**Lecture 1:**

**Activities:**

* **Icebreaker Activity (10 mins):**
  + Ask students to share their earliest memory of using the internet and how it has changed over time.
* **Interactive Presentation (15 mins):**
  + **Content:** History of the Internet, differences between the Internet and the Web, basic terminology (e.g., URLs, browsers, servers).
  + **Tools:** Use a timeline tool like Prezi to visually demonstrate the evolution of the Internet.
* **Discussion (10 mins):**
  + Discuss how the Web has transformed various industries (e.g., education, entertainment, business).
  + Encourage students to think about how their daily lives are impacted by the web.

**2. Understanding the HTTP Protocol**

**Objective:** Learn the basics of the HTTP protocol, its methods, and how it enables web communication.

**Activities:**

* **Interactive Presentation (10 mins):**
  + **Content:** Introduction to HTTP, how it works, basic methods (GET, POST, PUT, DELETE), and status codes.
  + **Tools:** Use diagrams and flowcharts to illustrate HTTP request and response cycles.
* **Live Demonstration (15 mins):**
  + Use tools like Postman or a simple HTML form to demonstrate how HTTP requests and responses work.
* **Hands-On Activity (10 mins):**
  + Have students perform a simple HTTP GET request using their browsers' developer tools and analyze the response headers.

**3. Client-Server Architecture**

**Objective:** Understand the client-server model and its role in web technologies.

**Activities:**

* **Interactive Presentation (10 mins):**
  + **Content:** Definition of client-server architecture, examples in real-world applications, advantages and disadvantages.
  + **Tools:** Use animation tools to show the interaction between client and server in a visual and dynamic way.
* **Group Activity (15 mins):**
  + Divide students into small groups. Each group will brainstorm and present a real-world application that uses the client-server architecture.
* **Q&A Session (10 mins):**
  + Open the floor for students to ask questions about anything they've learned in the lecture.

**Additional Tips:**

* **Use Visual Aids:** Diagrams, flowcharts, and animations can make complex concepts easier to understand.
* **Incorporate Real-World Examples:** Relate theoretical concepts to everyday applications to make them more relatable.
* **Interactive Tools:** Use tools like Kahoot for quizzes and Poll Everywhere for instant feedback.
* **Hands-On Practice:** Encourage students to try out HTTP requests, explore web development tools, and engage in practical exercises.

**Internet**

1. **Definition**: The Internet is a global network of interconnected computers and other devices. It is a massive network of networks that enables various forms of communication, data exchange, and access to information.
2. **Components**: The Internet includes hardware (such as routers, servers, and cables) and protocols that govern how data is transmitted and received.
3. **Services**: The Internet supports various services and applications, including email, file transfer (FTP), Voice over IP (VoIP), and the World Wide Web.
4. **Scope**: The Internet is the underlying infrastructure that makes different types of digital communication possible.

**World Wide Web (WWW)**

1. **Definition**: The World Wide Web is a collection of information that is accessed via the Internet. It consists of web pages and websites, which are interconnected through hyperlinks.
2. **Components**: The WWW uses Hypertext Transfer Protocol (HTTP) to transmit data, Hypertext Markup Language (HTML) to structure web pages, and browsers (like Chrome, Firefox, and Safari) to display this information to users.
3. **Functionality**: The Web is a service that runs on the Internet, allowing users to navigate through websites, access information, and interact with various forms of media.
4. **Scope**: The WWW is just one of the many services that operate over the Internet. It is a subset of the broader Internet.

**Key Differences**

* **Infrastructure vs. Service**: The Internet is the infrastructure, while the Web is a service that operates on that infrastructure.
* **Protocols**: The Internet uses a variety of protocols (TCP/IP, FTP, SMTP, etc.), whereas the Web primarily uses HTTP/HTTPS.
* **Usage**: The Internet enables numerous services (including the Web, email, file sharing), while the Web specifically refers to accessing and navigating websites.

**Analogy**

* **Internet**: Think of the Internet as a global transportation system that includes roads, highways, bridges, and vehicles.
* **WWW**: The World Wide Web would be like a specific type of vehicle that uses this transportation system to move people and goods around.

[Web and HTTP protocol tutorial | Hypertext Transfer Protocol explained | World Wide Web | TechTerms (youtube.com)](https://www.youtube.com/watch?v=qcALGDn0zpk)

**Lecture Plan: Concepts of Effective Web Design & Web Design Issues**

**1. Introduction (5 minutes)**

* **Objective:** Introduce the importance of effective web design.
* **Discussion Points:**
  + Why good web design matters.
  + The impact of design on user experience and business outcomes.

**2. Concepts of Effective Web Design (25 minutes)**

* **Objective:** Teach the key principles of effective web design.
* **Interactive Activity:** Show examples of well-designed and poorly designed websites. Discuss what makes the good examples effective.
* **Key Concepts:**

1. **User-Centered Design:**
   * Focus on the needs and preferences of the target audience.
   * Conduct user research and usability testing.
2. **Simplicity:**
   * Keep the design simple and uncluttered.
   * Avoid unnecessary elements that can distract users.
3. **Consistency:**
   * Maintain a consistent look and feel throughout the website.
   * Use a consistent color scheme, typography, and layout.
4. **Visual Hierarchy:**
   * Arrange elements in a way that guides users’ attention to important information first.
   * Use size, color, and positioning to create a clear hierarchy.
5. **Navigation:**
   * Make navigation intuitive and easy to use.
   * Ensure the navigation menu is accessible from every page.
6. **Responsive Design:**
   * Ensure the website looks and functions well on all devices, including desktops, tablets, and mobile phones.
   * Use responsive design techniques like flexible grids and media queries.
7. **Loading Speed:**
   * Optimize the website for fast loading times.
   * Compress images, minify CSS and JavaScript, and use caching.
8. **Accessibility:**
   * Design the website to be accessible to all users, including those with disabilities.
   * Follow web accessibility guidelines like WCAG (Web Content Accessibility Guidelines).
9. **Content:**
   * Provide high-quality, relevant content that meets users' needs.
   * Use clear and concise language, and organize content logically.
10. **Aesthetics:**
    * Ensure the website is visually appealing.
    * Use a cohesive color palette, balanced layout, and engaging imagery.

**3. Web Design Issues (25 minutes)**

* **Objective:** Highlight common web design issues and how to avoid them.
* **Interactive Activity:** Group discussion or brainstorming session to identify potential issues in example websites.
* **Key Issues:**

1. **Poor Navigation:**
   * Complicated or hidden navigation menus.
   * Solution: Simplify the navigation and ensure it is always visible.
2. **Inconsistent Design:**
   * Different styles across pages.
   * Solution: Create a style guide and adhere to it.
3. **Slow Loading Times:**
   * Large images, unoptimized code.
   * Solution: Optimize all media, use efficient coding practices.
4. **Not Mobile-Friendly:**
   * Poor display on mobile devices.
   * Solution: Implement responsive design from the start.
5. **Cluttered Layout:**
   * Too much content or too many elements on a page.
   * Solution: Simplify the layout and prioritize important content.
6. **Lack of Accessibility:**
   * Difficult for users with disabilities to navigate or use.
   * Solution: Follow accessibility guidelines and test with assistive technologies.
7. **Poor Readability:**
   * Hard-to-read fonts or poor contrast.
   * Solution: Use readable fonts and ensure sufficient contrast between text and background.
8. **Broken Links and Errors:**
   * Links that lead to 404 pages or malfunctioning scripts.
   * Solution: Regularly check and update links and scripts.
9. **Outdated Design:**
   * An old-fashioned look that can turn users away.
   * Solution: Keep up with modern design trends and update the design periodically.
10. **Lack of SEO Optimization:**
    * Not appearing in search engine results.
    * Solution: Use SEO best practices, such as proper use of meta tags, keywords, and content optimization.

**4. Real-World Examples and Case Studies (10 minutes)**

* **Objective:** Show examples of websites that follow effective design principles and those that suffer from common issues.
* **Activity:** Analyze these examples as a class and discuss what can be learned from them.

**5. Q&A and Recap (10 minutes)**

* **Objective:** Address any questions and summarize key takeaways.
* **Activity:** Open floor for questions and engage in a brief discussion.
* **Recap:**
  + Principles of effective web design.
  + Common web design issues and solutions.

**6. Homework Assignment**

* **Objective:** Reinforce the day's lessons through practice.
* **Assignment:** Review a website of their choice and write a short analysis identifying good design practices and areas for improvement.

**Lecture Plan: HTTP Request and Response**

**1. Introduction (5 minutes)**

* **Objective:** Provide an overview of the HTTP protocol and its importance in web communication.
* **Discussion Points:**
  + What is HTTP (HyperText Transfer Protocol)?
  + Role of HTTP in the web.

**2. HTTP Request (15 minutes)**

* **Objective:** Explain the components and structure of an HTTP request.
* **Interactive Activity:** Show a sample HTTP request and break down its components.
* **Key Concepts:**

1. **Request Line:**
   * **Method:** The action to be performed (e.g., GET, POST, PUT, DELETE).
   * **URL:** The resource being requested.
   * **HTTP Version:** The version of the HTTP protocol (e.g., HTTP/1.1).

http

Copy code

GET /index.html HTTP/1.1

1. **Headers:**
   * Provide additional information about the request.
   * Common headers: Host, User-Agent, Accept, Content-Type.

http

Copy code

Host: www.example.com

User-Agent: Mozilla/5.0

Accept: text/html

1. **Body:**
   * Optional part of the request, mainly\

**Chapter – 2**

### Examples of Good Web Design

1. **Apple**
   * [Apple](https://www.apple.com/)
   * Clean, intuitive, and visually appealing. Apple’s website has a strong focus on visuals and simplicity, with a well-organized layout.
2. **Dropbox**
   * [Dropbox](https://www.dropbox.com/)
   * Simple and clear design with a strong focus on the user’s journey. Easy navigation and a minimalistic approach make it user-friendly.
3. **Airbnb**
   * [Airbnb](https://www.airbnb.com/)
   * Visually appealing with high-quality images and clear calls-to-action. The site is user-friendly and responsive, with an intuitive search function.
4. **Slack**
   * [Slack](https://slack.com/)
   * Modern design with a focus on functionality and ease of use. Clear messaging and a user-friendly interface.

### Examples of Bad Web Design

1. **Yale School of Art**
   * Yale School of Art
   * Known for its overly complex and confusing layout. The design is cluttered and difficult to navigate, which can be frustrating for users.
2. **Arngren**
   * [Arngren](http://arngren.net/)
   * The website is extremely cluttered with poor layout and design. It lacks visual hierarchy and is overwhelming for users.
3. **Ling's Cars**
   * [Ling's Cars](https://www.lingscars.com/)
   * The website is chaotic with excessive animations and a confusing layout. It’s difficult to find information due to the cluttered design.
4. **The World’s Worst Website Ever**
   * [The World's Worst Website Ever](http://www.theworldsworstwebsiteever.com/)
   * Created as a humorous example of bad web design, this site includes many common design mistakes like poor color contrast, bad typography, and clutter.

### Learning from Examples

**Good Web Design Practices:**

* **Apple:** Clear, high-quality visuals, intuitive navigation, and a clean layout.
* **Dropbox:** Simple, focused content with easy navigation and a minimalistic design.
* **Airbnb:** User-centric design with high-quality images, clear CTAs, and a responsive layout.
* **Slack:** Modern, functional design with a focus on ease of use and clear messaging.

**Bad Web Design Practices:**

* **Yale School of Art:** Overly complex layout, lack of clear navigation, and cluttered design.
* **Arngren:** Cluttered layout, lack of visual hierarchy, and overwhelming amount of content.
* **Ling's Cars:** Excessive animations, confusing layout, and cluttered design.
* **The World’s Worst Website Ever:** Poor color contrast, bad typography, and cluttered content.

**JS**

| **Feature** | **var** | **let** | **const** |
| --- | --- | --- | --- |
| **Scope** | Function-scoped or Global-scoped | Block-scoped | Block-scoped |
| **Hoisting** | Hoisted with undefined | Hoisted, but in temporal dead zone | Hoisted, but in temporal dead zone |
| **Reassignment** | Allowed | Allowed | Not allowed |
| **Redeclaration** | Allowed | Not allowed | Not allowed |
| **Initialization** | Optional (can be initialized later) | Required before use | Required at declaration |

**--------------------------------------------------------**

**1. Comparison: querySelector vs. getElementById**

| **Feature** | **querySelector** | **getElementById** |
| --- | --- | --- |
| **Purpose** | Selects elements using any valid CSS selector. | Selects an element by its id attribute. |
| **Syntax** | document.querySelector('selector') | document.getElementById('elementId') |
| **Return Value** | The first matching element or null. | The element with the specified id or null. |
| **Selector Type** | Any CSS selector (id, class, tag, etc.). | Only id. |
| **Flexibility** | High—supports complex selectors. | Limited to id selection. |
| **Performance** | Slightly slower (parses selectors). | Faster for id selection. |
| **Use Case** | For complex or multiple conditions. | Quick selection when id is known. |

**2. Comparison: innerText vs. textContent vs. innerHTML vs. value**

| **Feature** | **innerText** | **textContent** | **innerHTML** | **value** |
| --- | --- | --- | --- | --- |
| **Purpose** | Gets/sets text, considers CSS styles. | Gets/sets text, including hidden text. | Gets/sets HTML content, including tags. | Gets/sets value of form elements. |
| **Includes Hidden Text** | No | Yes | N/A | N/A |
| **HTML Tags** | Ignores HTML tags. | Ignores HTML tags. | Includes HTML tags. | N/A |
| **Performance** | Slower (considers CSS/layout). | Faster (ignores CSS/layout). | Slower (parses HTML). | Fast (direct form element access). |
| **Use Case** | For visible text, respecting styles. | For all text, regardless of visibility. | For manipulating/retrieving HTML. | For interacting with form input values. |
| **Typical Elements** | Any visible elements (div, p, span). | Any elements (including hidden ones). | Any elements (HTML content). | Form elements (input, textarea). |

### ****Key JavaScript Built-in Objects:****

1. **Object**
   * The base object from which all other objects inherit.
   * Provides basic methods like Object.keys(), Object.values(), Object.entries(), and more.

**Example:**

javascript

Copy code

const obj = { name: "Alice", age: 25 };

console.log(Object.keys(obj)); // Output: ["name", "age"]

1. **Array**
   * Represents a collection of elements that can be accessed by their index.
   * Methods like push(), pop(), map(), filter(), reduce(), etc., are commonly used.

**Example:**

javascript

Copy code

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

fruits.push("Orange"); // Adds "Orange" to the array

console.log(fruits); // Output: ["Apple", "Banana", "Cherry", "Orange"]

1. **String**
   * Represents a sequence of characters.
   * Offers methods for string manipulation such as charAt(), slice(), substring(), toUpperCase(), etc.

**Example:**

javascript

Copy code

const greeting = "Hello, World!";

console.log(greeting.toUpperCase()); // Output: "HELLO, WORLD!"

1. **Number**
   * Represents numeric values, including integers and floating-point numbers.
   * Methods include toFixed(), toString(), isInteger(), etc.

**Example:**

javascript

Copy code

const num = 123.456;

console.log(num.toFixed(2)); // Output: "123.46"

1. **Boolean**
   * Represents a logical entity and can have two values: true or false.
   * Often used in conditional testing.

**Example:**

javascript

Copy code

const isTrue = Boolean(1); // Converts 1 to true

console.log(isTrue); // Output: true

1. **Date**
   * Represents a date and time.
   * Provides methods to manipulate dates and times such as getDate(), getMonth(), getFullYear(), etc.

**Example:**

javascript

Copy code

const today = new Date();

console.log(today.getFullYear()); // Output: current year

1. **Math**
   * Provides properties and methods for mathematical constants and functions.
   * Methods include Math.random(), Math.floor(), Math.ceil(), Math.max(), etc.

**Example:**

javascript

Copy code

const randomNum = Math.random(); // Generates a random number between 0 and 1

console.log(randomNum);

1. **RegExp**
   * Represents a regular expression, which is used for pattern matching in strings.
   * Methods include test(), exec(), and match().

**Example:**

javascript

Copy code

const pattern = /hello/i;

console.log(pattern.test("Hello, World!")); // Output: true

1. **Function**
   * Functions in JavaScript are objects, and you can treat them as variables or pass them as arguments.
   * Methods like call(), apply(), and bind() are available.

**Example:**

javascript

Copy code

function greet() {

console.log("Hello, world!");

}

greet(); // Output: "Hello, world!"

1. **Promise**
   * Represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
   * Common methods include then(), catch(), and finally().

**Example:**

javascript

Copy code

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Done!"), 1000);

});

myPromise.then(result => console.log(result)); // Output after 1 second: "Done!"

1. **Map and Set**
   * Map is a collection of key-value pairs where keys can be of any data type.
   * Set is a collection of unique values.

**Example for Map:**

javascript

Copy code

const map = new Map();

map.set('name', 'John');

map.set('age', 30);

console.log(map.get('name')); // Output: "John"

**Example for Set:**

javascript

Copy code

const set = new Set([1, 2, 3, 4, 4]);

console.log(set.size); // Output: 4 (duplicates are not allowed)

### ****Why JavaScript Built-in Objects Are Important:****

* **Efficiency:** These objects provide built-in methods that are optimized for performance and save you from writing common operations from scratch.
* **Utility:** They cover a broad range of functionalities from basic operations to complex mathematical calculations, string manipulations, and date/time handling.
* **Standardization:** Since they are part of the ECMAScript standard, they ensure that your code behaves consistently across different environments (browsers, Node.js, etc.).

**RegExp**

Regular expressions (RegExp) are like search patterns. They help you find specific patterns of characters in text. You can use them to check if an input matches a certain format.

Think of RegExp like this:

* **Text Filter**: It's like a filter that looks for specific text patterns in a bigger chunk of text.
* **Pattern Match**: If the input matches the pattern, it’s valid. Otherwise, it’s not.

**2. Structure of a Regular Expression**

A regular expression has two parts:

* **Pattern**: What you’re looking for (e.g., a sequence of digits, a letter, or a word).
* **Flags**: Optional settings that modify how the pattern works (like case-sensitivity or global search).

Let’s take a simple example:

let pattern = /abc/;

This pattern checks if the string "abc" exists in the input.

**3. Basic RegExp Components**

Here’s a breakdown of common RegExp elements to help you build simple patterns:

| **Character** | **Meaning** |
| --- | --- |
| ^ | Matches the **beginning** of the input. |
| $ | Matches the **end** of the input. |
| . | Matches **any character** (except newline). |
| \d | Matches any **digit** (0–9). |
| \w | Matches any **word character** (letters, digits, underscore). |
| \s | Matches any **whitespace** (spaces, tabs). |
| + | Matches **one or more** occurrences of the preceding element. |
| \* | Matches **zero or more** occurrences of the preceding element. |
| {n,m} | Matches **at least n and at most m** occurrences. |
| [abc] | Matches any **one** of the characters inside the brackets (e.g., 'a', 'b', or 'c'). |
| [^abc] | Matches any character **except** what's inside the brackets. |

**4. Simple Examples**

Let’s explore some examples step by step to understand how RegExp works:

**a) Check if an input contains only digits**

* You want to make sure the user entered only numbers (e.g., for a phone number).
* Pattern: /^\d+$/

Explanation:

* ^: Start of input.
* \d+: One or more digits (\d stands for digit, + means "one or more").
* $: End of input.

So, this pattern makes sure the input is **all digits**, from start to end.

Example in JavaScript:

let pattern = /^\d+$/;

console.log(pattern.test("12345")); // true

console.log(pattern.test("12a45")); // false (contains a letter)

**b) Check if an email is in the correct format**

Emails follow a specific format like username@domain.com.

* Pattern: /^[^\s@]+@[^\s@]+\.[^\s@]+$/

Explanation:

* ^: Start of input.
* [^\s@]+: Match any character except a space (\s) or @. This is the username part.
* @: There must be an @ symbol.
* [^\s@]+: Match the domain name.
* \.: The . symbol in the domain part.
* [^\s@]+: Match the domain extension (e.g., .com, .org).
* $: End of input.

Example in JavaScript:

let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

console.log(emailPattern.test("user@example.com")); // true

console.log(emailPattern.test("user@ example.com")); // false (has space)

**c) Check if a password is strong**

A strong password typically needs:

* At least 8 characters.
* At least one uppercase letter.
* At least one lowercase letter.
* At least one digit.

Pattern: /^(?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z]).{8,}$/

Explanation:

* ^: Start of input.
* (?=.\*\d): Look for at least one digit (\d).
* (?=.\*[a-z]): Look for at least one lowercase letter.
* (?=.\*[A-Z]): Look for at least one uppercase letter.
* .{8,}: Make sure the password is at least 8 characters long.
* $: End of input.

Example in JavaScript:

javascript

Copy code

let passwordPattern = /^(?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z]).{8,}$/;

console.log(passwordPattern.test("Password123")); // true

console.log(passwordPattern.test("pass123")); // false (no uppercase, too short)

**5. Testing and Using Regular Expressions**

In JavaScript, you use the .test() method to check if the input matches the regular expression. Here’s an example:

javascript

Copy code

let pattern = /^\d+$/; // Only digits

let input = "12345";

if (pattern.test(input)) {

console.log("Valid input!");

} else {

console.log("Invalid input!");

}

## **Introduction to jQuery**

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With jQuery, you can write less code and achieve more functionality.

### ****Why Use jQuery?****

* **Simplifies JavaScript**: It allows you to perform complex tasks with minimal code.
* **Cross-Browser Compatibility**: jQuery handles browser inconsistencies, ensuring your code works across different browsers (Chrome, Firefox, Safari, IE, etc.).
* **Rich Library**: jQuery provides built-in functions for common JavaScript tasks.
* **Plugins**: There is a vast ecosystem of plugins that extend jQuery’s functionality (e.g., sliders, date pickers, etc.).

## **Basic Concepts of jQuery**

### 1. ****The jQuery Syntax****

* The basic jQuery syntax is: $(selector).action();
  + $ is a shorthand for jQuery.
  + selector is used to "find" or "select" HTML elements.
  + action is what you want to do with the selected elements.

**Example:**

$("p").hide(); // Hides all <p> elements

### 2. ****Document Ready****

The $(document).ready() function ensures that the HTML document is fully loaded and ready before jQuery runs. This prevents errors where jQuery tries to interact with elements that haven't yet been loaded.

**Syntax:**

$(document).ready(function(){

// jQuery code goes here

});

Or, in shorthand:

$(function(){

// jQuery code goes here

});

## **Selectors in jQuery**

jQuery uses CSS selectors to select and manipulate elements.

### Common Selectors:

1. **ID Selector**: Selects an element by its ID.

$("#myId").hide(); // Hides the element with id="myId"

1. **Class Selector**: Selects elements by class.

$(".myClass").hide(); // Hides all elements with class="myClass"

1. **Tag Selector**: Selects elements by tag name.

$("p").hide(); // Hides all <p> elements

1. **Attribute Selector**: Selects elements with a specific attribute.

$("input[type='text']").hide(); // Hides all <input> elements with type="text"

## **Events in jQuery**

Events are actions that occur when a user interacts with a webpage, such as clicking, hovering, or submitting a form. jQuery makes it easy to handle these events.

### Common Event Methods:

1. **.click()**: Triggers a function when an element is clicked.

$("#button").click(function(){

alert("Button clicked!");

});

1. **.hover()**: Binds two functions to be executed when the mouse enters and leaves an element.

$("#element").hover(function(){

// Mouse enters

}, function(){

// Mouse leaves

});

1. **.keydown() and .keyup()**: Detect when a key is pressed or released.

$("#inputField").keydown(function(){

console.log("Key pressed");

});

## **DOM Manipulation with jQuery**

DOM manipulation is one of the most powerful features of jQuery. You can add, remove, and modify HTML elements and their content using jQuery methods.

### 1. ****Changing Content:****

* **.html()**: Gets or sets the HTML content of an element.

$("#content").html("New HTML content");

* **.text()**: Gets or sets the text content of an element.

$("#textElement").text("New text content");

### 2. ****Changing CSS Styles:****

* **.css()**: Changes the CSS properties of an element.

$("#element").css("color", "blue");

### 3. ****Adding and Removing Elements:****

* **.append()**: Inserts content at the end of selected elements.

$("body").append("<p>Appended paragraph</p>");

* **.prepend()**: Inserts content at the beginning of selected elements.

$("#element").prepend("<p>Prepended content</p>");

* **.remove()**: Removes selected elements from the DOM.

$("#element").remove();

## **jQuery Effects**

jQuery comes with a number of built-in animation and effect methods to make the web page dynamic.

### Common Effects:

1. **.hide() and .show()**: Hides or shows elements.

$("#element").hide(); // Hides the element

$("#element").show(); // Shows the element

1. **.toggle()**: Toggles between hiding and showing an element.

$("#element").toggle();

1. **.fadeIn() and .fadeOut()**: Fades in or out an element.

$("#element").fadeOut(); // Fades out the element

1. **.slideUp() and .slideDown()**: Slides up or down an element.

$("#element").slideUp(); // Slides the element up

## **Traversing the DOM**

jQuery provides methods to navigate (or "traverse") the DOM tree.

### Traversing Methods:

1. **.parent()**: Selects the parent element.

$("#element").parent();

1. **.children()**: Selects all direct children.

$("#element").children();

1. **.siblings()**: Selects all sibling elements.

$("#element").siblings();

1. **.find()**: Finds descendants of an element.

$("#element").find("p");

## **jQuery Chaining**

jQuery allows you to perform multiple methods on the same set of elements, one after the other, in a single statement. This is called **chaining**.

**Example:**

$("#element").css("color", "red").slideUp(1000).slideDown(1000);

**DOM Traversal and Manipulation**

**Objective**: Learn how to traverse the DOM tree and manipulate elements dynamically.

**1. Introduction to DOM Traversal**

**Concept**:

* DOM traversal allows you to navigate through the DOM tree to access and manipulate elements.
* The DOM tree is a hierarchical representation of the HTML document where each element is a node.

**Key Properties and Methods**:

* **parentNode**: Access the parent node of an element.
* **childNodes**: Access all child nodes (including text nodes) of an element.
* **firstChild**: Access the first child node of an element.
* **lastChild**: Access the last child node of an element.
* **nextSibling**: Access the next sibling node of an element.
* **previousSibling**: Access the previous sibling node of an element.
* **children**: Access only element nodes (excluding text nodes) that are direct children of an element.
* **firstElementChild**: Access the first child element.
* **lastElementChild**: Access the last child element.

**Code Examples**:

// Access parent node

const parent = document.querySelector('.child').parentNode;

// Access all child nodes

const children = document.querySelector('.parent').childNodes;

// Access first child element

const firstChild = document.querySelector('.parent').firstElementChild;

// Access next sibling

const next = document.querySelector('.child').nextElementSibling;

// Access previous sibling

const previous = document.querySelector('.child').previousElementSibling;

**Activity**:

1. Create an HTML structure with nested elements.
2. Write JavaScript to:
   * Log the parent node of a specific element.
   * Log all child nodes of a parent element.
   * Add a new element before the first child.

**2. Introduction to DOM Manipulation**

**Concept**:

* DOM manipulation involves creating, adding, modifying, and removing elements within the DOM.

**Key Methods**:

* **createElement(tagName)**: Create a new HTML element.
* **appendChild(node)**: Append a node as the last child of an element.
* **insertBefore(newNode, referenceNode)**: Insert a new node before a specified reference node.
* **removeChild(node)**: Remove a specified child node from an element.

**Code Examples**:

**// Create a new element**

const newItem = document.createElement('li');

newItem.textContent = 'New List Item';

**// Append the new item to a list**

const list = document.querySelector('ul');

list.appendChild(newItem);

**// Insert new item before the first list item**

const firstItem = list.firstElementChild;

list.insertBefore(newItem, firstItem);

**// Remove an element**

const itemToRemove = document.querySelector('#item-to-remove');

list.removeChild(itemToRemove);

**Activity**:

1. Create an HTML list with items.
2. Write JavaScript to:
   * Dynamically add new list items.
   * Remove a specific list item.
   * Insert a new list item before the first item.

**3. Practical Example: To-Do List Application**

**Objective**:

* Build a simple to-do list application that demonstrates DOM traversal and manipulation.

**HTML Structure**:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>To-Do List</title>

</head>

<body>

<h1>To-Do List</h1>

<input type="text" id="new-task" placeholder="Enter new task">

<button id="add-task">Add Task</button>

<ul id="task-list">

<!-- List items will be added here -->

</ul>

<script src="app.js"></script>

</body>

</html>

**JavaScript (app.js)**:

// Function to add a new task

function addTask() {

const taskInput = document.getElementById('new-task');

const taskValue = taskInput.value;

if (taskValue.trim() === '') {

alert('Task cannot be empty');

return;

}

// Create new task item

const newItem = document.createElement('li');

newItem.textContent = taskValue;

// Create remove button for the task

const removeButton = document.createElement('button');

removeButton.textContent = 'Remove';

removeButton.addEventListener('click', () => {

taskList.removeChild(newItem);

});

newItem.appendChild(removeButton);

// Add new task item to the list

const taskList = document.getElementById('task-list');

taskList.appendChild(newItem);

// Clear input field

taskInput.value = '';

}

// Event listener for add task button

document.getElementById('add-task').addEventListener('click', addTask);

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>To-Do List</title>**

**</head>**

**<body>**

**<h1>To-Do List</h1>**

**<input type="text" id="new-task" placeholder="Enter new task">**

**<button id="add-task" onclick="addTask()">Add Task</button>**

**<ul id="task-list">**

**<!-- List items will be added here -->**

**</ul>**

**<script>**

**function addTask() {**

**const taskInput = document.getElementById('new-task');**

**const taskValue = taskInput.value;**

**if (taskValue.trim() === '') {**

**alert('Task cannot be empty');**

**return;**

**}**

**// Create new task item**

**const newItem = document.createElement('li');**

**newItem.textContent = taskValue;**

**// Create remove button for the task**

**const removeButton = document.createElement('button');**

**removeButton.textContent = 'Remove';**

**removeButton.onclick = function() {**

**const taskList = document.getElementById('task-list');**

**taskList.removeChild(newItem);**

**};**

**newItem.appendChild(removeButton);**

**// Add new task item to the list**

**const taskList = document.getElementById('task-list');**

**taskList.appendChild(newItem);**

**// Clear input field**

**taskInput.value = '';**

**}**

**</script>**

**</body>**

**</html>**

**Activity**:

1. Implement the to-do list application.
2. Test the application to ensure tasks can be added and removed.

**Assignment**:

* **Task**: Enhance the to-do list application by adding the following features:
  1. **Task Completion**: Add a checkbox to mark tasks as completed. Completed tasks should be styled differently (e.g., strikethrough text).
  2. **Save Tasks**: Implement a feature to save tasks in the browser's local storage so that tasks persist after page reloads.

### ****Document Object Properties****

The document object represents the entire HTML document and provides properties and methods to interact with it. Here are some key properties of the document object:

#### ****1.**** document.documentElement

* **Description**: Returns the root element of the document, which is typically the <html> element.
* **Usage**:

let rootElement = document.documentElement;

console.log(rootElement); // <html>...</html>

#### ****2.**** document.head

* **Description**: Returns the <head> element of the document.
* **Usage**:

let headElement = document.head;

console.log(headElement); // <head>...</head>

#### ****3.**** document.body

* **Description**: Returns the <body> element of the document.
* **Usage**:

let bodyElement = document.body;

console.log(bodyElement); // <body>...</body>

#### ****4.**** document.title

* **Description**: Gets or sets the title of the document, which is displayed in the browser's title bar or tab.
* **Usage**:

// Get the title

let title = document.title;

console.log(title); // "Current Document Title"

// Set the title

document.title = "New Title";

#### ****5.**** document.URL

* **Description**: Returns the complete URL of the document.
* **Usage**:

let url = document.URL;

console.log(url); // "https://example.com/page.html"

#### ****6.**** document.baseURI

* **Description**: Returns the base URI of the document, which is the URL used as a base for resolving relative URLs.
* **Usage**:

let baseUri = document.baseURI;

console.log(baseUri); // "https://example.com/"

#### ****7.**** document.documentMode

* **Description**: Returns the mode of the document, such as the IE compatibility mode. This property is specific to Internet Explorer and may not be available in all browsers.
* **Usage**:

let mode = document.documentMode;

console.log(mode); // Number representing the document mode, e.g., 5 for IE5

#### ****8.**** document.implementation

* **Description**: Returns a DOMImplementation object that provides methods to create new DOM documents.
* **Usage**:

let implementation = document.implementation;

console.log(implementation); // DOMImplementation object

#### ****9.**** document.forms

* **Description**: Returns a collection of all <form> elements in the document.
* **Usage**:

let forms = document.forms;

console.log(forms); // HTMLCollection of <form> elements

#### ****10.**** document.images

* **Description**: Returns a collection of all <img> elements in the document.
* **Usage**:

let images = document.images;

console.log(images); // HTMLCollection of <img> elements

#### ****11.**** document.scripts

* **Description**: Returns a collection of all <script> elements in the document.
* **Usage**:

let scripts = document.scripts;

console.log(scripts); // HTMLCollection of <script> elements

### ****Example Combining Document Object Properties****

Here's a simple example that demonstrates the use of some of these document properties:

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>Document Object Properties Example</title>**

**</head>**

**<body>**

**<h1 id="header">Document Object Model</h1>**

**<p>Check the console for examples of document properties.</p>**

**<script>**

**// Document properties**

**console.log('Document Title:', document.title); // Get and log the title**

**document.title = 'New Title'; // Set a new title**

**console.log('Document URL:', document.URL); // Get and log the URL**

**console.log('Document Base URI:', document.baseURI); // Get and log the base URI**

**// Querying elements**

**let header = document.querySelector('#header');**

**console.log('Header Element:', header); // Log the header element**

**// Changing content**

**header.textContent = 'Updated Header Text'; // Change the text content of the header**

**// Manipulating styles**

**header.style.color = 'blue'; // Change the color of the header text**

**// Create and append a new element**

**let newElement = document.createElement('p');**

**newElement.textContent = 'This is a new paragraph!';**

**document.body.appendChild(newElement);**

**</script>**

**</body>**

**</html>**

Certainly! Here’s a detailed overview of the most commonly used methods of the document object, which allow you to interact with and manipulate the content of a webpage.

### ****Document Object Methods****

The document object provides a variety of methods to access, create, modify, and delete elements within the HTML document. Here’s a breakdown of some of the most commonly used methods:

#### ****1.**** document.getElementById(id)

* **Description**: Returns the element with the specified ID.
* **Usage**:

let element = document.getElementById('myId');

#### ****2.**** document.getElementsByClassName(className)

* **Description**: Returns a live HTMLCollection of elements with the specified class name.
* **Usage**:

let elements = document.getElementsByClassName('myClass');

#### ****3.**** document.getElementsByTagName(tagName)

* **Description**: Returns a live HTMLCollection of elements with the specified tag name.
* **Usage**:

let elements = document.getElementsByTagName('div');

#### ****4.**** document.querySelector(selector)

* **Description**: Returns the first element that matches the specified CSS selector.
* **Usage**:

let element = document.querySelector('.myClass'); // First element with class "myClass"

#### ****5.**** document.querySelectorAll(selector)

* **Description**: Returns a NodeList of all elements that match the specified CSS selector.
* **Usage**:

let elements = document.querySelectorAll('.myClass'); // All elements with class "myClass"

#### ****6.**** document.createElement(tagName)

* **Description**: Creates a new element with the specified tag name.
* **Usage**:

let newElement = document.createElement('div');

#### ****7.**** document.createTextNode(text)

* **Description**: Creates a new text node with the specified text.
* **Usage**:

let textNode = document.createTextNode('This is a text node');

#### ****8.**** document.appendChild(childNode)

* **Description**: Appends a child node to the end of the list of children of a specified parent node.
* **Usage**:

let parent = document.getElementById('parent');

let child = document.createElement('p');

child.textContent = 'New Child';

parent.appendChild(child);

#### ****9.**** document.removeChild(childNode)

* **Description**: Removes a specified child node from the DOM.
* **Usage**:

let parent = document.getElementById('parent');

let child = document.getElementById('child');

parent.removeChild(child);

#### ****10.**** document.replaceChild(newChild, oldChild)

* **Description**: Replaces a specified child node with a new node.
* **Usage**:

let parent = document.getElementById('parent');

let oldChild = document.getElementById('oldChild');

let newChild = document.createElement('p');

newChild.textContent = 'New Child';

parent.replaceChild(newChild, oldChild);

#### ****11.**** document.write(html)

* **Description**: Writes a string of text to the document stream (typically used during page loading).
* **Usage**:

document.write('<p>New content written to the document.</p>');

#### ****12.**** document.open()

* **Description**: Opens the document stream for writing. Usually used with document.write().
* **Usage**:

document.open();

document.write('<p>Overwritten content.</p>');

document.close();

#### ****13.**** document.close()

* **Description**: Closes the document stream opened by document.open().
* **Usage**:

document.close();

### ****Example Combining Document Object Methods****

Here’s a practical example using some of these document object methods:

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document Object Methods Example</title>

</head>

<body>

<div id="container">

<h1>Document Object Methods</h1>

</div>

<button id="addParagraph">Add Paragraph</button>

<button id="removeParagraph">Remove Paragraph</button>

<script>

// Add a new paragraph to the container

document.getElementById('addParagraph').addEventListener('click', function() {

let container = document.getElementById('container');

let newParagraph = document.createElement('p');

newParagraph.textContent = 'This is a new paragraph added dynamically.';

container.appendChild(newParagraph);

});

// Remove the last paragraph from the container

document.getElementById('removeParagraph').addEventListener('click', function() {

let container = document.getElementById('container');

let paragraphs = container.getElementsByTagName('p');

if (paragraphs.length > 0) {

container.removeChild(paragraphs[paragraphs.length - 1]);

}

});

</script>

</body>

</html>

---------------------------------------------------------------------------

**HTML element properties:**

HTML element properties are characteristics or attributes of DOM elements (HTML elements) that can be accessed or modified using JavaScript. These properties allow you to interact with and manipulate the appearance, behavior, and content of elements dynamically.

Here's a list of common **HTML element properties**:

**1. innerHTML**

* Represents the HTML content inside an element.
* You can set or get the content of an element as an HTML string.
* Example:

const element = document.getElementById('myDiv');

element.innerHTML = "<p>New content</p>"; // Sets new HTML content

console.log(element.innerHTML); // Gets the current HTML content

**2. textContent**

* Represents the text content inside an element, excluding HTML tags.
* It retrieves or sets the plain text content of an element.
* Example:

const element = document.getElementById('myDiv');

element.textContent = "New plain text"; // Sets new text

console.log(element.textContent); // Gets the current text content

**3. value**

* Used for form elements like <input>, <textarea>, and <select>. It gets or sets the value entered by the user or the default value of the form field.
* Example:

const inputElement = document.getElementById('myInput');

inputElement.value = "New value"; // Sets a new value

console.log(inputElement.value); // Gets the current value

**4. style**

* Represents the inline CSS styles applied to an element.
* You can get or set specific styles for an element using the style property.
* Example:

const element = document.getElementById('myDiv');

element.style.backgroundColor = "lightblue"; // Sets the background color

console.log(element.style.backgroundColor); // Gets the current background color

**5. className / classList**

* **className**: Represents the class or classes of an element as a string.
* **classList**: A more modern property that allows you to add, remove, or toggle classes.
* Example using classList:

const element = document.getElementById('myDiv');

element.classList.add('newClass'); // Adds a class

element.classList.remove('oldClass'); // Removes a class

element.classList.toggle('hidden'); // Toggles a class on or off

console.log(element.className); // Logs the current class attribute

**6. id**

* Represents the id attribute of an element, which is unique within the document.
* You can set or get the id of an element.
* Example:

const element = document.getElementById('myDiv');

console.log(element.id); // Gets the id of the element

element.id = "newId"; // Sets a new id

**7. src**

* Used for <img>, <iframe>, <audio>, and <video> elements to specify the source URL.
* Example:

const imgElement = document.getElementById('myImage');

imgElement.src = "newImage.jpg"; // Sets the image source

console.log(imgElement.src); // Gets the current image source

**8. href**

* Used for anchor (<a>) elements to specify the hyperlink reference (URL).
* Example:

const linkElement = document.getElementById('myLink');

linkElement.href = "https://example.com"; // Sets the URL

console.log(linkElement.href); // Gets the current URL

**9. disabled**

* Used for form elements (<button>, <input>, <select>, etc.) to enable or disable them.
* Example:

const buttonElement = document.getElementById('myButton');

buttonElement.disabled = true; // Disables the button

console.log(buttonElement.disabled); // Checks if the button is disabled

**10. checked**

* Used for checkbox or radio button elements to check or uncheck them.
* Example:

const checkboxElement = document.getElementById('myCheckbox');

checkboxElement.checked = true; // Checks the checkbox

console.log(checkboxElement.checked); // Gets whether the checkbox is checked

**11. alt**

* Used for <img> elements to set or get the alternative text displayed when the image is not loaded.
* Example:

const imgElement = document.getElementById('myImage');

imgElement.alt = "A description of the image"; // Sets the alt text

console.log(imgElement.alt); // Gets the current alt text

**12. type**

* For form elements like <input>, it specifies the type (e.g., text, password, checkbox).
* Example:

const inputElement = document.getElementById('myInput');

console.log(inputElement.type); // Gets the input type

inputElement.type = "password"; // Changes the type to password

**13. hidden**

* If true, the element is hidden from the user.
* Example:

const element = document.getElementById('myDiv');

element.hidden = true; // Hides the element

**14. childElementCount**

* Represents the number of child elements an element has.
* Example:

const parentElement = document.getElementById('myDiv');

console.log(parentElement.childElementCount); // Gets the number of child elements

**15. outerHTML**

* Returns the entire HTML content of an element, including the element itself.
* Example:

const element = document.getElementById('myDiv');

console.log(element.outerHTML); // Logs the entire HTML of the element

**Mouse Events:::::::**

**Mouse events** are a category of events in JavaScript that are triggered when the user interacts with a webpage using a mouse or other pointing devices. These events allow developers to execute code when specific mouse actions occur, such as clicking, hovering, dragging, or moving the mouse over an element.

Here is a list of common **mouse events**:

**1. click**

* Triggered when the user clicks on an element (presses and releases the mouse button).
* Example:

const button = document.getElementById('myButton');

button.addEventListener('click', function() {

alert('Button clicked!');

});

**2. dblclick**

* Triggered when the user double-clicks on an element (clicks twice in quick succession).
* Example:

const div = document.getElementById('myDiv');

div.addEventListener('dblclick', function() {

alert('Element double-clicked!');

});

**3. mousedown**

* Triggered when the user presses the mouse button down (but doesn't release it yet).
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mousedown', function() {

console.log('Mouse button pressed down');

});

**4. mouseup**

* Triggered when the user releases the mouse button after pressing it.
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mouseup', function() {

console.log('Mouse button released');

});

**5. mouseover**

* Triggered when the mouse pointer moves over an element (enters the element’s boundaries).
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mouseover', function() {

console.log('Mouse is over the element');

});

**6. mouseout**

* Triggered when the mouse pointer leaves an element (exits the element’s boundaries).
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mouseout', function() {

console.log('Mouse left the element');

});

**7. mousemove**

* Triggered continuously as the mouse moves over an element.
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mousemove', function(event) {

console.log(`Mouse moved to X: ${event.clientX}, Y: ${event.clientY}`);

});

**8. mouseenter**

* Triggered when the mouse pointer enters an element (similar to mouseover, but doesn’t bubble up).
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mouseenter', function() {

console.log('Mouse entered the element');

});

**9. mouseleave**

* Triggered when the mouse pointer leaves an element (similar to mouseout, but doesn’t bubble up).
* Example:

const element = document.getElementById('myElement');

element.addEventListener('mouseleave', function() {

console.log('Mouse left the element');

});

**10. contextmenu**

* Triggered when the user right-clicks on an element to open the context menu.
* Example:

const element = document.getElementById('myElement');

element.addEventListener('contextmenu', function(event) {

event.preventDefault(); // Prevents the default context menu

alert('Right-click detected');

});

**Example: Simple Mouse Event Demonstration**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>Mouse Events Example</title>**

**<style>**

**#box {**

**width: 200px;**

**height: 200px;**

**background-color: lightblue;**

**text-align: center;**

**line-height: 200px;**

**margin: 50px auto;**

**}**

**</style>**

**</head>**

**<body>**

**<div id="box">Hover or Click Me</div>**

**<script>**

**const box = document.getElementById('box');**

**// Click event**

**box.addEventListener('click', function() {**

**alert('Box clicked!');**

**});**

**// Mouseover event**

**box.addEventListener('mouseover', function() {**

**box.style.backgroundColor = 'yellow';**

**box.textContent = 'Mouse Over!';**

**});**

**// Mouseout event**

**box.addEventListener('mouseout', function() {**

**box.style.backgroundColor = 'lightblue';**

**box.textContent = 'Hover or Click Me';**

**});**

**// Mousedown event**

**box.addEventListener('mousedown', function() {**

**box.textContent = 'Mouse Down';**

**});**

**// Mouseup event**

**box.addEventListener('mouseup', function() {**

**box.textContent = 'Mouse Up';**

**});**

**</script>**

**</body>**

**</html>**

**Explanation:**

* **click**: Triggers when the box is clicked.
* **mouseover**: Changes the background color and text when the mouse hovers over the box.
* **mouseout**: Reverts the background color and text when the mouse leaves the box.
* **mousedown**: Triggers when the mouse button is pressed on the box.
* **mouseup**: Triggers when the mouse button is released over the box.

Keyboard events allow developers to capture user input from the keyboard and respond to it in various ways. There are three main keyboard events in JavaScript:

1. **keydown**: Fires when a key is pressed down.
2. **keyup**: Fires when a key is released.
3. **keypress**: (Deprecated, replaced by keydown) Fires when a key is pressed, but this event doesn't capture all keys and is no longer recommended.

Here's a breakdown of each event:

**Keyboard Events Overview:**

* **keydown**: Captures any key press, including special keys (like Shift, Alt, etc.). It fires repeatedly if the key is held down.
* **keyup**: Captures the moment the key is released.
* **keypress**: Was used to capture printable characters only but doesn’t handle non-character keys well. Use keydown instead.

**Example Code with keydown and keyup:**

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Keyboard Events Demo</title>

<style>

#box {

width: 300px;

height: 50px;

background-color: lightgray;

margin: 50px auto;

text-align: center;

line-height: 50px;

border: 2px solid black;

}

#output {

text-align: center;

margin-top: 20px;

font-size: 1.2em;

}

</style>

</head>

<body>

<div id="box" tabindex="0">Press any key</div>

<div id="output"></div>

<script>

const box = document.getElementById('box');

const output = document.getElementById('output');

// Focus on the box so it can capture key events

box.focus();

// Handle keydown event

box.addEventListener('keydown', function(event) {

box.style.backgroundColor = 'lightblue';

output.textContent = `Key down: ${event.key} (Code: ${event.code})`;

});

// Handle keyup event

box.addEventListener('keyup', function(event) {

box.style.backgroundColor = 'lightgreen';

output.textContent = `Key up: ${event.key} (Code: ${event.code})`;

});

</script>

</body>

</html>

**Explanation of the Example:**

1. **keydown**: When a key is pressed, it changes the box's background color to blue and displays the key and its code.
2. **keyup**: When the key is released, the box turns green and shows the key and its code again.
3. **tabindex="0"**: This allows the div to receive focus, so it can capture keyboard events. Without this, only form elements (like input fields) can receive keyboard focus by default.

**Useful Properties in Keyboard Events:**

* **event.key**: Gives the actual key pressed (like "A", "Enter", "Shift", etc.).
* **event.code**: Provides the physical location of the key on the keyboard (like "KeyA", "Space", "Enter").
* **event.ctrlKey, event.shiftKey, event.altKey**: Boolean properties that tell you if the Ctrl, Shift, or Alt keys are being pressed.

**Activities for Students:**

1. **Task**: Create a keyboard event handler that changes the background color of the page when different keys are pressed (R for red, B for blue, etc.).
2. **Assignment**: Build a small typing practice game where students are given a random letter to type, and they need to press the correct key to score points.

**CallBack:**

A **callback function** is a function that is passed as an argument to another function and is executed after the first function has completed its operation. This allows you to call a function after some task or operation has been performed.

**Example of a Callback Function:**

Here’s a simple example of a callback function in JavaScript:

function greeting(name) {

console.log(`Hello, ${name}!`);

}

function processUserInput(callback) {

const name = prompt("Please enter your name:");

callback(name); // Here, we call the callback function and pass `name` as an argument

}

// Passing the greeting function as a callback

processUserInput(greeting);

**Explanation:**

1. greeting(name) is a simple function that takes a name parameter and prints a greeting message.
2. processUserInput(callback) is a function that accepts a function as a parameter (callback). It prompts the user to enter their name, and then calls the callback function with the entered name.
3. When you call processUserInput(greeting);, the greeting function is passed as a callback. Once the user inputs their name, the greeting function is executed with the name as the argument.

**Why Use Callback Functions?**

* **Asynchronous Operations**: Callbacks are often used in asynchronous operations such as event handling, network requests (e.g., fetch), file reading, and timers (like setTimeout), where you want to perform an action only after an asynchronous task has completed.

**Example of a Callback in Asynchronous Code:**

function downloadData(callback) {

setTimeout(() => {

console.log("Data downloaded");

callback(); // This callback will be executed after the data is downloaded

}, 2000); // Simulates a 2-second download time

}

function processData() {

console.log("Processing data...");

}

downloadData(processData);

**----------------------------------------------------------------------------------------------------------------Arrow Function:**

Arrow functions are a concise way to write functions in JavaScript. Introduced in ES6 (ECMAScript 2015), they offer a more compact syntax compared to traditional function expressions.

**Syntax**

Here’s the basic syntax of an arrow function:

javascript

Copy code

(param1, param2, ...) => {

// function body

}

For example, instead of writing a traditional function like this:

javascript

Copy code

const add = function(a, b) {

return a + b;

};

You can use an arrow function:

javascript

Copy code

const add = (a, b) => {

return a + b;

};

Or, if there’s only one expression in the function body (and you’re just returning it), you can omit the curly braces and the return keyword:

javascript

Copy code

const add = (a, b) => a + b;

**Key Features of Arrow Functions**

1. **Concise Syntax**  
   Arrow functions allow you to write cleaner, shorter function expressions.

Example:

javascript

Copy code

const greet = (name) => `Hello, ${name}!`;

console.log(greet('Alice')); // Output: "Hello, Alice!"

1. **Implicit Return**  
   When the function body contains only one expression, the result is implicitly returned without needing to use the return keyword.

Example:

javascript

Copy code

const square = x => x \* x;

console.log(square(5)); // Output: 25

1. **No this Binding**  
   Arrow functions do **not** have their own this. Instead, they inherit this from the surrounding context where they are defined. This makes them especially useful in situations where the this value would otherwise be lost or require binding.

Example (without arrow function):

javascript

Copy code

function Person() {

this.age = 0;

setInterval(function() {

this.age++; // `this` refers to the global object, not the person instance

console.log(this.age); // NaN or undefined in strict mode

}, 1000);

}

const person = new Person();

With arrow functions, this refers to the Person object:

javascript

Copy code

function Person() {

this.age = 0;

setInterval(() => {

this.age++; // `this` refers to the person object

console.log(this.age); // Correctly logs the incremented age

}, 1000);

}

const person = new Person();

1. **No arguments Object**  
   Arrow functions do **not** have their own arguments object. If you need to access the arguments of a function, you can use the rest parameters syntax.

Example:

javascript

Copy code

const showArguments = (...args) => console.log(args);

showArguments(1, 2, 3); // Output: [1, 2, 3]

**Arrow Functions with No Parameters**

If an arrow function takes no parameters, you just use empty parentheses:

const sayHello = () => console.log('Hello!');

**Examples**

**1. Simple Arrow Function**

const double = x => x \* 2;

console.log(double(4)); // Output: 8

**2. Arrow Function with Multiple Parameters**

const multiply = (a, b) => a \* b;

console.log(multiply(2, 3)); // Output: 6

**3. Arrow Function Returning an Object**

To return an object literal from an arrow function, wrap it in parentheses:

const getPerson = (name, age) => ({ name: name, age: age });

console.log(getPerson('Alice', 30)); // Output: { name: "Alice", age: 30 }

**4. Arrow Functions with Higher-Order Functions**

Arrow functions work great with array methods like .map(), .filter(), .reduce(), etc.

Example using .map():

const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // Output: [2, 4, 6, 8, 10]

**When to Use Arrow Functions**

* When you need a simple, concise function expression.
* When you want to avoid using .bind() to maintain the correct this context.
* For shorter callback functions, especially in array methods like .map(), .filter(), and .reduce().

----------------------------------------------------------------------------------------------------------------

ES6 introduced several new array methods that make working with arrays more efficient and expressive. Here’s a breakdown of some of the most important ES6 array methods, with examples:

**1. forEach()**

The forEach() method executes a provided function once for each array element.

const numbers = [1, 2, 3, 4];

numbers.forEach(num => console.log(num));

// Output: 1 2 3 4

**2. map()**

The map() method creates a new array populated with the results of calling a provided function on every element in the array.

const numbers = [1, 2, 3];

const squared = numbers.map(num => num \* num);

console.log(squared); // Output: [1, 4, 9]

**3. filter()**

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

const numbers = [1, 2, 3, 4, 5];

const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // Output: [2, 4]

**4. reduce()**

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((accumulator, current) => accumulator + current, 0);

console.log(sum); // Output: 10

**5. find()**

The find() method returns the value of the first element in the array that satisfies the provided testing function.

const numbers = [1, 2, 3, 4];

const firstEven = numbers.find(num => num % 2 === 0);

console.log(firstEven); // Output: 2

**6. findIndex()**

The findIndex() method returns the index of the first element in the array that satisfies the provided testing function.

const numbers = [1, 2, 3, 4];

const index = numbers.findIndex(num => num > 2);

console.log(index); // Output: 2

**7. every()**

The every() method tests whether all elements in the array pass the test implemented by the provided function.

const numbers = [1, 2, 3, 4];

const allEven = numbers.every(num => num % 2 === 0);

console.log(allEven); // Output: false

**8. some()**

The some() method tests whether at least one element in the array passes the test implemented by the provided function.

const numbers = [1, 2, 3, 4];

const hasEven = numbers.some(num => num % 2 === 0);

console.log(hasEven); // Output: true

**9. includes()**

The includes() method determines whether an array includes a certain value among its entries, returning true or false.

const fruits = ["apple", "banana", "cherry"];

const hasBanana = fruits.includes("banana");

console.log(hasBanana); // Output: true

**10. sort()**

The sort() method sorts the elements of an array in place and returns the sorted array. By default, it sorts elements as strings.

const numbers = [3, 1, 4, 2];

numbers.sort((a, b) => a - b); // Sort numerically

console.log(numbers); // Output: [1, 2, 3, 4]

**11. concat()**

The concat() method is used to merge two or more arrays. It does not modify the original arrays but returns a new array.

const arr1 = [1, 2];

const arr2 = [3, 4];

const merged = arr1.concat(arr2);

console.log(merged); // Output: [1, 2, 3, 4]

**12. slice()**

The slice() method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included).

const numbers = [1, 2, 3, 4, 5];

const sliced = numbers.slice(1, 3);

console.log(sliced); // Output: [2, 3]

**13. splice()**

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

const numbers = [1, 2, 3, 4];

numbers.splice(2, 1, 5); // Removes 1 element at index 2 and inserts 5

console.log(numbers); // Output: [1, 2, 5, 4]

These are some of the most important ES6 array methods that simplify array manipulation and functional programming patterns in JavaScript.

**Activity:**const products = [ { id: 1, name: "Laptop", price: 1200, category: "Electronics", stock: 10 }, { id: 2, name: "Smartphone", price: 800, category: "Electronics", stock: 15 }, { id: 3, name: "Headphones", price: 150, category: "Accessories", stock: 25 }, { id: 4, name: "Keyboard", price: 100, category: "Accessories", stock: 30 }, { id: 5, name: "Monitor", price: 300, category: "Electronics", stock: 8 }, { id: 6, name: "Mouse", price: 50, category: "Accessories", stock: 40 }, { id: 7, name: "Desk Chair", price: 250, category: "Furniture", stock: 5 }, { id: 8, name: "Desk", price: 400, category: "Furniture", stock: 7 }, { id: 9, name: "Tablet", price: 500, category: "Electronics", stock: 12 }, { id: 10, name: "Smartwatch", price: 200, category: "Electronics", stock: 20 } ];

**Find All Products in a Specific Category (e.g., "Electronics")**:

**Find a Product by Name**:­­

**Calculate the Total Value of All Stock**:

**Filter Out of Stock Products**:

**List Products Sorted by Stock (Low to High)**:

**Filter Out of Stock Products**:

**Bootstrap**

Typography in Bootstrap refers to the system of styles and utilities that manage the presentation of text on a webpage. Bootstrap offers a variety of typography classes and settings that help developers control the size, style, alignment, and spacing of text elements in a consistent and responsive way.

### Key Features of Bootstrap Typography

1. **Headings**: Bootstrap provides predefined classes for all six heading levels (<h1> to <h1 class="display-1">This is an h1 heading</h1>

<h2>This is an h2 heading</h2>

<h3>This is an h3 heading</h3>

1. **Display Headings**: Larger, more prominent headings using .display-1 to .display-6 classes.

<h1 class="display-1">Display 1</h1>

<h1 class="display-2">Display 2</h1>

1. **Lead Paragraphs**: A lead paragraph stands out using .lead, providing larger font size and lighter weight.

<p class="lead">This is a lead paragraph. It stands out from regular text.</p>

1. **Inline Text Elements**: Special classes for inline elements to style text like bold, italic, small, marked, etc.

<p>You can use <mark>highlighted text</mark> for important information.</p>

<p>This text is <small>small</small> for fine print.</p>

<p><strong>Bold text</strong> can emphasize a point.</p>

<p><em>Italicized text</em> can add style.</p>

1. **Blockquotes**: Styled quotations with .blockquote. Optionally, add blockquote-footer for source attribution.

<blockquote class="blockquote">

<p>This is a blockquote.</p>

<footer class="blockquote-footer">Someone famous</footer>

</blockquote>

1. **Text Alignment**: Use text alignment classes like .text-left, .text-center, and .text-right to align text. With responsive alignment classes, you can adjust alignment at different breakpoints.

<p class="text-center">This text is centered.</p>

<p class="text-md-right">Right aligned on medium and larger screens.</p>

1. **Text Colors and Backgrounds**: Bootstrap offers classes for text colors (.text-primary, .text-secondary, .text-success, etc.) and background colors (.bg-primary, .bg-light, etc.).

<p class="text-primary">Primary text</p>

<p class="text-danger">Danger text</p>

<p class="bg-warning text-white">Text with warning background</p>

1. **Text Transform and Font Weight**: Use classes like .text-uppercase, .text-lowercase, .text-capitalize for text transformation, and .fw-bold, .fw-normal, .fw-light for font weights.

<p class="text-uppercase">Uppercase text</p>

<p class="fw-bold">Bold text</p>

<p class="fw-light">Light weight text</p>

1. **Line Height**: Bootstrap provides classes like .lh-1, .lh-sm, .lh-base, .lh-lg for line heights.

<p class="lh-sm">This text has a small line height.</p>

1. **Monospace Text**: Use .font-monospace for monospace fonts.

<p class="font-monospace">This is monospace text.</p>

**Responsive Image:**

In Bootstrap, images can be styled and manipulated using a variety of classes to make them responsive, rounded, aligned, and more. The framework makes it easy to handle images, ensuring they look good across different screen sizes and layouts.

### Key Features for Working with Images in Bootstrap

1. **Responsive Images**: Bootstrap provides the .img-fluid class to make images responsive. This class ensures that the image scales well across different screen sizes by setting the max-width: 100% and height: auto.

<img src="image.jpg" class="img-fluid" alt="Responsive Image">

1. **Image Shapes**: You can apply classes to make images appear rounded, circular, or as thumbnails.
   * **Rounded Corners**: The .rounded class rounds the corners of the image.

<img src="image.jpg" class="rounded" alt="Rounded Image">

* + **Circle Image**: The .rounded-circle class creates a circular image by making the image's borders fully rounded.

<img src="image.jpg" class="rounded-circle" alt="Circular Image">

* + **Thumbnail**: The .img-thumbnail class adds a border and some padding, making the image look like a thumbnail.

<img src="image.jpg" class="img-thumbnail" alt="Thumbnail Image">

1. **Aligning Images**: Bootstrap provides utility classes for aligning images. You can align images to the left, right, or center using text alignment classes.
   * **Left Align**:

<img src="image.jpg" class="float-start" alt="Left Aligned Image">

* + **Right Align**:

<img src="image.jpg" class="float-end" alt="Right Aligned Image">

* + **Center Align**:

<img src="image.jpg" class="mx-auto d-block" alt="Centered Image">

1. **Image Grids**: You can combine Bootstrap's grid system with images to create responsive image galleries. Using the col classes and rows, images can be arranged in columns.

<div class="row">

<div class="col-md-4">

<img src="image1.jpg" class="img-fluid" alt="Image 1">

</div>

<div class="col-md-4">

<img src="image2.jpg" class="img-fluid" alt="Image 2">

</div>

<div class="col-md-4">

<img src="image3.jpg" class="img-fluid" alt="Image 3">

</div>

</div>

1. **Image Placeholders**: Bootstrap provides easy-to-use classes to add placeholder images from the https://via.placeholder.com service, which are often used during development.

<img src="https://via.placeholder.com/150" class="img-fluid" alt="Placeholder">

1. **Figure Component**: Bootstrap offers the <figure> component to wrap an image with a caption using the <figcaption> tag.

<figure class="figure">

<img src="image.jpg" class="figure-img img-fluid rounded" alt="Figure Image">

<figcaption class="figure-caption">This is a caption for the image.</figcaption>

</figure>

Utility classes in Bootstrap are a powerful feature designed to enhance productivity and consistency in the styling of elements without needing to write custom CSS. They allow for the quick application of styles such as margin, padding, color, background, and more, through predefined class names. These classes help avoid writing complex CSS rules and enable rapid development of responsive designs.

Here’s a detailed look at some of the most commonly used Bootstrap utility classes:

**1. Spacing (Margin & Padding)**

* **Margin (m-)** and **Padding (p-)** classes control the space around or inside elements.
* They follow a simple structure: m for margin and p for padding, followed by a size number (0-5 for fixed sizes, auto for automatic margin).

**Example:**

<div class="m-3 p-3 bg-light">Margin and Padding Applied</div>

You can also apply margin and padding to specific sides:

* **mt-\***: Margin top
* **mb-\***: Margin bottom
* **ml-\***: Margin left
* **mr-\***: Margin right
* **pt-\***: Padding top
* **pb-\***: Padding bottom

**Example:**

<div class="mt-4 mb-2 p-3 bg-secondary text-white">Custom top and bottom margins</div>

**2. Text Alignment & Styling**

* **text-start**: Left-align text.
* **text-center**: Center-align text.
* **text-end**: Right-align text.
* **fw-bold**: Bold text.
* **text-uppercase**: Uppercase text.

**Example:**

<p class="text-center fw-bold">This is centered bold text.</p>

**3. Display**

* **d-** utilities control the display property of an element.
* **d-block**, **d-inline**, **d-none**, **d-flex**, etc., to set the display value for different elements.

**Example:**

<div class="d-none d-md-block">This element is hidden on small screens, visible on medium and larger screens.</div>

**4. Color & Background**

* **text-\*** classes for text color (e.g., text-primary, text-danger, text-success, etc.).
* **bg-\*** classes for background colors (e.g., bg-primary, bg-success, bg-warning, etc.).

**Example:**

<div class="text-success">This text is green.</div>

<div class="bg-warning p-3">This background is yellow.</div>

**5. Sizing**

* **w-**: Sets width of an element (e.g., w-50 for 50% width).
* **h-**: Sets height of an element (e.g., h-25 for 25% height).
* **min-vh-100**: Makes the element’s height 100% of the viewport height.

**Example:**

<div class="w-50 h-50 bg-light">This box takes 50% of the width and height.</div>

**6. Border**

* **border**: Adds a border to an element.
* **border-0**: Removes the border.
* **border-top, border-bottom, etc.**: Adds a border to specific sides.
* **rounded**, **rounded-circle**: Applies border-radius.

**Example:**

<div class="border border-danger rounded p-3">This element has a red border and rounded corners.</div>

**7. Visibility**

* **visible**: Ensures the element is visible.
* **invisible**: Makes the element invisible but keeps its space in the layout.

**Example:**

<div class="invisible">This content is invisible but occupies space.</div>

**8. Positioning**

* **position-static**, **position-relative**, **position-absolute**, **position-fixed**, and **position-sticky** control the positioning behavior of elements.

**Example:**

<div class="position-fixed bottom-0 end-0 p-3 bg-dark text-white">Fixed positioned element</div>

**9. Flexbox Utilities**

Bootstrap comes with several flexbox utilities to easily manage layouts and align elements.

* **d-flex**: Activates flexbox on the element.
* **justify-content-start**, **justify-content-center**, **justify-content-end**: Aligns flex items horizontally.
* **align-items-start**, **align-items-center**, **align-items-end**: Aligns flex items vertically.

**Example:**

<div class="d-flex justify-content-between">

<div>Item 1</div>

<div>Item 2</div>

<div>Item 3</div>

</div>

**10. Shadows**

* **shadow**, **shadow-sm**, **shadow-lg**: Adds varying levels of box shadows.
* **shadow-none**: Removes box shadow.

**Example:**

<div class="shadow-lg p-3 bg-body rounded">This box has a large shadow.</div>

**11. Float Utilities**

* **float-start**: Floats an element to the left.
* **float-end**: Floats an element to the right.
* **float-none**: Removes float behavior.

**Example:**

<img src="image.jpg" class="float-start me-3" alt="Float Example">

<p>This text wraps around the image floated to the left.</p>