```
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
export default Counter;
```

In this example:

- The this.state is used to store the state (count).
- this.setState() is used to update the state.

2. Props

Props (short for **properties**) are read-only data that are passed from a parent component to a child component. Props are used to pass information, including data and event handlers, between components. A component cannot modify its own props, but it can pass them to other components as needed.

- **Props are immutable**: They cannot be changed by the component receiving them.
- **Props allow communication** between components by sending data down the component tree.

Example of Props in Functional Component:

```
import React from 'react';
const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

```
const App = () => {
   return <Greeting name="John" />;
};
export default App;
```

In this example:

- The Greeting component receives the name prop from the parent (App component).
- props.name is used inside the Greeting component to display the name passed from the parent.

Example of Props in Class Component:

```
import React, { Component } from 'react';
class Greeting extends Component {
   render() {
      return <h1>Hello, {this.props.name}!</h1>;
   }
}
class App extends Component {
   render() {
      return <Greeting name="John" />;
   }
}
export default App;
```

In this example:

- The Greeting component receives the name prop from the parent (App component).
- this.props.name is used inside the Greeting component to display the name passed from the parent.

UNIT- III

REACT JS

State vs Props

Feature	State	Props
Definition	Data that is owned and managed by the component.	Data that is passed down from parent to child components.
Mutability	Mutable. Can be changed using setState() (class) or useState() (functional).	Immutable. Cannot be changed by the child component.
Scope	Local to the component where it is defined.	Can be passed to child components, making them global within the component tree.
Use Case	Used for internal state that affects the component's behavior or appearance.	Used to pass data or event handlers from a parent to a child.
Update Trigger	Re-renders the component when state changes.	A change in props will cause the child component to re-render.
Initialization	Initialized within the component itself.	Initialized in the parent component and passed down to the child.

Example: Combining State and Props

In a real-world scenario, we often combine both state and props in a component to manage data and pass it around.

```
import React, { useState } from 'react';
// Child Component
const Message = (props) => {
    return <p>{props.text}</p>;
};
```

UNIT- III

REACT JS

```
// Parent Component
const App = () => {
  const [message, setMessage] = useState("Welcome to React!");
  return (
    <div>
      <Message text={message} />
      <button onClick={() => setMessage("You clicked the button!")}>
        Change Message
      </button>
    </div>
  );
};
export default App;
```

In this example:

- The **parent component (App)** manages the state message using useState().
- The **child component (Message)** receives the message as a prop and displays it.
- Clicking the button in the parent component updates the state (message), which automatically re-renders the parent and child components.

Conclusion

- **State** is used to manage data that can change over time and is local to the component.
- **Props** are used to pass data from a parent component to a child component and are immutable within the child component.

React Component Lifecycle

Class Component Lifecycle Methods

1. **Mounting**:
    - constructor(): Initializes state and binds methods.

- render(): Renders UI.
- componentDidMount(): Called after the component mounts.

2. **Updating**:
   - shouldComponentUpdate(): Determines if a re-render is needed.
   - render(): Re-renders the component.
   - componentDidUpdate(): Called after the component updates.
3. **Unmounting**:
   - componentWillUnmount(): Cleanup before removal from the DOM.
4. **Error Handling**:
   - getDerivedStateFromError(), componentDidCatch(): Handle errors in child components.

Functional Component with Hooks (Using useEffect)

1. **Mounting**:
   - useEffect(() => { ... }, []): Runs once when the component mounts (similar to componentDidMount).
2. **Updating**:
   - useEffect(() => { ... }, [dependency]): Runs when a specific state/prop changes (similar to componentDidUpdate).
3. **Unmounting**:
   - useEffect(() => { return () => { ... }; }, []): Cleanup on unmount (similar to componentWillUnmount).

Comparison Table:

Feature	Class Component	Functional Component
Mounting	componentDidMount()	useEffect(() => { ... }, [])
Updating	shouldComponentUpdate()	useEffect(() => { ... }, [dep])

UNIT- III

REACT JS

Feature	Class Component	Functional Component
Unmounting	componentWillUnmount()	useEffect(() => { return cleanup }, [])
Error Handling	componentDidCatch()	Error boundaries

In summary, **class components** use lifecycle methods to manage state and effects, while **functional components** leverage useEffect for the same purpose with simpler syntax.

Here are simple examples for both **class components** and **functional components** demonstrating the lifecycle methods and hooks:

Class Component Example (Using Lifecycle Methods)

```
import React, { Component } from 'react';
class LifecycleExample extends Component {
constructor(props) {
   super(props);
   this.state = {
     message: 'Hello, World!',
   };
   console.log('Constructor: Component initialized');
 }
 // Called after the component mounts
 componentDidMount() {
 console.log('componentDidMount: Component mounted');
 }
 // Called before the component updates
 shouldComponentUpdate(nextProps, nextState) {
 console.log('shouldComponentUpdate: Checking if re-render is needed');
 return true; // Allows the re-render
 }
```

```
  // Called after the component updates
  componentDidUpdate(prevProps, prevState) {
  console.log('componentDidUpdate: Component updated');
  }
  // Called before the component unmounts
  componentWillUnmount() {
  console.log('componentWillUnmount: Cleanup before removal');
  }
  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <button
          onClick={() => this.setState({ message: 'Hello, React!' })}
        >
          Change Message
        </button>
      </div>
    );
  }
}
export default LifecycleExample;
```

Explanation:

1. **constructor()** initializes the state.
2. **componentDidMount()** runs once after the component is mounted.
3. **shouldComponentUpdate()** determines whether the component should re-render.
4. **componentDidUpdate()** runs after the component is updated.
5. **componentWillUnmount()** is used for cleanup before the component is removed.

UNIT- III

REACT JS

Functional Component Example (Using useEffect Hook)

```
import React, { useState, useEffect } from 'react';
const LifecycleExample = () => {
  const [message, setMessage] = useState('Hello, World!');
  // ComponentDidMount and ComponentDidUpdate
  useEffect(() => {
    console.log('useEffect: Component mounted or updated');
    // Cleanup function (ComponentWillUnmount)
    return () => {
      console.log('Cleanup: Component will unmount');
    };
  }, [message]); // Runs when 'message' changes
  return (
    <div>
      <h1>{message}</h1>
      <button onClick={() => setMessage('Hello, React!')}>
        Change Message
      </button>
    </div>
  );
};
export default LifecycleExample;
```

Explanation:

1. **useEffect()** is used for mounting and updating effects.
2. It runs after the component is mounted and when the message state changes (like componentDidMount and componentDidUpdate).
3. The **cleanup function** inside useEffect works like componentWillUnmount() and runs before the component is unmounted or when message changes again.

UNIT- III

REACT JS

Comparison:

- **Class Component**: Uses specific lifecycle methods like componentDidMount, componentDidUpdate, etc.
- **Functional Component**: Uses the useEffect hook for managing side effects and cleanup.

1. Rendering List and Portals

In **React.js**, **rendering lists** and using **portals** are common techniques used in component rendering and DOM manipulation.

Rendering a List in React

Rendering lists is typically done using the .map() function.

Example: Rendering a list of users

```
import React from 'react';
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user, index) => (
        <li key={user.id || index}>{user.name}</li>
      ))}
    </ul>
  );
};
// Example usage:
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
];
export default function App() {
```

```
  return <UserList users={users} />;
}
```

Notes:

- Always use a unique key when rendering lists.
- Avoid using index as key unless no unique id is available.

2. Portals in React

Portals allow you to **render children into a DOM node that exists outside the parent component's hierarchy**.

Example: Modal using React Portal

Step 1: Create a DOM node in your public/index.html

```
<!-- Add this outside the root div -->
<div id="modal-root"></div>
```

Step 2: Create the Portal Component

```
import React from 'react';
import ReactDOM from 'react-dom';
const Modal = ({ children }) => {
return ReactDOM.createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById('modal-root')
  );
};
export default Modal;
```

Step 3: Use it in your app

```
import React, { useState } from 'react';
```

UNIT- III

REACT JS

```
import Modal from './Modal';
export default function App() {
  const [showModal, setShowModal] = useState(false);
  return (
    <div>
      <button onClick={() => setShowModal(true)}>Open Modal</button>
      {showModal && (
        <Modal>
          <div>
            <h2>Hello from Portal!</h2>
            <button onClick={() => setShowModal(false)}>Close</button>
          </div>
        </Modal>
      )}
    </div>
  );
}
```

Error Handling

Error handling in **React.js** is important to keep your app stable and user-friendly. React provides several ways to handle errors depending on **where** and **how** they occur:

1. Try-Catch in Event Handlers or Async Functions

For errors in functions like API calls or button clicks:

```
const handleClick = async () => {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
```

```
    console.error('Something went wrong:', error);
    alert('Failed to fetch data!');
  }
};
```

2. Error Boundaries (Class Components Only)

For rendering errors (e.g., a component crashes):

Step 1: Create an Error Boundary Component

```
import React, { Component } from 'react';
class ErrorBoundary extends Component {
constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    console.error('ErrorBoundary caught an error', error, info);
  }
  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong □</h2>;
    }
    return this.props.children;
  }
}
export default ErrorBoundary;
```

Step 2: Wrap it around risky components

```
import ErrorBoundary from './ErrorBoundary';
import BuggyComponent from './BuggyComponent';
function App() {
  return (
    <div>
      <ErrorBoundary>
        <BuggyComponent />
      </ErrorBoundary>
    </div>
  );
}
```

3. Handling Errors in Functional Components with Hooks

If you're using hooks, use useEffect() + try-catch or use libraries like react-error-boundary:

Note: npm install react-error-boundary (Install in Terminal)

```
import { ErrorBoundary } from 'react-error-boundary';
function ErrorFallback({ error, resetErrorBoundary }) {
return (
    <div>
      <p>Something went wrong:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>Try again</button>
    </div>
  );
}
function MyComponent() {
  throw new Error('Oops!');
}
```

```
function App() {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <MyComponent />
    </ErrorBoundary>
  );
}
```

4. 404 and Network Errors (Route Not Found)

Use React Router for 404 handling:

```
<Route path="*" element={<NotFound />} />
```

Handle API/network errors in try-catch as shown earlier.

Summary

Error Type	Handling Method
API / Network Errors	try-catch in async functions
Rendering Errors	Error Boundaries (class or library)
Form Validation Errors	Conditional rendering, e.g. {error && Error }
Route Not Found	React Router path="*" route

Routers

In **React**, routing is managed using a library called **React Router** (react-router-dom) for web apps. It allows navigation between different components/pages without refreshing the whole page—this is known as **client-side routing**.

UNIT- III

REACT JS

Step-by-Step: React Router Setup

1. Install React Router

npm install react-router-dom

2. Basic Routing Example

□ ***App.js***

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
import NotFound from './pages/NotFound';
function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link> |
        <Link to="/contact">Contact</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
export default App;
```

UNIT- III

REACT JS

3. Create Page Components

□ *pages/Home.js*

```
const Home = () => <h2>□ Home Page</h2>;
export default Home;
```

□ *pages/About.js*

```
const About = () => <h2>ℹ About Us</h2>;
export default About;
```

□ *pages/Contact.js*

```
const Contact = () => <h2>□ Contact Us</h2>;
export default Contact;
```

□ *pages/NotFound.js*

```
const NotFound = () => <h2>✖ 404 Page Not Found</h2>;
export default NotFound;
```

□ Optional: Redirects and Nested Routes

➤ Redirect Example

```
import { Navigate } from 'react-router-dom';
<Route path="/old-contact" element={<Navigate to="/contact" replace />} />
```

➤ Nested Routes Example

```
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<Profile />} />
  <Route path="settings" element={<Settings />} />
</Route>
```

Use <Outlet /> inside Dashboard.js to render nested components.

UNIT- III

REACT JS

Summary Table

Feature	Usage
Define route paths	<Route path="/path" element={<Comp />} />
Navigate links	<Link to="/about">About</Link>
404 fallback	<Route path="*" element={<NotFound />}/>
Redirect	<Navigate to="/new" replace />
Nested Routes	Use <Outlet /> and child <Route>s

React Router demo project with working routes, navigation, a 404 page, and nested routes

Project Structure

react-router-demo/

├── App.js

├── index.js

└── pages/

├── Home.js

├── About.js

├── Contact.js

├── Dashboard.js

UNIT- III

REACT JS

├── **Profile.js**

├── **Settings.js**

└── **NotFound.js**

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.getElementById('root'));
```

App.js

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
import Dashboard from './pages/Dashboard';
import Profile from './pages/Profile';
import Settings from './pages/Settings';
import NotFound from './pages/NotFound';
function App() {
  return (
    <Router>
      <nav style={{ padding: '10px', background: '#eee' }}>
        <Link to="/">Home</Link> |
        <Link to="/about"> About</Link> |
        <Link to="/contact"> Contact</Link> |
        <Link to="/dashboard"> Dashboard</Link>
      </nav>
```

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        {/* Nested Routes */}
        <Route path="/dashboard" element={<Dashboard />}>
          <Route path="profile" element={<Profile />} />
          <Route path="settings" element={<Settings />} />
        </Route>
        {/* 404 Page */}
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
export default App;
```

pages/Home.js

```
const Home = () => <h2> Welcome to Home Page</h2>;
export default Home;
```

pages/About.js

```
const About = () => <h2>ℹ About Us</h2>;
export default About;
```

pages/Contact.js

```
const Contact = () => <h2> Contact Page</h2>;
export default Contact;
```

pages/Dashboard.js

```
import React from 'react';
import { Link, Outlet } from 'react-router-dom';
```

UNIT- III

REACT JS

```
const Dashboard = () => (
  <div>
    <h2> Dashboard</h2>
    <Link to="profile">Profile</Link> | <Link to="settings">Settings</Link>
    <Outlet />
  </div>
);
export default Dashboard;
```

pages/Profile.js

```
const Profile = () => <p> User Profile Section</p>;
export default Profile;
```

pages/Settings.js

```
const Settings = () => <p> Account Settings</p>;
export default Settings;
```

pages/NotFound.js

```
const NotFound = () => <h2>✕404 - Page Not Found</h2>;
export default NotFound;
```

To Run This Project

npx create-react-app react-router-demo
cd react-router-demo
npm install react-router-dom
Replace the default files with the above and run:
npm start

UNIT- III

REACT JS

Redux and Redux Saga

What is Redux?

Redux is a **state management** library that helps manage and centralize application state. It uses:

- **Store** – Global state container
- **Actions** – Events that describe state changes
- **Reducers** – Functions that update the store based on actions

What is Redux-Saga?

Redux-Saga is a **middleware** library used to handle **asynchronous side effects** (like API calls) in a Redux app using generator functions (function*).

It allows:

- Delayed dispatches
- Asynchronous API requests
- Background processes (e.g., polling, debouncing)

Step-by-Step Setup: Redux + Redux Saga

1. Install Required Packages

npm install redux react-redux redux-saga

2. Project Structure

src/
├── redux/
│ ├── actions.js
│ ├── reducers.js
│ ├── sagas.js

```
│   └── store.js
├── components/
│   └── Counter.js
└── App.js
```

Example Use Case: Counter with Saga Delay

redux/actions.js

```
export const INCREMENT = "INCREMENT";
export const DECREMENT = "DECREMENT";
export const INCREMENT_ASYNC = "INCREMENT_ASYNC";
export const increment = () => ({ type: INCREMENT });
export const decrement = () => ({ type: DECREMENT });
export const incrementAsync = () => ({ type: INCREMENT_ASYNC });
```

redux/reducers.js

```
import { INCREMENT, DECREMENT } from './actions';
const initialState = { count: 0 };
export const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT: return { count: state.count + 1 };
    case DECREMENT: return { count: state.count - 1 };
    default: return state;
  }
};
```

redux/sagas.js

```
import { put, takeEvery, delay } from 'redux-saga/effects';
import { INCREMENT_ASYNC, increment } from './actions';
function* incrementAsyncSaga() {
  yield delay(1000); // wait 1 sec
  yield put(increment());
```

```
}
export function* rootSaga() {
  yield takeEvery(INCREMENT_ASYNC, incrementAsyncSaga);
}
```

redux/store.js

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import { counterReducer } from './reducers';
import { rootSaga } from './sagas';
const sagaMiddleware = createSagaMiddleware();
export const store = createStore(
  counterReducer,
  applyMiddleware(sagaMiddleware)
);
sagaMiddleware.run(rootSaga);
```

components/Counter.js

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementAsync } from '../redux/actions';
const Counter = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => dispatch(increment())}>+ Increment</button>
      <button onClick={() => dispatch(decrement())}>- Decrement</button>
      <button onClick={() => dispatch(incrementAsync())}>+ Async (Saga)</button>
    </div>
```

```
  );
};
export default Counter;
```

App.js

```
import React from 'react';
import { Provider } from 'react-redux';
import { store } from './redux/store';
import Counter from './components/Counter';
const App = () => (
  <Provider store={store}>
    <h1>Redux + Redux-Saga Demo</h1>
    <Counter />
  </Provider>
);
export default App;
```

Summary

Tool	Purpose
redux	Global state management
react-redux	Connect Redux with React components
redux-saga	Handle async logic via generators

Immutable.js

Immutable.js is a JavaScript library from Facebook designed to create **immutable** data structures like Lists, Maps, Sets, etc. In **React**, it's often used to improve performance and ensure predictable state updates.

UNIT- III

REACT JS

Why Use in React?

1. **Immutability by** Immutable.js **Design**: Prevents accidental mutations of state.
2. **Performance Optimization**: Fast comparisons via === (reference equality).
3. **Cleaner Code**: Chained operations like map(), filter(), etc., are supported out of the box.
4. **Easy Undo/Redo Logic**: Track state history easily.

Common Immutable.js Data Structures

Immutable Structure	JavaScript Equivalent
Map	Object {}
List	Array []
Set	Set

✅Example Usage in React

npm install immutable

1. Using Map for state

```
import { Map } from 'immutable';
import React, { useState } from 'react';
function Counter() {
  const [state, setState] = useState(Map({ count: 0 }));
  const increment = () => {
    setState(prevState => prevState.update('count', val => val + 1));
  };
  return (
    <div>
      <h2>Count: {state.get('count')}</h2>
      <button onClick={increment}>Increment</button>
```

```
    </div>
  );
}
```

2. Using List for arrays

```
import { List } from 'immutable';
const [list, setList] = useState(List([1, 2, 3]));
setList(prev => prev.push(4)); // Add an element immutably
```

Service Side Rendering

Server-Side Rendering (SSR) in React refers to the process of rendering React components on the server (Node.js) rather than in the browser. The server sends the fully rendered HTML to the client, improving performance and SEO. SSR is commonly used with **Next.js**, a React framework that makes SSR easy.

✅Benefits of Server-Side Rendering (SSR)

- **Faster First Paint (especially on slow devices)**
- **Improved SEO** (important for content-heavy apps)
- **Pre-rendered HTML** sent to the client, improving crawlability

How SSR Works in React (with Next.js)

1. **Request** sent to server (e.g., https://yourdomain.com/products)
2. Server runs React code and fetches necessary data
3. Server renders the React component to HTML
4. Server sends the HTML to the browser
5. Browser hydrates the page (adds event listeners)

Example using Next.js (React SSR Framework)

Install Next.js:

UNIT- III

REACT JS

```
npx create-next-app@latest my-ssr-app
cd my-ssr-app
npm run dev
```

Create a page with SSR:

```
// pages/products.js
export async function getServerSideProps() {
  const res = await fetch('https://fakestoreapi.com/products');
  const products = await res.json();
  return { props: { products } };
}
export default function Products({ products }) {
  return (
    <div>
      <h1>Products</h1>
      {products.map(p => (
        <div key={p.id}>
          <h2>{p.title}</h2>
          <p>{p.price}</p>
        </div>
      ))}
    </div>
  );
}
```

Key Concepts in SSR

Concept	Description
getServerSideProps	Runs **on every request** to fetch data on server

UNIT- III

REACT JS

Concept	Description
getStaticProps	For **Static Site Generation** (not SSR)
Hydration	Client React takes over server-rendered HTML
SEO Optimization	Meta tags, structured data improve SEO ranking

Unit Testing

Unit testing in React means testing individual components or functions in isolation to ensure they behave as expected.

✅Popular Libraries for React Unit Testing

Tool	Purpose
Jest	Test runner + assertion library
React Testing Library (RTL)	Interact with components like a user
Enzyme	(Legacy) Shallow rendering (not recommended anymore)

➡ The recommended stack today is **Jest + React Testing Library (RTL)**.

Install Required Packages

npm install --save-dev jest @testing-library/react @testing-library/jest-dom

Add this to your package.json:

```
"scripts": {
  "test": "jest"
}
```

UNIT- III

REACT JS

Sample React Component

Counter.js

```
import React, { useState } from 'react';
export default function Counter() {
const [count, setCount] = useState(0);
return (
   <div>
     <h1>Count: {count}</h1>
     <button onClick={() => setCount(count + 1)}>Increment</button>
   </div>
 );
}
```

Unit Test File

Counter.test.js

```
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';
import '@testing-library/jest-dom';

test('renders initial count', () => {
  render(<Counter />);
  expect(screen.getByText(/count: 0/i)).toBeInTheDocument();
});

test('increments count on button click', () => {
  render(<Counter />);
```

```
  fireEvent.click(screen.getByText(/increment/i));
  expect(screen.getByText(/count: 1/i)).toBeInTheDocument();
});
```

▸ Run Tests

npm test

You'll see output like:

PASS ./Counter.test.js
✓ renders initial count (xx ms)
✓ increments count on button click (xx ms)

Webpack

Webpack in React is a **module bundler** that compiles your JavaScript, CSS, images, and other assets into optimized bundles for the browser. It's commonly used in custom React setups when you're not using tools like **Create React App** or **Next.js**.

Why Use Webpack in React?

- Bundles JS, CSS, images, fonts, etc.
- Supports hot-reloading and development server
- Allows JSX, ES6+ transpiling via Babel
- Tree-shaking and optimization for production

✅Step-by-Step Setup: React + Webpack + Babel

Folder Structure

```
my-react-app/
├── public/
│   └── index.html
├── src/
```

UNIT- III

REACT JS

```
│   ├── App.js
│   └── index.js
├── webpack.config.js
├── .babelrc
├── package.json
```

1 Initialize Project

```
mkdir my-react-app && cd my-react-app
npm init -y
```

2 Install Dependencies

```
npm install react react-dom
npm install --save-dev webpack webpack-cli webpack-dev-server
npm install --save-dev babel-loader @babel/core @babel/preset-env @babel/preset-react
npm install --save-dev html-webpack-plugin
```

3 Babel Configuration (.babelrc)

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

4 Webpack Config (webpack.config.js)

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  mode: 'development',
```

```
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: 'babel-loader'
      },
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      }
    ]
  },
  resolve: {
    extensions: ['.js', '.jsx']
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html'
    })
  ],
  devServer: {
    static: './dist',
    hot: true,
    port: 3000
  }
};
```

5 React Files

public/index.html

UNIT- III

REACT JS

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

src/App.js

```
import React from 'react';

const App = () => {
  return <h1>Hello from Webpack + React</h1>;
};

export default App;
```

src/index.js

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import App from './App';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

UNIT- III

REACT JS

6 Add Scripts to package.json

```
"scripts": {
  "start": "webpack serve --open",
  "build": "webpack"
}
```

▸ Run the App

```
npm start
```

Open http://localhost:3000 to see your app.

CASE STUDY 1: Student Management App

Scenario:

Create a React app to manage student details:

- Add student
- Display list of students
- Delete student

Questions:

1. Create **components** (App, StudentList, StudentItem)
2. Use **state** to store student data
3. Pass data using **props**
4. Render list using **map()**
5. Handle events (add/delete)

CASE STUDY 2: Counter Application

Scenario:

Build a counter with:

- Increment
- Decrement
- Reset

Questions:

1. Use **state** to store count
2. Update state using event handlers
3. Explain **component lifecycle** (mount/update)
4. Display count using JSX

UNIT- III

REACT JS

CASE STUDY 3: E-Commerce Product Page

Scenario:

Display products with:

- Name
- Price
- Add to cart

Questions:

1. Use **JSX templating**
2. Render product list
3. Use **props** to pass product data
4. Manage cart using **state**
5. Use **keys while rendering list**

CASE STUDY 4: Multi-Page Website using Router

Scenario:

Create pages:

- Home
- About
- Contact

Questions:

1. Use **React Router**
2. Create navigation links
3. Switch between pages without reload
4. Explain routing concept

CASE STUDY 5: Login System with Error Handling

Scenario:

User enters login details:

- Show success if correct
- Show error if wrong

Questions:

1. Use **state for input fields**
2. Handle form submission
3. Implement **error handling**
4. Display error messages

UNIT- III

REACT JS

CASE STUDY 6: Todo List Application

Scenario:

User can:

- Add tasks
- Delete tasks
- Mark as completed

Questions:

1. Use **components** (Todo, TodoList)
2. Manage data using **state**
3. Render list dynamically
4. Use **event handling**

CASE STUDY 7: Redux-Based Cart System

Scenario:

Manage shopping cart globally

Questions:

1. Explain **Redux store, actions, reducers**
2. Use **Redux for state management**
3. Handle actions like add/remove item
4. Explain **Redux Saga (for async operations)**

CASE STUDY 8: Server-Side Rendering (SSR)

Scenario:

Improve performance of React app

Questions:

1. Explain **SSR concept**
2. Difference between CSR & SSR
3. Benefits of SSR

CASE STUDY 9: Immutable Data Handling

Scenario:

Update state without modifying original data

Questions:

1. Explain **immutability concept**
2. Use **Immutable.js or spread operator**

3. Why immutability is important in React

CASE STUDY 10: React App Testing & Build

Scenario:

Test and bundle React application

Questions:

1. Explain **unit testing in React**
2. Use tools like Jest (concept)
3. Explain **Webpack role**
4. Why bundling is required

UNIT- III: REACT JS:		
Introduction, Templating using JSX, Components, State and Props, Lifecycle of Components, Rendering List and Portals, Error Handling, Routers, Redux and Redux Saga, Immutable.js, Service Side Rendering, Unit Testing, Webpack.		
PART -A		
1.	What is ReactJS?	L1
2.	Define JSX with example.	L1
3.	What are components in React?	L1
4.	What is the use of props in React?	L2
5.	What is a state in React?	L1
6.	List the lifecycle methods in React.	L1
7.	Define Redux.	L1
8.	What is a portal in ReactJS?	L2
9.	What is server-side rendering?	L2
10.	What is the role of Webpack in React?	L2
PART-B		
1.	Explain the concept of components in React with example.	L2
2.	Describe the state and props in React with examples.	L2
3.	Explain lifecycle methods of a React component.	L2
4.	Write a React component to display a list of items.	L3
5.	What is Redux? How does it work with React?	L2
6.	Explain JSX templating with examples.	L2
7.	Discuss the use of routers in React applications.	L2
8.	Explain error handling techniques in React.	L2
9.	How is unit testing done in React applications?	L3
10.	Discuss the role and working of Webpack in ReactJS development.	L2

FULL STACK WEB DEVELOPMENT

UNIT –IV

NODE JS

First, solve the problem. Then, write the code.

— John Johnson

Unit -IV: NODE JS

1. Unit Overview

This unit introduces **Node.js**, a server-side JavaScript runtime, covering its setup, core concepts, modules, events, console usage, command utilities, integration with Express.js, and database access. Students will learn to build scalable, event-driven server-side applications and interact with databases using Node.js.

2. Objectives of the Unit

By the end of this unit, students should be able to:

- Understand the **Node.js environment** and its advantages for server-side development.
- Set up Node.js and use the **Node console and command-line utilities**.
- Work with **Node.js modules** for modular programming.
- Handle **events** and asynchronous operations efficiently.
- Develop **server-side applications using Express.js** framework.
- Perform **database operations** using Node.js with popular databases.

3. Learning Outcomes

After completing this unit, students will be able to:

- Write **server-side JavaScript programs** using Node.js.
- Execute Node.js commands and debug using the **Node console**.
- Structure applications using **Node modules**.
- Implement **event-driven programming** in Node.js applications.
- Build **Express.js applications** with routing, middleware, and server logic.
- Connect Node.js applications to **databases** and perform CRUD operations.

4. Importance of Studying this Unit

- Node.js enables **JavaScript to run on the server**, making it essential for full-stack web development.
- Event-driven, non-blocking architecture allows **high-performance, scalable web applications**.
- Knowledge of Node.js is critical for building **RESTful APIs, web services, and real-time applications**.
- Integration with **Express.js and databases** prepares students for professional **backend development roles**.

5. Key Concepts

- **NodeJS Overview:** Features, architecture, advantages, and use cases.
- **Basics and Setup:** Installing Node.js, setting environment variables, writing first script.
- **NodeJS Console:** REPL environment, console methods, debugging basics.
- **Command Utilities:** node, npm, npx, and other Node.js CLI tools.
- **NodeJS Modules:** Built-in modules, creating custom modules, module.exports and require.
- **NodeJS Concepts:** Asynchronous programming, callbacks, promises, and event loop.
- **NodeJS Events:** EventEmitter class, handling and emitting events.
- **NodeJS with ExpressJS:** Setting up Express server, routing, middleware, request/response handling.
- **NodeJS Database Access:** Connecting to databases (MongoDB, MySQL), performing CRUD operations.

UNIT- IV: NODE JS

Node.js Overview

Node.js is a **runtime environment** that allows you to run JavaScript **on the server-side**, outside of a browser. It's built on **Chrome's V8 JavaScript engine**, which makes it **fast and efficient** for building scalable and high-performance network applications.

What is Node.js?

- **Open-source**, cross-platform
- Executes JavaScript code **outside** a web browser
- Designed for **non-blocking**, event-driven servers
- Ideal for **data-intensive real-time applications** (like chat apps)

Key Features

Feature	Description
Asynchronous and Event-Driven	All APIs are non-blocking; Node.js handles multiple requests simultaneously.
Fast Execution	Built on Google Chrome's V8 engine, which compiles JavaScript to native machine code.
Single Programming Language	Use JavaScript both on the **frontend** and **backend**.
Package Ecosystem	Comes with **npm** (Node Package Manager) – the largest ecosystem of open-source libraries.
Cross-platform	Runs on Windows, macOS, and Linux.

Node.js Architecture

- **Single-threaded** with **event loop**
- Uses **non-blocking I/O** operations (vs blocking I/O in traditional systems)
- Can handle **thousands of concurrent connections** efficiently

UNIT- IV: NODE JS

Common Use Cases

- Real-time applications (e.g. **chat apps**, live updates)
- REST APIs and GraphQL servers
- Microservices
- Streaming services
- Command-line tools
- IoT applications

Example: Simple Node.js Server

```
// Load HTTP module
const http = require('http');

// Create HTTP server
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Node.js!\n');
});

// Start the server
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

npm – Node Package Manager

- Use npm to install third-party packages
- Example: npm install express installs Express.js (a popular Node.js web framework)

Popular Node.js Frameworks

- **Express.js** – Minimalist and flexible web framework
- **NestJS** – Scalable and structured, uses TypeScript
- **Fastify** – High-performance HTTP framework
- **Socket.io** – Real-time bidirectional event-based communication

UNIT- IV: NODE JS

Why Choose Node.js?

- Great for **real-time** apps (chat, collaboration tools)
- Easy to learn if you already know JavaScript
- Huge developer community
- Scalable for modern application

Node.js Basics and Setup

1. What You Need to Get Started

- A computer (Windows, macOS, Linux)
- Basic knowledge of JavaScript
- Internet connection to download Node.js

2. Installing Node.js

1. Go to the official Node.js website:
   https://nodejs.org/
2. You will see two versions:
   - **LTS (Long Term Support)** — recommended for most users and production
   - **Current** — latest features but less stable
3. Download the **LTS version** for your OS and install it using the installer.
4. After installation, verify by opening your terminal (Command Prompt, PowerShell, or Terminal) and typing:

```
node -v
npm -v
```

You should see the versions printed out, confirming the installation.

3. Your First Node.js Script

Create a file called app.js anywhere on your computer with the following content:

```
console.log('Hello, Node.js!');
```

Run this script in the terminal:

node app.js

You should see the output:

Hello, Node.js!

4. Running a Simple HTTP Server

Create a file called server.js with:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Node.js server!');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

Run the server:

node server.js

Open your browser and go to http://localhost:3000 to see the message.

5. Using npm (Node Package Manager)

- Initialize a new Node.js project:

npm init

This will prompt you to enter details about your project and create a package.json file.

- To install a package (example: Express.js):

npm install express

- The installed package goes into the node_modules folder and is recorded in package.json.

6. Basic Project Structure Example

my-node-app/
├── app.js
├── package.json
├── node_modules/

- app.js — Your main script
- package.json — Project metadata and dependencies
- node_modules/ — Installed packages

7. Helpful Commands

Command	Purpose
node <filename>	Run a Node.js file
npm init	Initialize a new Node.js project
npm install <name>	Install a package
npm start	Run the start script (defined in package.json)
npm uninstall <name>	Remove a package

Summary:

- Install Node.js and npm
- Run JavaScript files with node
- Create simple servers with built-in HTTP module
- Manage dependencies with npm

UNIT- IV: NODE JS

NodeJS Console

The Node.js Console refers to the command-line interface (CLI) or terminal environment where you can run Node.js commands and scripts.

Opening the Node.js Console

1. **Windows:**
    - Open **Command Prompt** or **PowerShell**.
    - Type node and press Enter.
2. **Mac/Linux:**
    - Open your terminal.
    - Type node and press Enter.

REPL Mode (Read-Eval-Print Loop)

Once you type node, you enter the **interactive Node.js console** (REPL mode). It allows you to run JavaScript line by line:

```
$ node
> 2 + 2
4
> console.log('Hello, Node.js!')
Hello, Node.js!
> .exit   # or press Ctrl+C twice to exit
```

Running Node.js Files

To run a Node.js script file:

1. Create a file named app.js:

```
console.log("Welcome to Node.js!");
```

2. Run it in the terminal:

```
node app.js
```

Output:

Welcome to Node.js!

Common Console Commands in Node.js with Examples

Command	Description	Example	Output
console.log()	Prints to stdout (standard output)	js console.log("Hello, Node.js!");	Hello, Node.js!
console.error()	Prints to stderr (error output)	js console.error("Something went wrong!");	Something went wrong! (in red)
console.warn()	Prints a warning message	js console.warn("This is a warning!");	This is a warning! (in yellow)
console.table()	Displays tabular data	js console.table([{ name: "Alice" }, { name: "Bob" }]);	Neatly formatted table
console.time() / timeEnd()	Measures execution time	js console.time("timer"); for(let i=0;i<100000;i++){} console.timeEnd("timer");	timer: X ms
console.clear()	Clears the console	js console.clear();	(Console gets cleared)

✅**You can test all of these in a file like:**

```
// app.js
console.log("Hello, Node.js!");
console.error("Something went wrong!");
console.warn("This is a warning!");
```

```
const data = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 }
];
console.table(data);

console.time("loop");
for (let i = 0; i < 1000000; i++) {}
console.timeEnd("loop");

console.clear(); // Comment this out to see earlier output
```

NodeJS Command Utilities

Basic Node.js Commands

Command	Description
node filename.js	Run a Node.js script file
node	Open Node.js REPL (interactive shell)
node --version or node -v	Show Node.js version installed
npm --version or npm -v	Show npm (Node Package Manager) version
npm init	Initialize a new package.json file interactively
npm init -y	Initialize package.json with default values
npm install package-name or npm i package-name	Install a package locally (in node_modules)
npm install -g package-name	Install a package globally

Command	Description
npm uninstall package-name	Remove a package
npm update	Update all packages
npm list or npm ls	List installed packages
npm outdated	Check for outdated packages
npm run script-name	Run custom scripts defined in package.json
npx command	Run package binaries without installing globally

Useful Node.js Utilities and Tools

1. nvm (Node Version Manager)

- Manage multiple Node.js versions on the same machine.
- Install nvm: https://github.com/nvm-sh/nvm
- Commands:
  - nvm install 18 (install Node.js v18)
  - nvm use 18 (switch to Node.js v18)
  - nvm ls (list installed Node versions)
  - nvm alias default 18 (set default Node version)

2. nodemon

- Utility that monitors changes in your source files and automatically restarts the Node.js app.
- Install: npm install -g nodemon
- Usage: nodemon app.js

3. pm2

- Process manager for Node.js apps, supports clustering, auto restarts, logs.

- Install: npm install -g pm2
- Basic usage:
    - pm2 start app.js
    - pm2 status
    - pm2 restart app
    - pm2 logs

4. eslint

- JavaScript linter to check code quality and style.
- Install: npm install -g eslint
- Initialize config: eslint --init
- Run linter: eslint yourfile.js

5. http-server

- Simple zero-configuration command-line HTTP server.
- Install: npm install -g http-server
- Usage: http-server ./public (serve static files)

Useful Node.js Debugging Commands

Command	Description
node --inspect filename.js	Start app with debugging enabled (Chrome DevTools can connect)
node --inspect-brk filename.js	Start app with debugging and break on first line
node debug filename.js	Start legacy debug mode (REPL debugging)
node --trace-warnings filename.js	Show stack traces on warnings

UNIT- IV: NODE JS

Miscellaneous Useful Commands

Command	Description
npm cache clean --force	Clear npm cache
npm doctor	Check environment and setup for common errors
npm audit	Run a security audit of your dependencies
npm audit fix	Fix vulnerabilities automatically
npm ci	Clean install node modules based on package-lock.json (for CI/CD)
npm start	Run the start script in package.json

NodeJS Modules

In Node.js, modules are reusable blocks of code whose scope is local by default but can be shared/exported and imported into other files. They help organize code into manageable pieces.

Types of Node.js Modules

1. Core Modules

Built-in modules shipped with Node.js. You don't need to install them separately.

Examples:

- fs — File system operations
- http — HTTP server/client
- path — File path utilities
- os — Operating system info
- events — Event emitter pattern
- crypto — Cryptography functions

Usage: const fs = require('fs');

2. Local Modules

Modules you create yourself within your project.

Example:
Create a file math.js:

```
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

Use it in another file:

```
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

3. Third-Party Modules

Modules installed via npm (Node Package Manager) from the npm registry.

Example: Installing and using express:

```
npm install express

const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000);
```

UNIT- IV: NODE JS

Module Systems in Node.js

CommonJS (used in Node.js by default)

1. Exporting a single function or object

```
// greet.js
module.exports = function() {
  return "Hello from CommonJS!";
};
```

Importing it:

```
// app.js
const greet = require('./greet');
console.log(greet());  // Output: Hello from CommonJS!
```

2. Exporting multiple things (properties)

```
// utils.js
module.exports = {
  sayHi: function() { return "Hi!"; },
  number: 123,
};
```

Importing it:

```
// app.js
const utils = require('./utils');
console.log(utils.sayHi());  // Output: Hi!
console.log(utils.number);   // Output: 123
```

3. Shortcut to export multiple properties

```
// helpers.js
exports.sayHello = function() { return "Hello!"; };
exports.value = 99;
```

ES Modules (modern JS using import/export)

1. Named exports (export multiple things by name)

```
// utils.js
export function sayHi() {
  return "Hi from ES Module!";
}
```

```
export const number = 456;
```

Importing named exports:

```
// app.js
import { sayHi, number } from './utils.js';
console.log(sayHi());  // Output: Hi from ES Module!
console.log(number);  // Output: 456
```

2. Default export (export a single default thing)

```
// greet.js
export default function() {
  return "Hello from default export!";
}
```

Importing default export:

```
// app.js
import greet from './greet.js';
console.log(greet());  // Output: Hello from default export!
```

Summary:

Concept	CommonJS	ES Modules
Export single	module.exports = fn	export default fn
Export multiple	module.exports = {} or exports.foo = ...	export function foo() {} or export const bar = ...
Import	const x = require()	import {x} from '...' or import x from '...'

NodeJS Concepts

1. What is Node.js?

- **Node.js** is a JavaScript runtime built on Chrome's V8 engine that lets you run JS code server-side.
- It uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient.

- Primarily used for building scalable network applications (web servers, APIs, real-time apps).

2. Asynchronous & Non-blocking I/O

- **Non-blocking I/O** means operations like reading files or making network requests don't block the main thread.
- Node.js uses **callbacks**, **Promises**, and **async/await** to handle these operations asynchronously.
- **Why important?** Prevents the application from freezing or waiting, enabling high concurrency.

Example (callback style):

```
const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('This prints before file content');
```

- Output order:

  This prints before file content
  (then file content)

3. Event Loop

- The **event loop** is the core mechanism that allows Node.js to perform non-blocking I/O.
- Node.js runs on a **single thread**, but can handle multiple requests by putting tasks in an **event queue**.
- The event loop checks this queue and executes callbacks as soon as the call stack is empty.

Phases of Event Loop:

Phase	Description
Timers	Executes callbacks scheduled by setTimeout and setInterval
I/O callbacks	Executes some system I/O callbacks
Idle, prepare	Internal operations
Poll	Retrieves new I/O events; executes them
Check	Executes callbacks scheduled by setImmediate
Close callbacks	Executes close event callbacks (e.g., socket close)

- Example of difference between setImmediate and setTimeout:

```
setTimeout(() => console.log('timeout'), 0);
setImmediate(() => console.log('immediate'));
```

Output order can differ based on phase timing, but setImmediate often runs before zero-delay timeout.

4. Single Threaded but Highly Scalable

- Node.js handles multiple connections on a single thread using the event loop.
- Unlike traditional multi-threaded servers (like Apache), Node.js avoids the overhead of thread context switching.
- For CPU-heavy operations, Node.js offers **Worker Threads** or offloads tasks externally (e.g., to a database).

5. Modules

- Node.js uses the **CommonJS** module system.
- Each file is treated as a separate module with its own scope.
- Use require() to import and module.exports to export.

Example:

```
// greet.js
function sayHello(name) {
  return `Hello, ${name}!`;
}
module.exports = sayHello;

// app.js
const greet = require('./greet');
console.log(greet('Alice')); // Hello, Alice!
```

ES Modules (ESM) Support

- Node.js now also supports ES6 module syntax (import/export), but requires .mjs extension or "type": "module" in package.json.
- E.g.,

```
// greet.mjs
export function sayHello(name) {
  return `Hello, ${name}!`;
}

// app.mjs
import { sayHello } from './greet.mjs';
console.log(sayHello('Alice'));
```

6. Core Modules

Some core built-in modules to know:

Module	Description
fs	File system operations (read/write files)
http	Creating HTTP servers and clients
path	Handling file paths in a platform-independent way

Module	Description
os	Operating system info (CPU, memory, network)
crypto	Encryption, hashing, signing
events	EventEmitter implementation
stream	Handling streaming data (files, network, etc.)

7. NPM (Node Package Manager)

- Comes bundled with Node.js.
- Used for installing third-party libraries or publishing your own packages.
- Uses package.json to track dependencies, scripts, and metadata.
- Commands:
  - npm init → Create a new package.json.
  - npm install <package> → Installs and adds dependency.
  - npm update → Updates packages.
  - npm uninstall <package> → Removes dependency.

8. Creating HTTP Servers

Node.js allows building simple servers without frameworks:

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('<h1>Welcome to Node.js server</h1>');
  } else {
    res.writeHead(404);
    res.end('Not found');
  }
});
```

```
server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

9. Callbacks, Promises, Async/Await

Callbacks

- Basic async pattern but can cause “callback hell” if nested deeply.

Promises

- Represents a future value.
- Can be chained, improving readability.

```
const fs = require('fs').promises;

fs.readFile('file.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Async/Await

- Syntactic sugar over promises, easier to read/write async code.

```
async function readFile() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
readFile();
```

10. EventEmitter

- Node.js uses **EventEmitter** to handle events.
- Many core modules (like http) inherit from EventEmitter.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('start', () => {
  console.log('Started');
});

emitter.emit('start');
```

11. Streams

- Streams allow processing data piece-by-piece instead of all at once.
- Useful for large files or continuous data (videos, audio, network packets).

Types:

- **Readable:** source you can read from (fs.createReadStream)
- **Writable:** destination you can write to (fs.createWriteStream)
- **Duplex:** both readable and writable (network sockets)
- **Transform:** modifies data as it passes through

Example of reading a file stream:

```
const fs = require('fs');
const readStream = fs.createReadStream('largefile.txt', 'utf8');

readStream.on('data', chunk => {
  console.log('Received chunk:', chunk);
});
```

```
readStream.on('end', () => {
  console.log('Finished reading');
});
```

12. Buffer

- Buffers are used to work with raw binary data.
- Can be created from strings or arrays.

```
const buf = Buffer.from('Hello');
console.log(buf); // <Buffer 48 65 6c 6c 6f>
```

Useful for handling network protocols, file I/O, encryption.

13. Process & Global Objects

- process provides info and control over the running Node.js process:
  - process.argv — Command-line arguments.
  - process.env — Environment variables.
  - process.exit() — Exit the process.
  - Events: exit, uncaughtException, SIGINT.

Example:

```
console.log(process.argv);
```

- Global objects like __dirname, __filename, global, console.

14. Error Handling

- Synchronous code: use try/catch.
- Async callbacks: handle errors in callback parameters ((err, result) => {}).
- Promises: use .catch() or try/catch with async/await.
- Handle uncaught exceptions globally:

```
process.on('uncaughtException', (err) => {
  console.error('Unhandled Exception', err);
  process.exit(1); // recommended to restart
```

```
});
```

15. Worker Threads (Advanced)

- For CPU-intensive tasks that block the event loop.
- Allows multi-threading in Node.js.

```
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js');

worker.on('message', message => console.log(message));
worker.postMessage('start');
```

16. Debugging & Profiling

- Use built-in debugger:

  node inspect app.js

- Or use Chrome DevTools:

  node --inspect app.js

- Profiling tools: clinic, node --prof.

NodeJS Events

1. What Are Events in Node.js?

- Node.js uses an **event-driven architecture**.
- An **event** is a signal that something happened — like a user click, a server request, or a file read completion.
- Node.js apps listen for events and react when they occur.
- This is core to Node.js's **non-blocking, asynchronous** behavior.

2. EventEmitter Class

- The main way to work with events in Node.js is through the **EventEmitter** class from the events module.
- Almost all Node.js core modules (like HTTP, Streams, Sockets) inherit from EventEmitter.
- An **EventEmitter** object emits named events that cause functions ("listeners" or "handlers") to be called.

3. Basic Usage of EventEmitter

Import the module and create an emitter:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
```

Register an event listener (subscribe):

```
emitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});
```

Emit the event (publish):

```
emitter.emit('greet', 'Alice');  // Output: Hello, Alice!
```

- on() listens for every time the event occurs.
- emit() triggers the event and calls all listeners registered for it.

4. Common Methods of EventEmitter

Method	Description
on(event, fn)	Register a listener for the event
once(event, fn)	Register a one-time listener; auto-removed after first call
emit(event, [...args])	Emit the event, calling all listeners with optional args

Method	Description
removeListener(event, fn)	Remove a specific listener for an event
removeAllListeners(event)	Remove all listeners for an event
listenerCount(event)	Get the number of listeners registered for an event
listeners(event)	Returns an array of listeners for an event

5. once() vs on()

- on() will call the listener every time the event fires.
- once() will call the listener only the **first time** the event fires, then remove it.

Example:

```
emitter.once('init', () => console.log('Initialization done'));
emitter.emit('init'); // Prints once
emitter.emit('init'); // Does nothing
```

6. Handling Errors

- error is a special event.
- If an error event is emitted and there is no listener for it, Node.js will **throw and crash**.
- Always add an error listener to catch errors gracefully.

Example:

```
emitter.on('error', (err) => {
  console.error('Error occurred:', err.message);
});


emitter.emit('error', new Error('Something went wrong'));
```

7. Example: Custom Event Emitter

```
const EventEmitter = require('events');
```

```
class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

// Listener function
myEmitter.on('start', () => {
console.log('Started processing');
});

myEmitter.on('data', (data) => {
  console.log('Data received:', data);
});

myEmitter.on('end', () => {
  console.log('Processing ended');
});

// Emit events
myEmitter.emit('start');
myEmitter.emit('data', 'Some payload');
myEmitter.emit('end');
```

8. EventEmitter Max Listeners

- By default, an EventEmitter warns if more than **10 listeners** are added to the same event to avoid potential memory leaks.
- You can change this limit with:

```
emitter.setMaxListeners(20);
```

- Or disable the warning:

```
emitter.setMaxListeners(0);
```

9. Event Propagation and prependListener

- You can register a listener that runs **before** others with:

```
emitter.prependListener('event', listenerFn);
```

- Similar to on(), but called before existing listeners.

10. Real-World Uses of Events in Node.js

- **HTTP server:** Emits events like 'request', 'close'.
- **Streams:** Emit 'data', 'end', 'error'.
- **File system:** Emit 'open', 'close', 'change'.
- **Child processes:** Emit 'exit', 'error'.
- **Custom modules:** Implement event emitters to notify about async operations.

11. Event Loop & EventEmitter

- The **event loop** manages when event callbacks run.
- When an event is emitted, its listeners are queued and run on the next tick of the event loop.
- EventEmitter is the basis for Node's asynchronous event-driven I/O model.

12. Advanced: Using once with Promises (Node.js v11+)

Node.js provides a built-in way to wait for an event once, using events.once:

```
const { once } = require('events');
async function run() {
  const myEmitter = new EventEmitter();

  setTimeout(() => myEmitter.emit('finish'), 1000);

  await once(myEmitter, 'finish');
  console.log('Finished event detected');
}
run();
```

UNIT- IV: NODE JS

Summary

- Events are the heart of Node.js's asynchronous programming.
- EventEmitter class lets you create, emit, and listen for events.
- Events enable modular, decoupled, and scalable code.
- Always handle 'error' events to prevent app crashes.
- Use once() to listen for single-time events.
- Understand the event loop to grasp how and when event callbacks execute.

NodeJS with ExpressJS

What is Express.js?

- **Express.js** is a minimal and flexible **web application framework** for Node.js.
- It simplifies building APIs and web apps by handling:
  - Routing
  - Middleware
  - Request & Response handling
  - Templating (if needed)
- It's **built on top of Node.js's HTTP module**, but with much less boilerplate.

Installing Express

```
npm init -y
npm install express
```

Basic Express Server

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
```

```
});
```

Core Concepts of Express.js

1. Routing

- Define how your app responds to HTTP requests at different endpoints and methods.

```
app.get('/users', (req, res) => res.send('Get users'));
app.post('/users', (req, res) => res.send('Create user'));
app.put('/users/:id', (req, res) => res.send(`Update user ${req.params.id}`));
app.delete('/users/:id', (req, res) => res.send(`Delete user ${req.params.id}`));
```

2. Middleware

- Functions that execute **in sequence** during the request lifecycle.
- Can:
  - Modify req, res
  - End the request-response cycle
  - Call the next middleware

Types of Middleware:

- Application-level
- Router-level
- Error-handling
- Built-in (express.json(), express.static())
- Third-party (e.g. cors, morgan)

```
app.use(express.json()); // Built-in middleware to parse JSON body
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

3. Request and Response

- req (request): Contains HTTP request info (headers, query params, body, etc.)
- res (response): Used to send a response (HTML, JSON, status codes)

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  res.status(200).json({ message: `Welcome, ${username}` });
});
```

4. Serving Static Files

```
app.use(express.static('public')); // Now you can access public/index.html
```

5. Routing with Parameters

```
app.get('/user/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

6. Route Grouping using Router

```
const userRouter = express.Router();

userRouter.get('/', (req, res) => res.send('All users'));
userRouter.post('/', (req, res) => res.send('Create user'));

app.use('/api/users', userRouter);
```

Connecting with MongoDB (via Mongoose)

```
npm install mongoose
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/testdb')
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB error', err));
```

Handling Errors

```
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
  res.status(500).send('Something broke!');
});
```

Project Structure Example

```
project/
├── app.js
├── routes/
│   └── userRoutes.js
├── controllers/
│   └── userController.js
├── models/
│   └── User.js
├── middleware/
│   └── authMiddleware.js
```

Sample Features with Express

✅**Authentication (Login example):**

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  // Dummy check
  if (username === 'admin' && password === '123') {
    return res.json({ token: 'dummy-jwt-token' });
  }
  res.status(401).send('Unauthorized');
});
```

Sending JSON Response

```
app.get('/json', (req, res) => {
  res.json({ name: "Express", type: "Framework" });
});
```

⏳ **Middlewares for Logging**

```
const morgan = require('morgan');
```

```
app.use(morgan('dev'));  // logs requests like: GET /users 200 10ms
```

Handling 404

```
app.use((req, res) => {
  res.status(404).send('Page Not Found');
});
```

Summary Table

Concept	Express.js Equivalent
HTTP Server	app.listen(port, callback)
Routing	app.get(), app.post(), etc.
Middleware	app.use()
Static Files	express.static()
Parsing JSON	express.json()
Modular Routing	express.Router()
Error Handling	Custom middleware with 4 params
MongoDB Support	Via mongoose

NodeJS Database Access

Node.js can interact with **any type of database** using appropriate drivers or ORMs.

Types of Databases

Type	Examples	Data Model	Use Case
SQL	MySQL, PostgreSQL, SQLite	Table/Rows	Structured, relational data

Type	Examples	Data Model	Use Case
NoSQL	MongoDB, Redis, Cassandra	Document/Key-Value	Flexible, unstructured data

1. MongoDB (NoSQL)

Setup

npm install mongoose

Connecting with Mongoose

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/testdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('Mongo Error:', err));
```

Creating a Schema and Model

```
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

const User = mongoose.model('User', userSchema);
```

CRUD Operations

```
// Create
await User.create({ name: 'John', email: 'john@example.com', age: 25 });
```

```
// Read
const users = await User.find({ age: { $gt: 18 } });

// Update
await User.updateOne({ name: 'John' }, { age: 26 });

// Delete
await User.deleteOne({ name: 'John' });
```

2. MySQL (SQL)

Setup

```
npm install mysql2
```

Connect to MySQL

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'testdb'
});

connection.connect(err => {
  if (err) throw err;
  console.log('MySQL Connected');
});
```

CRUD Operations

```
// Create
connection.query(
```

```
  'INSERT INTO users (name, email) VALUES (?, ?)',
  ['Alice', 'alice@example.com'],
  (err, results) => {
    if (err) throw err;
    console.log('Inserted:', results.insertId);
  }
);

// Read
connection.query('SELECT * FROM users', (err, rows) => {
  console.log(rows);
});

// Update
connection.query(
  'UPDATE users SET name = ? WHERE id = ?',
  ['Bob', 1],
  (err, result) => {
    console.log('Updated:', result.affectedRows);
  }
);

// Delete
connection.query('DELETE FROM users WHERE id = ?', [1]);
```

3. PostgreSQL

Setup

```
npm install pg
```

Connect and Query

```
const { Client } = require('pg');
```

```
const client = new Client({
  user: 'postgres',
  host: 'localhost',
  database: 'testdb',
  password: 'password',
  port: 5432
});

client.connect()
  .then(() => console.log('PostgreSQL Connected'))
  .catch(err => console.error(err));

// Query
client.query('SELECT * FROM users', (err, res) => {
  console.log(res.rows);
});
```

CASE STUDY 1: Student Management System (Node + Express)

Scenario:

Build a backend system to:

- Add student
- View all students
- Delete student

Questions:

1. Setup **Node.js project**
2. Create **Express server**
3. Define routes:
    - GET /students
    - POST /students
    - DELETE /students/:id
4. Use **modules** to organize code
5. Store data in array or database

CASE STUDY 2: Simple REST API

Scenario:

Create an API for a product system:

- Get products
- Add product

Questions:

1. Use **Express.js**
2. Create REST endpoints
3. Handle JSON data
4. Use **middleware**

CASE STUDY 3: User Login System

Scenario:

User sends login request:

- Validate username & password

Questions:

1. Use **request & response objects**
2. Implement **logic using Node.js**
3. Send response using res.send()
4. Handle errors

CASE STUDY 4: Event Handling System

Scenario:

Create a system where:

- Event is triggered when user logs in

Questions:

1. Use **EventEmitter**
2. Create custom event
3. Trigger and handle event

CASE STUDY 5: File System Logger

Scenario:

Store user activity in a file

Questions:

1. Use **Node.js file system module (fs)**
2. Write logs to file
3. Read logs from file

CASE STUDY 6: Database Connection (Node + MongoDB/MySQL)

Scenario:

Store student data in database

Questions:

1. Connect Node.js with database
2. Perform:
    - Insert
    - Fetch
3. Use proper query handling

CASE STUDY 7: Command Line Utility

Scenario:

Create a CLI tool to:

- Add numbers
- Show result

Questions:

1. Use **process.argv**
2. Perform operations
3. Display result in console

CASE STUDY 8: Modular Application

Scenario:

Split application into multiple files

Questions:

1. Create separate **modules**
2. Use require() to import
3. Export functions

CASE STUDY 9: Express Middleware Example

Scenario:

Log every request coming to server

Questions:

1. Create **middleware function**
2. Use app.use()
3. Log request details

CASE STUDY 10: Error Handling in Node.js

Scenario:

Handle server errors properly

Questions:

1. Use **try-catch**
2. Handle async errors
3. Send proper error response

UNIT- IV: NODE JS		
NodeJS: NodeJS Overview, NodeJS - Basics and Setup, NodeJS Console, NodeJS Command Utilities, NodeJS Modules, NodeJS Concepts, NodeJS Events, NodeJS with ExpressJS, NodeJS Database Access.		
PART-A		
1.	What is NodeJS?	L1
2.	List the features of NodeJS.	L1
3.	What is npm in NodeJS?	L1
4.	What are modules in NodeJS?	L1
5.	Write syntax to import a module in NodeJS.	L3
6.	What is an event in NodeJS?	L1
7.	List built-in modules of NodeJS.	L1
8.	What is ExpressJS?	L1
9.	Write the use of 'fs' module in NodeJS.	L2
10.	Define NodeJS REPL.	L1
PART-B		
1.	Explain the architecture of NodeJS.	L2
2.	Describe how to set up a NodeJS application.	L2
3.	What are NodeJS modules? Explain types with examples.	L2
4.	Write a program in NodeJS to create a simple server.	L3
5.	Discuss the event-driven model in NodeJS with example.	L2
6.	Explain ExpressJS and how it is used in NodeJS applications.	L2
7.	What are the core concepts of NodeJS?	L1
8.	How can NodeJS be used to interact with a database?	L3
9.	Describe NodeJS command-line utilities with examples.	L2
10.	Compare synchronous and asynchronous programming in NodeJS.	L4

FULL STACK WEB DEVELOPMENT

UNIT –V

MONGO DB

Data is a precious thing and will last longer than the systems themselves.

— Tim Berners-Lee

UNIT- V: MONGODB

1. Unit Overview

This unit introduces **MongoDB**, a popular NoSQL database, covering concepts of SQL vs. NoSQL, database creation and management, data migration, integration with ReactJS and NodeJS, and the services offered by MongoDB. Students learn how to use MongoDB for building scalable, flexible, and high-performance web applications.

2. Objectives of the Unit

By the end of this unit, students should be able to:

- Understand the **difference between SQL and NoSQL databases**.
- Create and manage databases and collections in **MongoDB**.
- Perform **data migration** from relational databases to MongoDB.
- Integrate MongoDB with **ReactJS** and **NodeJS** applications.
- Utilize **MongoDB services** such as Atlas for cloud database management.

3. Learning Outcomes

After completing this unit, students will be able to:

- Design **NoSQL database schemas** for web applications.
- Perform **CRUD operations** using MongoDB.
- Migrate data from **SQL databases to MongoDB**.
- Connect and interact with MongoDB using **Node.js and React.js**.
- Leverage **MongoDB services** for cloud deployment, backups, and analytics.

4. Importance of Studying this Unit

- MongoDB is widely used in **modern web and mobile applications**, especially for scalable and flexible data storage.
- Understanding NoSQL and MongoDB is essential for **full-stack development**, particularly in the MERN stack (MongoDB, Express, React, Node.js).
- Knowledge of MongoDB helps students develop applications that can handle **large-scale, unstructured data efficiently**.
- Cloud services like **MongoDB Atlas** make deployment and maintenance of databases easier and industry-relevant.

5. Key Concepts

- **SQL vs NoSQL:** Differences, advantages, and use cases.
- **Creating and Managing MongoDB:** Databases, collections, documents, indexes.
- **Data Migration:** Moving data from relational databases to MongoDB.
- **MongoDB with ReactJS:** Using MongoDB as a backend database for React applications.
- **MongoDB with NodeJS:** Connecting Node.js applications to MongoDB, performing CRUD operations.
- **Services Offered by MongoDB:** MongoDB Atlas, cloud hosting, backup, monitoring, and analytics tools.
- **Querying and Aggregation:** Using MongoDB query language and aggregation pipelines for data analysis.

UNIT- V: MONGO DB

MongoDB: SQL and NoSQL Concepts Create and Manage MongoDB

1. SQL vs NoSQL (MongoDB) Concepts

SQL (Relational DB)	MongoDB (NoSQL, Document-based)
Database	Database
Table	Collection
Row	Document
Column	Field
Primary Key	_id field (unique by default)
Joins	Embedded documents or manual references
Schemas (Strict)	Schemaless (Flexible documents)
SQL Query (SELECT)	MongoDB Query (find)

2. Creating and Managing MongoDB

A. Installation & Setup

- **Install MongoDB:**
  MongoDB Installation Docs
- **Start MongoDB Server:**

  mongod

- **MongoDB Shell (CLI):**

  mongosh

B. Basic Operations

1. Create a Database

```
use myDatabase;
```

2. Create a Collection

```
db.createCollection("users");
```

3. Insert Documents

```
db.users.insertOne({
  name: "John Doe",
  email: "john@example.com",
  age: 25
});
```

4. Insert Multiple

```
db.users.insertMany([
  { name: "Alice", email: "alice@example.com" },
  { name: "Bob", email: "bob@example.com" }
]);
```

5. Read (Query) Data

```
db.users.find();                    // Get all
db.users.find({ name: "Alice" });       // Find by field
```

6. Update Document

```
db.users.updateOne(
  { name: "John Doe" },
  { $set: { age: 30 } }
);
```

7. Delete Document

```
db.users.deleteOne({ name: "Bob" });
```

C. Indexing

```
db.users.createIndex({ email: 1 }); // Ascending index on email
```

D. Relationships (Embedding vs Referencing)

1. Embedding (Like JOIN-less relationship)

```
db.orders.insertOne({
  userId: "123",
  items: [
    { productId: "p1", quantity: 2 },
    { productId: "p2", quantity: 1 }
  ]
});
```

2. Referencing

```
// In users collection
{ _id: ObjectId("userId123"), name: "User A" }

// In posts collection
{ userId: ObjectId("userId123"), title: "Post Title" }
```

Tools for Managing MongoDB

Tool	Purpose
MongoDB Compass	GUI for managing databases
Robo 3T	Another popular MongoDB GUI
mongosh	Shell for interacting with MongoDB
Mongoose (Node.js)	ODM for defining schemas in code

UNIT- V: MONGO DB

Migration of Data into MongoDB

Migrating data into **MongoDB** means transferring your existing data (often from relational databases like MySQL, PostgreSQL, or CSV/Excel files) into MongoDB's flexible document-based structure. Here's a step-by-step guide based on the **type of source** you're migrating from:

1. Migration from SQL (MySQL/PostgreSQL) to MongoDB

A. Using MongoDB's Official Tool: mongoimport

1. **Export SQL Data to CSV or JSON**
   Example using MySQL:

   ```
   SELECT * FROM users INTO OUTFILE '/tmp/users.csv'
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   LINES TERMINATED BY '\n';
   ```

2. **Import CSV to MongoDB**

   ```
   mongoimport --db mydb --collection users --type csv --file /tmp/users.csv --headerline
   ```

3. **Import JSON to MongoDB**
   If your SQL data is exported as JSON:

   ```
   mongoimport --db mydb --collection users --file users.json --jsonArray
   ```

B. Using a Node.js Script (ETL Process)

1. **Install Required Packages**

   ```
   npm install mysql2 mongoose
   ```

2. **Sample Migration Script (MySQL → MongoDB)**

   ```
   const mysql = require('mysql2/promise');
   const mongoose = require('mongoose');
   ```

```
async function migrate() {
  await mongoose.connect('mongodb://localhost:27017/mydb');
  const User = mongoose.model('User', new mongoose.Schema({}, { strict: false }));

  const connection = await mysql.createConnection({ host: 'localhost', user: 'root',
password: '', database: 'sql_db' });
  const [rows] = await connection.execute('SELECT * FROM users');

  for (const row of rows) {
    await User.create(row); // Dynamic schema
  }

  console.log('Migration complete');
  await connection.end();
  await mongoose.disconnect();
}

migrate();
```

2. Migration from CSV/Excel Files

A. Using mongoimport (CSV)

```
mongoimport --db mydb --collection products --type csv --file products.csv --headerline
```

B. Using Node.js + csv-parser

```
npm install csv-parser mongoose
const fs = require('fs');
const csv = require('csv-parser');
const mongoose = require('mongoose');
```

```
mongoose.connect('mongodb://localhost:27017/mydb');
const Product = mongoose.model('Product', new mongoose.Schema({}, { strict: false }));

fs.createReadStream('products.csv')
  .pipe(csv())
  .on('data', async (row) => {
    await Product.create(row);
  })
  .on('end', () => {
    console.log('CSV file successfully processed');
    mongoose.disconnect();
  });
```

MongoDB with ReactJS

Integrating **MongoDB with ReactJS** involves building a **full-stack application** using:

- **Frontend:** ReactJS (handles UI)
- **Backend:** Node.js with Express (handles APIs)
- **Database:** MongoDB (stores data)

Since **ReactJS** can't connect directly to MongoDB (as it would expose credentials), we use a **Node.js backend** to act as a middle layer.

Full-Stack Architecture Overview

```
 [ ReactJS Frontend ]
        t (HTTP)
[ Express.js API Backend ]
        t (Mongoose ODM)
[ MongoDB Database ]
```

UNIT- V: MONGO DB

✅**Step-by-Step Guide: MongoDB + ReactJS**

1. Setup Backend (Node.js + Express + MongoDB)

A. Create a Backend Folder

```
mkdir backend && cd backend
npm init -y
npm install express mongoose cors dotenv
```

B. Create File Structure

```
backend/
│
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── .env
├── server.js
```

C. .env

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/mydb
```

D. models/User.js

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
});
module.exports = mongoose.model('User', userSchema);
```

E. routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
const User = require('../models/User');

router.get('/', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

router.post('/', async (req, res) => {
  const newUser = new User(req.body);
  await newUser.save();
  res.status(201).json(newUser);
});

module.exports = router;
```

F. server.js

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const dotenv = require('dotenv');

dotenv.config();
const app = express();
app.use(cors());
app.use(express.json());

mongoose.connect(process.env.MONGO_URI)
```

```
  .then(() => console.log('MongoDB Connected'))
  .catch((err) => console.error('MongoDB Error:', err));

app.use('/api/users', require('./routes/userRoutes'));

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

2. Setup Frontend (ReactJS)

A. Create React App

```
npx create-react-app frontend
cd frontend
npm install axios
```

B. Sample React Component (App.js)

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

function App() {
  const [users, setUsers] = useState([]);
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const fetchUsers = async () => {
    const res = await axios.get('http://localhost:5000/api/users');
    setUsers(res.data);
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
```

```
    await axios.post('http://localhost:5000/api/users', { name, email });
    setName('');
    setEmail('');
    fetchUsers();
  };

  useEffect(() => {
    fetchUsers();
  }, []);

  return (
    <div>
      <h1>User Manager</h1>
      <form onSubmit={handleSubmit}>
        <input value={name} onChange={(e) => setName(e.target.value)} placeholder="Name" />
        <input value={email} onChange={(e) => setEmail(e.target.value)} placeholder="Email" />
        <button type="submit">Add User</button>
      </form>

      <ul>
        {users.map(user => (
          <li key={user._id}>{user.name} ({user.email})</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

UNIT- V: MONGO DB

✅**Final Notes**

- Run backend:

  cd backend
  node server.js

- Run frontend:

  cd frontend
  npm start

MongoDB with NodeJS

Integrating **MongoDB with Node.js** is a common and powerful setup for building backend applications or APIs. Below is a **complete guide** to help you set up and work with **MongoDB in a Node.js environment** using **Mongoose**, the most popular ODM (Object Data Modeling) library.

✅**1. Prerequisites**

- Node.js installed
- MongoDB running locally or using **MongoDB Atlas**
- Basic knowledge of JavaScript

2. Project Setup

A. Initialize the Project

mkdir mongodb-node-app
cd mongodb-node-app
npm init -y
npm install express mongoose dotenv

3. Folder Structure

```
mongodb-node-app/
│
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── .env
├── server.js
```

4. .env File

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/mydb
```

Replace MONGO_URI with your **MongoDB Atlas URI** if you're using the cloud.

5. Create a Mongoose Model (models/User.js)

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  createdAt: { type: Date, default: Date.now },
});

module.exports = mongoose.model('User', userSchema);
```

6. Create Routes (routes/userRoutes.js)

```
const express = require('express');
const router = express.Router();
```

```
const User = require('../models/User');

// Get all users
router.get('/', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

// Add a new user
router.post('/', async (req, res) => {
  try {
    const newUser = await User.create(req.body);
    res.status(201).json(newUser);
  } catch (err) {
    res.status(400).json({ message: err.message });
  }
});

module.exports = router;
```

7. Setup Express Server (server.js)

```
const express = require('express');
const mongoose = require('mongoose');
const dotenv = require('dotenv');
const userRoutes = require('./routes/userRoutes');

dotenv.config();
const app = express();
app.use(express.json());

mongoose.connect(process.env.MONGO_URI)
```

```
  .then(() => console.log('✅MongoDB connected'))
  .catch((err) => console.error('❌MongoDB connection error:', err));

app.use('/api/users', userRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`🚀 Server running on http://localhost:${PORT}`));
```

8. Test API with Postman or Curl

➕Add User

```
POST http://localhost:5000/api/users
Content-Type: application/json

{
  "name": "Alice",
  "email": "alice@example.com"
}
```

Get All Users

```
GET http://localhost:5000/api/users
```

Services Offered by MongoDB

MongoDB offers a variety of **services and tools** designed to make modern application development faster, more scalable, and secure. Below is a categorized list of the **key services offered by MongoDB**:

1. MongoDB Atlas (Fully Managed Cloud Database)

MongoDB Atlas is the **flagship service** — a fully-managed cloud database service.

Key Features:

- Automated deployment on **AWS, Azure, GCP**
- Auto-scaling, backups, monitoring
- Built-in **security** and **compliance** (SOC 2, HIPAA, etc.)
- Global clusters for low-latency apps
- Performance optimization tools

2. Database Services

Service	Description
MongoDB Community Edition	Free, open-source version for local/self-hosted deployments
MongoDB Enterprise Edition	Advanced security, in-memory storage, and commercial support
MongoDB Atlas	Cloud-native managed database with built-in automation & security
Atlas for Government	FedRAMP-authorized Atlas version for U.S. government use

3. Developer Services

Tool/Service	Description
MongoDB Compass	GUI tool to explore, visualize, and query MongoDB data
Atlas CLI	Command-line interface to manage Atlas resources
MongoDB Shell	Powerful modern shell (mongosh) to interact with MongoDB
MongoDB Charts	Data visualization tool natively integrated with MongoDB Atlas
MongoDB VS Code Plugin	Browse collections, run queries, and manage databases from VS Code

4. Data Services

Feature/Service	Purpose
Atlas Search	Full-text search using Lucene, built into Atlas
Atlas Data Federation	Query across multiple data sources (e.g., S3, other clusters)
Atlas Data Lake	Run analytical queries on archived data in cloud storage
Change Streams	Real-time data change notifications for event-driven architectures
Triggers	Serverless functions triggered by database changes

5. Application Services (Backend as a Service)

Feature	Description
Atlas App Services	Serverless backend with authentication, triggers, and GraphQL APIs
Authentication Providers	Built-in support for OAuth, Email/Password, JWT, and Custom auth
Functions	Run serverless functions inside MongoDB Atlas
GraphQL API	Auto-generated GraphQL API layer over your MongoDB collections
Data Sync	Sync data between client apps (mobile/web) and Atlas backend in real time

6. Mobile Services

Service	Description
Realm Database	Local embedded database for iOS/Android with sync capabilities
Realm Sync	Real-time sync between MongoDB Atlas and mobile apps
Device Sync	Bi-directional syncing and offline-first capabilities for mobile apps

7. Security & Compliance Services

Feature	Description
Encryption at Rest & TLS	Built-in encryption for stored data and data in transit
Role-Based Access Control (RBAC)	Granular access control for users and applications
Auditing	Track access and operations for compliance
Backup & Restore	Continuous backups and point-in-time recovery
VPC Peering & PrivateLink	Secure private networking in cloud deployments

8. Monitoring & Analytics

Tool	Purpose
Atlas Monitoring	Real-time dashboards for performance and metrics
Performance Advisor	Query optimization suggestions
Slow Query Analyzer	Analyze inefficient queries

9. Migration Services

Tool/Service	Description
MongoMirror	Migrate from on-prem MongoDB to Atlas
Live Migration Service	Move data from another cloud provider to Atlas with minimal downtime
MongoDB Compass	Import/export data through CSV/JSON and run migration queries

10. Support & Training

Service	Details
MongoDB University	Free online training courses on MongoDB (Beginner to Advanced)
Enterprise Support	Dedicated technical support for enterprise customers
Documentation & SDKs	Extensive docs + SDKs for Node.js, Python, Java, C#, Go, Rust, etc.

UNIT- V: MONGO DB

CASE STUDY 1: Student Database System (MongoDB CRUD)

Scenario:

A college wants to store student details:

- Name
- Age
- Course

Questions:

1. Create **MongoDB database & collection**
2. Perform:
    - Insert student
    - Find students
    - Update student
    - Delete student
3. Write MongoDB queries

CASE STUDY 2: SQL to NoSQL Migration

Scenario:

A company wants to migrate from SQL to MongoDB

Questions:

1. Compare **SQL vs NoSQL**
2. Convert table structure into **JSON document**
3. Explain advantages of MongoDB
4. Show sample document

CASE STUDY 3: Node.js + MongoDB Integration

Scenario:

Build backend to store user data

Questions:

1. Connect MongoDB with Node.js
2. Use **MongoDB driver / Mongoose**
3. Perform CRUD operations
4. Handle database errors

UNIT- V: MONGO DB

CASE STUDY 4: React + MongoDB Application

Scenario:

Frontend (React) sends data to backend and stores in MongoDB

Questions:

1. Explain **data flow (React → Node → MongoDB)**
2. Send data using API
3. Store data in database
4. Fetch and display in React

CASE STUDY 5: E-Commerce Product Database

Scenario:

Store product details:

- Name
- Price
- Category

Questions:

1. Design **MongoDB collection schema**
2. Insert multiple products
3. Query products by category
4. Update price

CASE STUDY 6: User Authentication Database

Scenario:

Store login credentials

Questions:

1. Create collection for users
2. Insert user data
3. Validate login using query
4. Explain security (basic concept)

CASE STUDY 7: MongoDB Services

Scenario:

Use MongoDB cloud services

Questions:

UNIT- V: MONGO DB

1. Explain **MongoDB Atlas**
2. Benefits of cloud database
3. Backup and scaling features

CASE STUDY 8: Data Import into MongoDB

Scenario:

Import data from JSON/CSV

Questions:

1. Use **mongoimport tool**
2. Import file into collection
3. Verify data

CASE STUDY 9: Real-Time Application Database

Scenario:

Chat application storing messages

Questions:

1. Store messages as documents
2. Use timestamp (date)
3. Retrieve recent messages

CASE STUDY 10: Aggregation Example

Scenario:

Calculate average marks of students

Questions:

1. Use **aggregation pipeline**
2. Apply $group
3. Display result

UNIT- V: MONGO DB		
MongoDB: SQL and NoSQL Concepts Create and Manage MongoDB, Migration of Data into MongoDB, MongoDB with ReactJS, MongoDB with NodeJS, Services Offered by MongoDB.		
PART-A		
1.	What is MongoDB?	L1
2.	Define NoSQL.	L1
3.	What is the difference between SQL and NoSQL?	L2
4.	List the data types supported by MongoDB.	L1
5.	What is a document in MongoDB?	L1
6.	What is a collection in MongoDB?	L1
7.	Write the command to create a database in MongoDB.	L3
8.	What is the role of MongoDB in MERN stack?	L2
9.	List two services offered by MongoDB.	L1
10.	Define data migration in context of MongoDB.	L1
PART-B		
1.	Compare SQL and NoSQL databases.	L4
2.	Explain how to create and manage a database in MongoDB.	L3
3.	Describe the process of data migration to MongoDB.	L2
4.	Explain the use of MongoDB in NodeJS application with example.	L2
5.	Write a program to insert and retrieve documents in MongoDB.	L3
6.	Discuss services offered by MongoDB in detail.	L2
7.	How is MongoDB used in ReactJS applications?	L3
8.	Describe the structure of a MongoDB document.	L2
9.	Explain CRUD operations in MongoDB with examples.	L3
10.	Discuss advantages and limitations of using MongoDB.	L4