Javascript(DOM ,html-css ,debugging , etc..)

1) Prettier-Code formatter

Navigate to File > Preferences > Settings ->Text Editor->Formatting->Format on Save ✔
Extension ->Install Prettier-Code formatter and go to Settings-> Default Formatter -> Select respective installed Code formatter.

Enabling Format on Save with a formatter like Prettier ensures consistent, readable, and clean code automatically. It saves time, reduces errors, improves code reviews, supports team collaboration, and maintains project-wide formatting standards effortlessly.

Before Saving (Unformatted Code)

```
function greet(name){console.log("Hello, "+name+"!");}
greet('John');
```

After Saving (Formatted Code)

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

greet("John");
```

Key Improvements After Saving:

1. **Consistent Indentation**: Properly indents the function body.
2. **Spaced Syntax**: Adds spaces around curly braces and operators for better readability.
3. **Uniform Quotes**: Ensures all strings use double quotes (or single, based on your Prettier config).

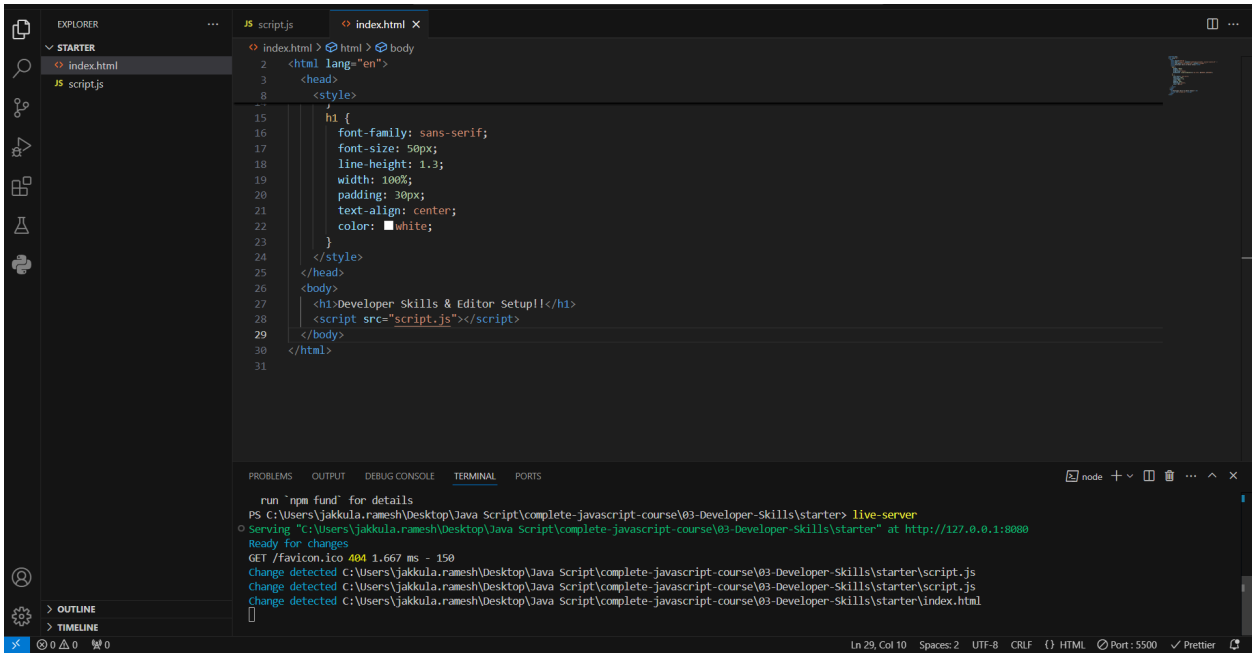
2) Live Server extension:

The **Live Server** extension for VS Code is a powerful tool that provides a live preview of your web pages in the browser, automatically reloading the page when you save changes to your code. Here's what it offers:
Extension ->live server

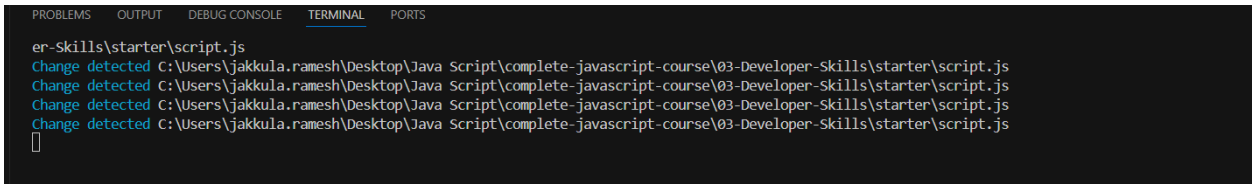
Or

We can also use a **live server** using an external server environment like **node js.**
npm install live-server -g

It will install npm modules in the respective working folder , so whenever we give command **live-server** then it automatically opens the index.hml page.



So whenever we make changes on any file it will be notified in terminal.



Developer Skills && Editor Setup
Debugging with console and breakpoints

Common console Methods:

console.log(): Outputs general information (variables, messages).
const x = 10;
const y = 20;
console.log(x + y); // Output: 30
console.warn(): Outputs a warning message (useful for indicating potential issues).
console.warn('This is a warning message!');
console.error(): Outputs an error message (useful for debugging errors).
console.error('Something went wrong!');
console.table(): Outputs data in a tabular format (good for arrays and objects).
const person = { name: 'John', age: 30 };
console.table(person);

console.group() and console.groupEnd(): Groups related log messages together for better readability.

```
console.group('User Info');
console.log('Name: John');
console.log('Age: 30');
console.groupEnd();
```

console.time() and console.timeEnd(): Measures the time taken by a block of code.

```
console.time('MyTimer');
for (let i = 0; i < 1000; i++) {} // Some code block
console.timeEnd('MyTimer'); // Outputs the time taken
```

```
const person = {
  nam: "dhoni",
  age: 67,
  job: "cricketer",
};
console.log(person);
console.warn("hello");
console.error("error");
console.table(person); //it will create a table
console.group(" hi good morning");
console.groupEnd();
console.time("MyTimer");
for (let i = 0; i < 30; i++) {} // Some code block
console.timeEnd("MyTimer"); // Outputs the time taken
```

Debugging
Inspect the page on the browser and click on the **Source** tab and then **page**.
Click on which .js file you want to keep breakpoints .
Keep breakpoints and clear consoles . Restart the server and observe the value in scope contour debugging.

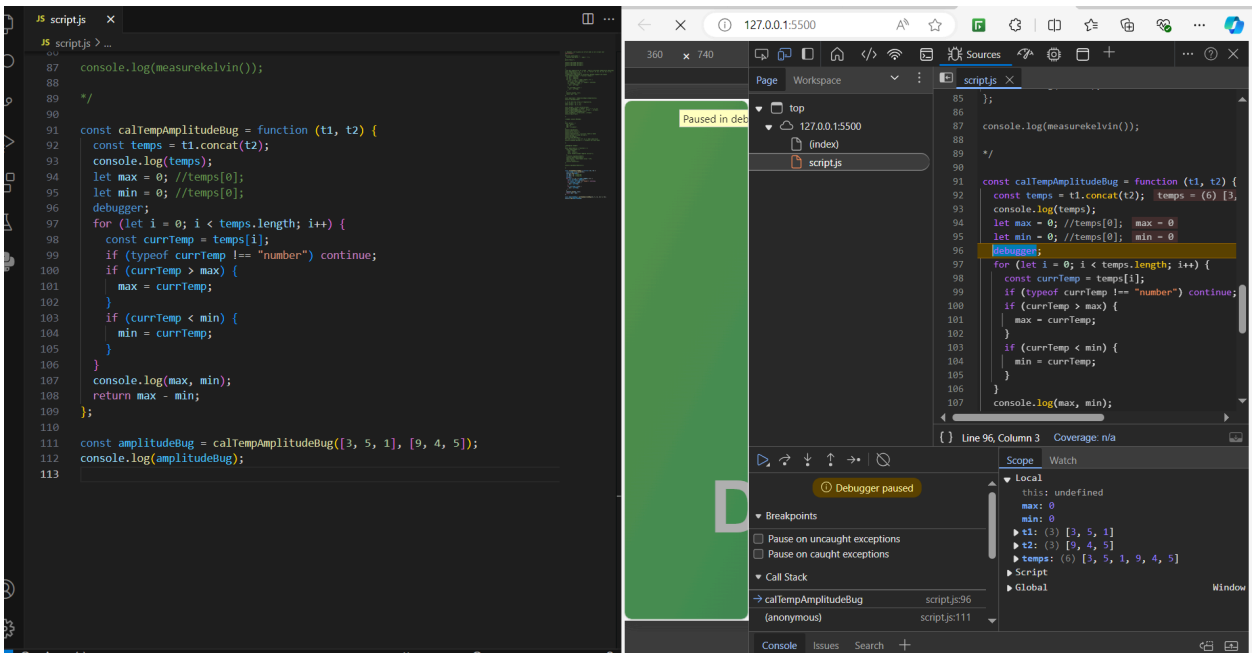
Or

Using debugger Statement

You can also use the debugger statement in your code to pause execution at specific points.

Shortcut to open debugger is write a line code in code as debugger;
Then it will automatically take us to the debugger page .

```
const calTempAmplitudeBug = function (t1, t2) {
  const temps = t1.concat(t2);
  console.log(temps);
  let max = 0; //temps[0];
  let min = 0; //temps[0];
  debugger;
  for (let i = 0; i < temps.length; i++) {
    const currTemp = temps[i];
    if (typeof currTemp !== "number") continue;
    if (currTemp > max) {
```




```
// Remember, we're gonna use strict mode in all scripts now!
"use strict";
/*
function greet(name) {
  console.log("Hello, " + name + "!");
}
greet("dhoni");

console.log("good morning");
console.log("good morning");
console.log("good morning");
*/

/*
```

```
//one day temperatures at a home , need to calculate temperature amplitude.
const temperatures = [3, -2, -6, -1, "error", 9, 13, 17, 15, 14, 9, 5];
console.log(temperatures);
//temperature amplitude is difference between highest and lowest
const temperatureAmpli = function (temps) {
  let max = temps[0];
  let min = temps[0];
  for (let i = 0; i < temps.length; i++) {
    const currTemp = temps[i];
    if (typeof currTemp !== "number") continue;
    if (currTemp > max) {
      max = currTemp;
    }
    if (currTemp < min) {
      min = currTemp;
    }
  }
  console.log(max, min);
  return max - min;
};

const amplitude = temperatureAmpli(temperatures);
console.log(amplitude);

// if we have two arrays of temperatures.
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];

const merged = array1.concat(array2);
console.log(merged); // Output: [1, 2, 3, 4, 5, 6]
const usingSpreadOperator = [...array1, ...array2];
console.log(usingSpreadOperator);
const usingPush = array1.push(...array2);
console.log(array1);

*/

//Common console Methods:

/*
const person = {
  nam: "dhoni",
  age: 67,
  job: "cricketer",
};
console.log(person);
```

```
console.warn("hello");
console.error("error");
console.table(person); //it will create a table
console.group(" hi good morning");
console.groupEnd();
console.time("MyTimer");
for (let i = 0; i < 30; i++) {} // Some code block
console.timeEnd("MyTimer"); // Outputs the time taken

*/

//debugging example
/*
const measurekelvin = function () {
  const measurement = {
    type: "temp",
    unit: "celsius",
    value: Number(prompt("degree celsius")),
  };
  //console.log(measurement);
  console.table(measurement);
  const kelvin = measurement.value + 273;
  return kelvin;
  console.log(kelvin);
};

console.log(measurekelvin());

*/
/*
//Below we have changed temp[0] to 0 values , then the result will be impacted. so we breakpoint
and observe the flow
const calTempAmplitudeBug = function (t1, t2) {
  const temps = t1.concat(t2);
  console.log(temps);
  let max = 0; //temps[0];
  let min = 0; //temps[0];
  debugger;
  for (let i = 0; i < temps.length; i++) {
    const currTemp = temps[i];
    if (typeof currTemp !== "number") continue;
    if (currTemp > max) {
      max = currTemp;
    }
    if (currTemp < min) {
      min = currTemp;
```

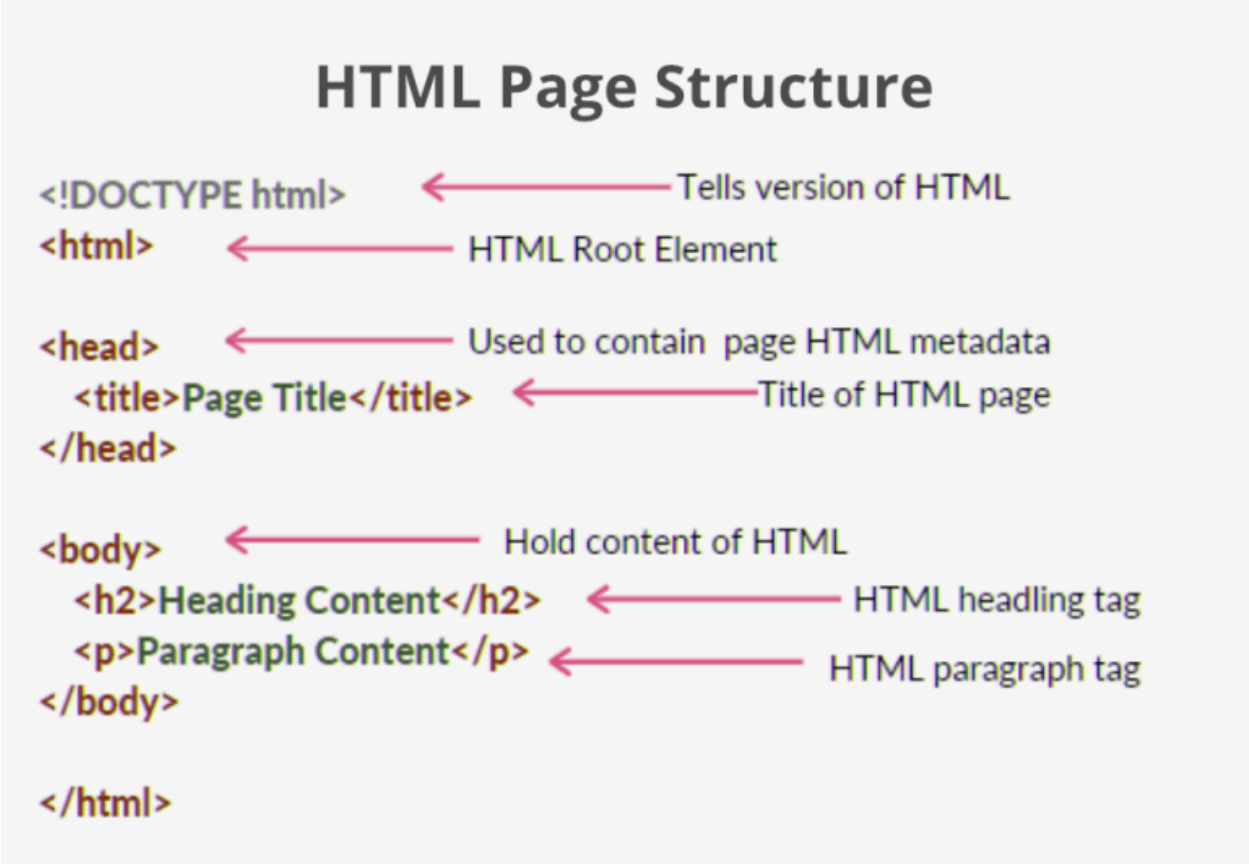
```
    }
  }
  console.log(max, min);
  return max - min;
};

const amplitudeBug = calTempAmplitudeBug([3, 5, 1], [9, 4, 5]);
console.log(amplitudeBug);

*/
/*
const arr = [17, 21, 23]; // [12, 5, -5, 0, 4]

console.log(arr);
const printForecast = function (arr) {
  let count = 1;
  let str = "";
  for (let i = 0; i < arr.length; i++) {
    str = str + `...${arr[i]}°C in ${count} days`;
    count++;
  }
  console.log(str);
};
printForecast(arr);

*/
```

HTML & CSS

HTML (HyperText Markup Language) is the standard markup language used to create web pages. It structures content on the web by defining elements like headings, paragraphs, links, images, tables, forms, etc. Here's a quick example of basic HTML structure:

HTML Page Structure

The basic structure of an HTML page is shown below. It contains the essential building-block elements (i.e. doctype declaration, HTML, head, title, and body elements) upon which all web pages are created.



- <!DOCTYPE html> – This is the document type declaration, not a tag. It declares that the document is an HTML5 document.
- <html> – This is called the HTML root element. All other elements are contained within it.
- <head> – The head tag contains the "behind the scenes" elements for a webpage. Elements within the head aren't visible on the front end of a webpage. Typical elements inside the <head> include:
    - <title>: Defines the title displayed on the browser tab.
    - <meta>: Provides information like the character set or viewport settings.
    - <link>: Links external stylesheets or resources.
    - <style>: Embeds internal CSS styles.
    - <script>: Embeds JavaScript for functionality.
- <title> – The title is what is displayed on the top of your browser when you visit a website and contains the title of the webpage that you are viewing.
- <h2> – The <h2> tag is a second-level heading tag.
- <p>– The <p> tag represents a paragraph of text.
- <body> – The body tag is used to enclose all the visible content of a webpage. In other words, **the body content is what the browser will show on the front end**.

HTML Attributes are special words **used within the opening tag of an HTML element**. They provide additional information about HTML elements. HTML attributes are used to configure and adjust the element's behavior, appearance, or functionality in a variety of ways.

Each attribute has a name and a value, formatted as **name="value"**. Attributes tell the browser how to render the element or how it should behave during user interactions.

Syntax:

<tagname attribute_name = "attribute_value"> content... </tagname>

Examples of HTML Attributes

Below is a categorized list of commonly used HTML attributes:

1. Global Attributes

These attributes can be used on almost all HTML elements:

class: Specifies one or more class names for an element.

id: Specifies a unique id for an element.
Hello World

style: Specifies inline CSS styles.
<h1 style="color: blue;">Hello</h1>

title: Provides additional information (tooltip).

<abbr title="HyperText Markup Language">HTML</abbr>

data-*: Custom data attributes.

2. Form-Related Attributes

type: Specifies the type of input.
<input type="text" />

name: Specifies the name of the input field.
<input name="username" />

value: Specifies the default value of an input.
<input value="John" />

placeholder: Specifies a short hint for an input field.
<input placeholder="Enter your name" />

disabled: Disables an input field.
<input disabled />

3. Link and Anchor Attributes

href: Specifies the URL of the link.

Visit

target: Specifies where to open the link.
New Tab

rel: Specifies the relationship between the current and linked documents.

Secure Link

4. Image Attributes

src: Specifies the source of the image.

alt: Provides alternative text for the image.

width and height: Specify the dimensions of the image.

5. Table Attributes

colspan: Specifies the number of columns a cell should span.
<td colspan="2">Merged Cell</td>

rowspan: Specifies the number of rows a cell should span.

<td rowspan="3">Merged Row</td>

6. Media Attributes

controls: Adds play, pause, and volume controls to media.

<video src="video.mp4" controls></video>

autoplay: Starts media playback automatically.

<audio src="audio.mp3" autoplay></audio>

7. Boolean Attributes

Boolean attributes are either present or absent. If present, they are true:

checked: Indicates a default selection in checkboxes or radio buttons.

<input type="checkbox" checked />

readonly: Makes input fields read-only.
<input type="text" value="Read Only" readonly

Classes and ids

Classes and IDs are used to identify and style elements in HTML. Both are key tools in CSS and JavaScript to apply styles or functionality to specific elements.

1. Classes

Definition:

- A **class** is a reusable identifier that can be applied to multiple elements.
- It is specified using the class attribute.

HTML Syntax:

```
<div class="example">This is a class</div>

<p class="example">This is another element with the same class</p>
```

CSS Styling:

```
.example {

   color: blue;

   font-size: 18px;

}
```

Multiple elements can share the same class.

An element can have multiple classes, separated by spaces:

```
<div class="class1 class2 class3">Multiple classes</div>
```

2. IDs

Definition:

- An **ID** is a unique identifier that is applied to a single element.
- It is specified using the id attribute.
- IDs must be unique within a page. Two elements cannot share the same ID.
- IDs are used for single, unique elements like headers, footers, or specific containers.

Usage:

HTML Syntax:

```
<div id="uniqueElement">This is an ID</div>
```

CSS Styling:

```
#uniqueElement {

   color: red;

   font-size: 20px;

     }
```

Comparison of Classes and IDs

Feature	Classes	IDs
Uniqueness	Can be reused on multiple elements.	Must be unique in the document.
Selector in CSS	`.classname`	`#idname`
JavaScript Access	`document.querySelector('.class')`	`document.getElementById('id')`
Use Case	Group styling or behavior for multiple elements.	Targeting a single, specific element.

Ex

```
<!DOCTYPE html>
<html>
<head>
    <style>
        .button {
            background-color: lightblue;
            padding: 10px;
            border: none;
            cursor: pointer;
        }
        #submitButton {
            background-color: green;
            color: white;
        }
    </style>
```

```
</head>

<body>

    <button class="button">Button 1</button>

    <button class="button" id="submitButton">Submit</button>

</body>

</html>
```

<form> Tag

Definition:

- <form> is a **semantic container** specifically designed to collect and submit user input data to a server or process it client-side.
- It provides **built-in functionality** for form submission and validation.

Features:

- Contains form elements like <input>, <textarea>, <button>, <select>, etc.
- Has built-in attributes for action, method, and validation.
- Can be submitted to a server or processed with JavaScript.

```
<form action="/submit" method="POST">

    <label for="name">Name:</label>

    <input type="text" id="name" name="name" required />

    <button type="submit">Submit</button>

</form>
```

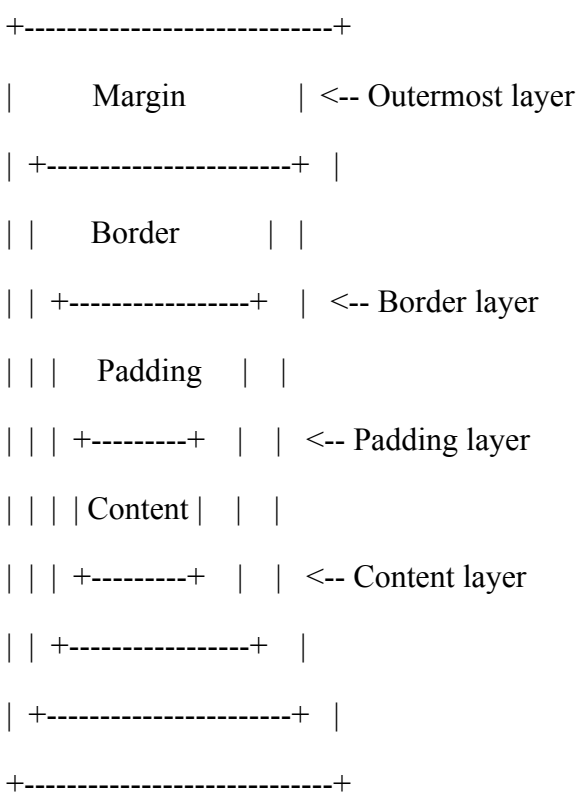
Box Model Structure

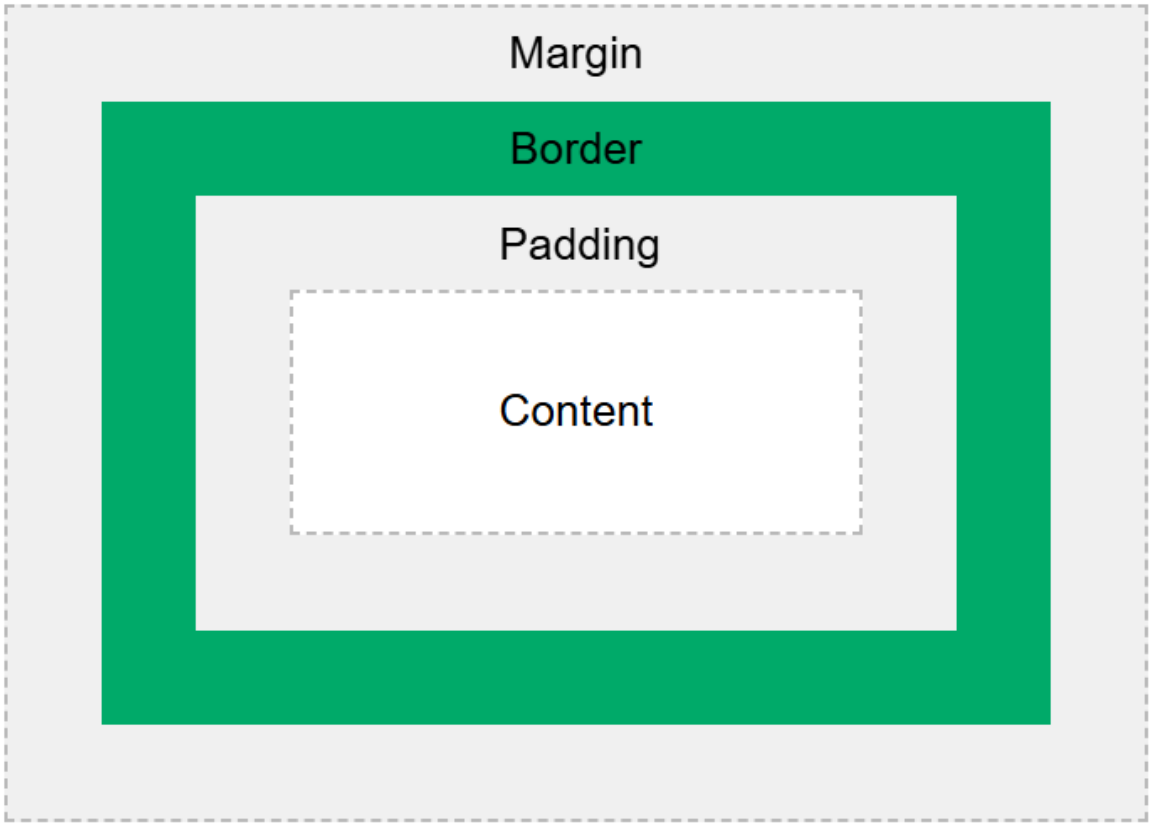
The CSS box model is made up of the following components, from the innermost to the outermost layer:

1. **Content**:
    - This is the actual content of the element, such as text or images.
    - The size of the content is determined by the width and height properties.
2. **Padding**:

- Padding is the space between the content and the border.
- It is transparent and is used to provide spacing inside the element.
- You can set padding for all sides or individually for each side using: padding-top, padding-right, padding-bottom, and padding-left.

3. **Border**:
   - The border wraps around the padding (if any) and content.
   - Borders have thickness and color, which you can control using border-width, border-style, and border-color.
   - You can apply borders individually to each side using: border-top, border-right, border-bottom, and border-left.
4. **Margin**:
   - The margin is the outermost space surrounding the border.
   - It is used to create space between the element and its surrounding elements.
   - Margins are transparent and can be controlled with margin-top, margin-right, margin-bottom, and margin-left.





What is DOM

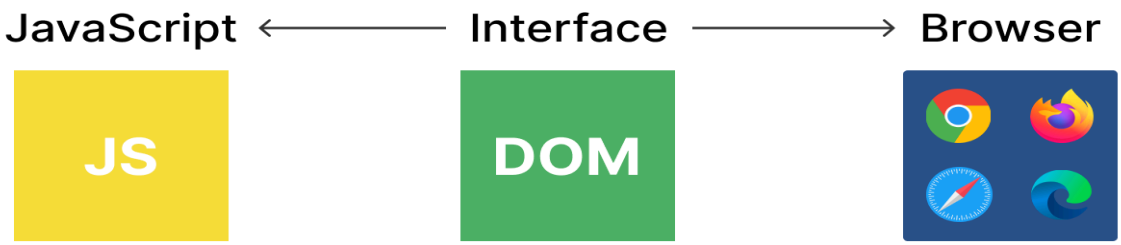
Document object model is a **structured representation of html elements** .
Allows javascript to access **html elements and styles to manipulate them**.

DOM is not part of JavaScript and it is a part of web apis.

A **Web API** refers to a **set of browser-provided interfaces** that allow developers to perform specific tasks using JavaScript. These APIs provide functionalities like making HTTP requests, manipulating the DOM, handling user interactions, or working with multimedia.

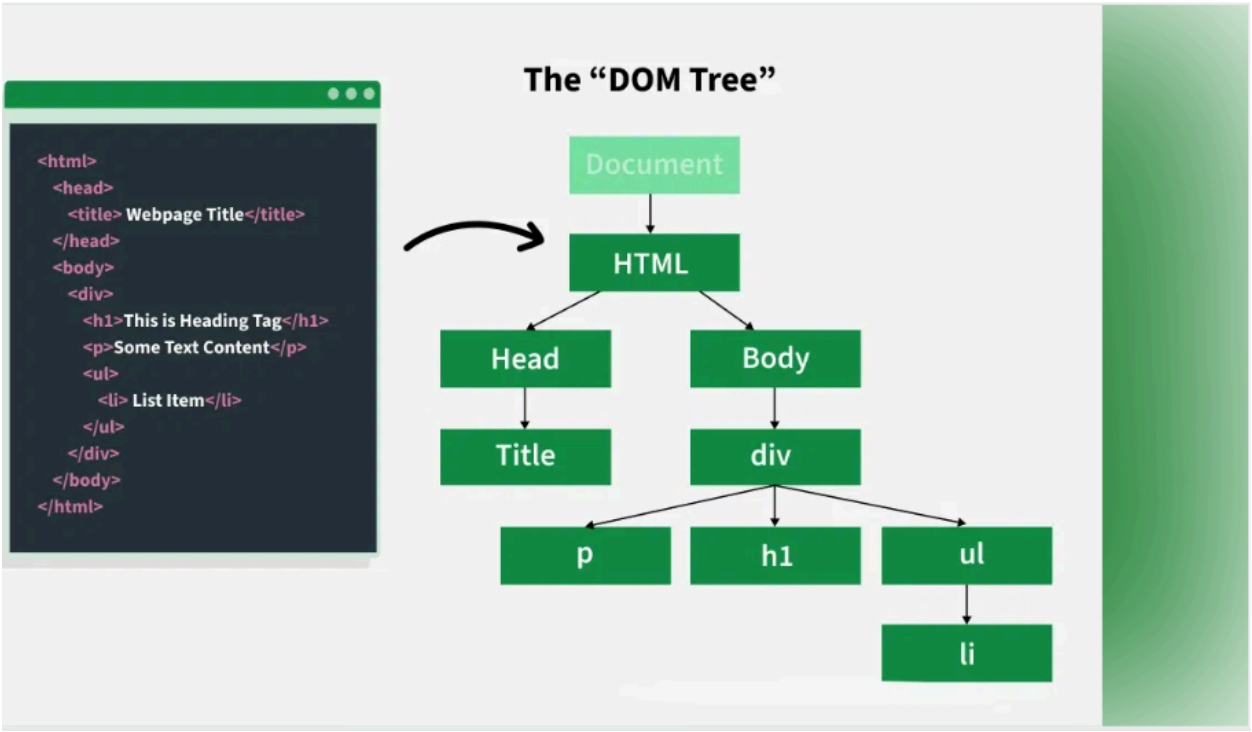
Types of Web APIs:

1. **DOM API:** Access and manipulate the document (e.g., document.getElementById).
2. **Fetch API:** Make HTTP requests to servers.
3. **Canvas API:** Draw graphics on a webpage.
4. **Geolocation API:** Get the user's geographic location.
5. **Storage API:** Store data in the browser (e.g., localStorage and sessionStorage).



DOM is created automatically as soon as the browser loads the html page .

Dom structure:



Why is DOM Required?

The DOM is essential because

- **Dynamic Content Updates**: Without reloading the page, the DOM allows content updates (e.g., form validation, AJAX responses).
- **User Interaction**: It makes your webpage interactive (e.g., responding to button clicks, form submissions).
- **Flexibility**: Developers can add, modify, or remove elements and styles in real-time.

- **Cross-Platform Compatibility**: It provides a standard way for scripts to interact with web documents, ensuring browser compatibility.

How does the DOM Works?

The DOM connects your webpage to JavaScript, allowing you to:

- Access elements (like finding an <h1> tag).
- Modify content (like changing the text of a <p> tag).
- React to events (like a button click).
- Create or remove elements dynamically.

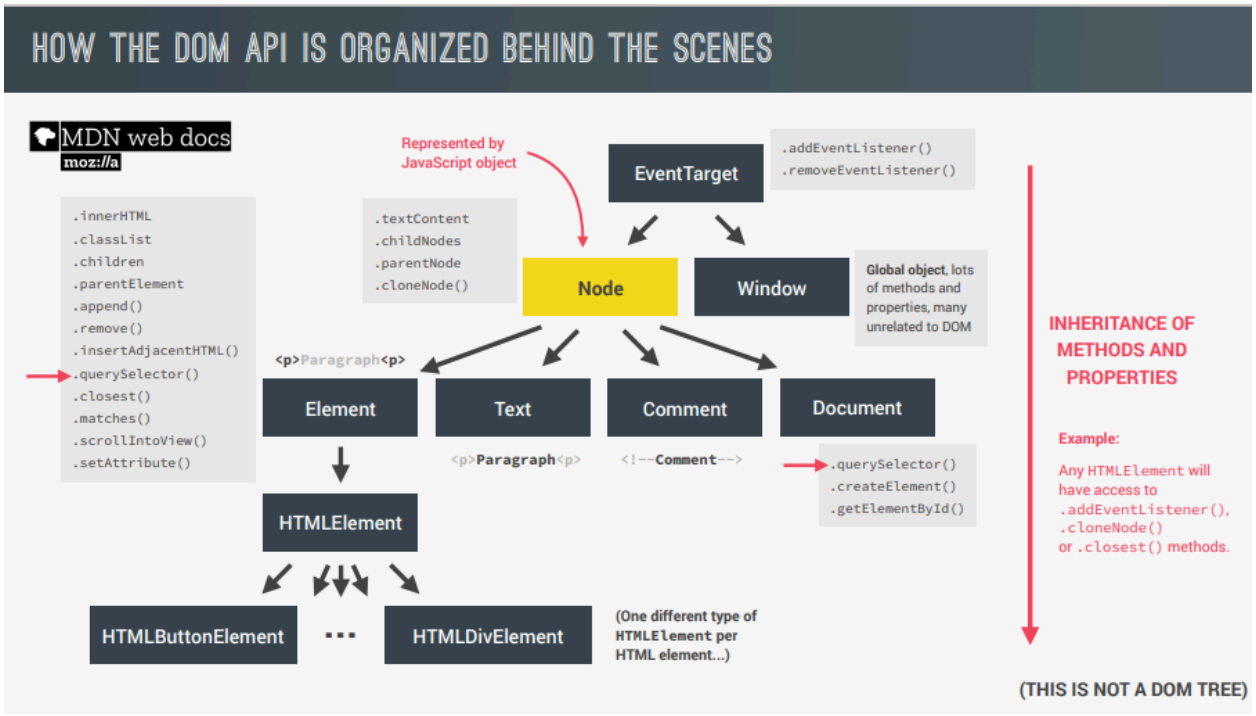
Properties of the DOM

- **Node-Based**: Everything in the DOM is represented as a node (e.g., element nodes, text nodes, attribute nodes).
- **Hierarchical:** The DOM has a parent-child relationship, forming a tree structure.
- **Live:** Changes made to the DOM using JavaScript are immediately reflected on the web page.
- **Platform-Independent:** It works across different platforms, browsers, and programming languages.

Commonly Used DOM Methods

Methods	Description
getElementById(id)	Selects an element by its ID.
getElementsByClassName(class)	Selects all elements with a given class.

querySelector(selector)	Selects the first matching element.
querySelectorAll(selector)	Selects all matching elements.
createElement(tag)	Creates a new HTML element.
appendChild(node)	Adds a child node to an element.
remove()	Removes an element from the DOM.
addEventListener(event, fn)	Attaches an event handler to an element.



Ex:

```
<!DOCTYPE html>
<html>
<head>
    <title>DOM Example</title>
</head>
<body>
    <h1 id="title">Hello, World!</h1>
    <button onclick="changeTitle()">Click Me</button>
    <script>
        function changeTitle() {
            // Access the DOM element
            const titleElement = document.getElementById("title");
```

```
        // Modify the content
        titleElement.textContent = "DOM Manipulation in Action!";
    }
</script>
</body>
</html>
```

When you click the button, JavaScript accesses the <h1> element through the DOM and changes its text content.

1) **. Select by Class (.)**
   - **CSS Selector**: Use a **dot (.)** to select elements with a specific class.
   - **JavaScript Method**: You can use document.querySelector() or document.querySelectorAll().
   - Or using **getElementsByClassName(class)**

     Example:

     ```
     <div class="box"></div>
     <div class="box"></div>
     ```

     JavaScript:

     ```
     // Select the first element with class 'box'
     const box = document.querySelector(".box");//getElementsByClassName('box');
     // Select all elements with class 'box'
     const boxes = document.querySelectorAll(".box");
     ```

   - **Explanation**:
     - The . indicates that you're selecting by class.
     - querySelector() returns the first matching element.
     - querySelectorAll() returns a **NodeList** of all matching elements.
     - **getElementsByClassName**: Returns a live HTMLCollection of elements matching the class name.

---

2. Select by ID (#)

- **CSS Selector**: Use a **hash symbol (#)** to select an element with a specific id.
- **JavaScript Method**: Use document.getElementById() or document.querySelector('#id').

  Example:

  JavaScript:

  // Select the element with id 'uniqueElement'

  const element = document.getElementById("uniqueElement");

  // You can also use querySelector (works for id as well)

  const elementById = document.querySelector("#uniqueElement");

- **Explanation**:
  - The # indicates that you're selecting by id.
  - getElementById() is a fast, specific method for selecting by id.
  - querySelector() can also be used with an id, just like in CSS.

---

3. Select by Element (Tag Name)

- **CSS Selector**: Use the **tag name** itself (e.g., div, p, h1, etc.) to select all elements of that type.
- **JavaScript Method**: Use document.getElementsByTagName() or document.querySelectorAll().

  Example:

  <h1></h1>

  JavaScript:

  // Select all elements

  const divs = document.getElementsByTagName("div");

  // Select all elements

  const paragraphs = document.querySelectorAll("p");

- **Explanation**:
    - No symbol is used for selecting by tag name.
    - getElementsByTagName() returns all elements of the given tag name.
    - querySelectorAll() can also be used to select elements by tag name.

—---

1) **document.querySelector('.score').textContent = score;:**
- **Purpose**: This sets the **text content** of an HTML element with the class .score to the value of the score variable.
- **Usage**: This expression is typically used to **display** or **update** the content of an HTML element (like a paragraph or div).

2) **document.querySelector('.guess').value:**
- **Purpose**: This retrieves the **current value** of an HTML element with the class .guess (which is likely an input field).
- **Usage**: This expression is typically used to get the value that a user has entered into an input field.

3) **document.querySelector('.check').addEventListener('click', function () {})** is an example of adding an **event listener** to an HTML element. It is used to trigger a function when a specific event occurs, such as a button click. In this case, it's listening for the **click** event on an element with the class .check.

Breakdown of the Code:

1. **document.querySelector('.check')**:
    - This part selects the first element in the document that has the class check. The querySelector method allows you to find HTML elements using CSS selectors.
    - The element could be a button, div, or any other HTML element that has the class .check.
2. **.addEventListener('click', function () {})**:
    - **addEventListener** is a method that attaches an event handler (listener) to an element.
    - The event we're listening for is the **click** event. When the user clicks the .check element (which is typically a button), the function inside the event listener will be triggered.
    - The function inside addEventListener is executed when the event occurs. It can contain any JavaScript code that you want to run when the event happens.

Project 1: Project Overview: Guess-My-Number

This is a number-guessing game built with JavaScript where the player tries to guess a secret number between 1 and 20.

Features:

1. Secret Number Generation: Each time the game starts, a random number between 1 and 20 is generated.
2. Player Input: The player enters a guess in an input field.
3. Feedback:
    - If the player enters a guess, the game checks if the guess is correct.
    - If the guess is too high or too low, the game informs the player and decreases the score.
    - If the player guesses correctly, the game congratulates them, and the number display is updated. It also changes the background color to green and increases the number's size.
    - If the guess is incorrect and the score reaches 0, the game ends with a message saying "You lost the game."
4. High Score: The game tracks the highest score achieved. If the player beats their highscore, it updates.
5. Reset Option: A button allows the player to restart the game, resetting the score, secret number, and UI elements to their initial state.

Key Concepts:

- DOM Manipulation: The game updates various HTML elements (like the score, secret number, and message) dynamically.
- **Event Listeners:** The game listens for user actions (clicking the "check" or "again" buttons).
- Game Logic: Conditional statements manage the flow of the game, such as checking if the player's guess is correct or if they've lost.

Code:

```
'use strict'; // Enabling strict mode to enforce stricter parsing and error handling in JavaScript


let score = 20; // Initializes the score to 20. This is the player's starting score.

let highscore = 0; // Initializes the high score to 0. This tracks the highest score achieved.

let secretNumber = Math.floor(Math.random() * 20) + 1; // Generates a random secret number between 1 and 20.



// Using message function to reduce code complexity, as we use the same functionality multiple times.
```

```
const displayMessage = function (message) {

  document.querySelector('.message').textContent = message; // Sets the text content of the element with class 'message' to the provided message.

};



// Adds an event listener to the 'check' button that runs a function when clicked.

document.querySelector('.check').addEventListener('click', function () {

  const guessNumber = Number(document.querySelector('.guess').value); // Converts the user's guess input to a number.

  console.log(guessNumber, typeof guessNumber); // Logs the guess number and its type for debugging purposes.



  if (!guessNumber) {

    // If the user did not enter a number (guess is falsy)

    displayMessage('⛔ No number!'); // Calls the displayMessage function to show 'No number' message.

  } else if (secretNumber === guessNumber) {

    // If the guessed number matches the secret number

    displayMessage('✅ Correct Number and 😎 You win the game!'); // Displays the success message.

    document.querySelector('.number').textContent = secretNumber; // Updates the displayed number to the secret number.

    document.querySelector('body').style.backgroundColor = '#60b347'; // Changes background color to green (indicating success).

    document.querySelector('.number').style.width = '30rem'; // Increases the width of the number display to emphasize the correct guess.



    // Checks if the current score is greater than the highscore, if so, updates the highscore.
```

```
    if (score > highscore) {
      highscore = score; // Sets the highscore to the current score.
      document.querySelector('.highscore').textContent = highscore; // Updates the displayed highscore.
    }
  } else if (guessNumber !== secretNumber) {
    // If the guessed number does not match the secret number
    if (score > 1) {
      // If the score is greater than 1, continue the game.
      displayMessage(
        guessNumber > secretNumber ? '📈 Too high!' : '📉 Too low!'
      ); // Displays 'Too high' or 'Too low' based on the guess.
      score--; // Decreases the score by 1 as the guess was incorrect.
      document.querySelector('.score').textContent = score; // Updates the displayed score.
    } else {
      // If the score reaches 0, the game is over.
      displayMessage('💥 You lost the game!'); // Displays the game over message.
      document.querySelector('.score').textContent = 0; // Sets the score to 0.
    }
  }
  // The following code has been refactored and commented out to avoid redundancy.
  // These checks are no longer necessary since they are incorporated in the ternary condition above.
  /*else if (guessNumber > secretNumber) {
```

```
    if (score > 1) {
      document.querySelector('.message').textContent = 'high value';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = 'you lost the game';
      document.querySelector('.score').textContent = 0;
    }
  } else if (guessNumber < secretNumber) {
    if (score > 1) {
      document.querySelector('.message').textContent = 'too low';
      score--;
      document.querySelector('.score').textContent = score;
    } else {
      document.querySelector('.message').textContent = 'you lost the game';
      document.querySelector('.score').textContent = 0;
    }
  }*/
});


// Adds an event listener to the 'again' button that resets the game when clicked.
document.querySelector('.again').addEventListener('click', function () {
  score = 20; // Resets the score to 20.
  secretNumber = Math.floor(Math.random() * 20) + 1; // Generates a new random secret number between 1 and 20.
```

```
  displayMessage('Start guessing...'); // Resets the message to prompt the user to start guessing.

  document.querySelector('.score').textContent = score; // Updates the score display.

  document.querySelector('.number').textContent = '?'; // Hides the secret number by displaying '?'.

  document.querySelector('.guess').value = ''; // Clears the input field where the user enters their guess.

  document.querySelector('body').style.backgroundColor = '#222'; // Resets the background color to the initial dark color.

  document.querySelector('.number').style.width = '15rem'; // Resets the number display width to the initial size.

});
```

Events

What are Events in JavaScript?

In JavaScript, **events are actions or occurrences that happen in the browser**, **which the JavaScript code can respond to**. Examples of events include clicking a button, moving the mouse, typing on the keyboard, resizing the browser window, or loading a webpage.

JavaScript uses event listeners to listen for these events and execute code when they occur.

How Events Work

1. Event Target: The element on which the event occurs (e.g., a button, link, or the window itself).
2. Event Type: The type of event (e.g., click, mouseover, keydown).
3. Event Listener: A function that is executed when the event occurs.

Usage of Events

JavaScript events are widely used to make web applications interactive and dynamic. Some common applications include:

1. Form Validation: Validating input fields when a user submits a form.

2. Dynamic UI Updates: Showing/hiding elements, sliders, modals, or tooltips.
3. User Interaction Tracking: Logging clicks, scrolling, or keyboard input for analytics.
4. Asynchronous Operations: Loading content dynamically without refreshing the page (e.g., with AJAX or Fetch API).
5. Animations and Transitions: Starting or stopping animations on hover or click.
6. Real-time Feedback: Showing error messages, autocomplete suggestions, or notifications.

Types of Events

Here are some commonly used JavaScript events:

Mouse Events:(Mouse events in JavaScript are triggered by user interactions with a pointing device, like a mouse or trackpad. These events enable developers to respond to actions such as clicking, hovering, dragging, or scrolling.)

- click: When an element is clicked.
- dblclick: When an element is double-clicked.
- mouseover: When the mouse pointer enters an element.
- mouseout: When the mouse pointer leaves an element.
- mousedown / mouseup: When a mouse button is pressed or released.
- The mouseenter event is triggered only when the mouse pointer enters the boundaries of the element it's attached to.When the mouse pointer enters an element (does not trigger for child elements).

2) Keyboard Events:(Keyboard events in JavaScript are triggered when a user interacts with a keyboard. These events allow developers to detect when a key is pressed or released and respond accordingly, enabling functionalities like form validation, navigation, or triggering custom actions.)

keydown:

- Triggered when a key is pressed down.
- Fires continuously if the key is held down.
- Captures all keys, including special keys like Shift, Ctrl, and Arrow keys.

keyup:

- Triggered when a key is released after being pressed.

keypress (Deprecated):

Triggered when a key produces a character value.

It’s deprecated and should be replaced with keydown or keyup.

3) Form Events(Form events in JavaScript are triggered by user interactions with form elements (e.g., input fields, checkboxes, or the form itself). These events are essential for validating user input, capturing data, and enhancing the user experience.**):**

- submit: When a form is submitted.
- focus / blur: When an element gains or loses focus.

4) Window Events(Window events are triggered by interactions with the browser's window or the document as a whole. These events allow developers to handle scenarios such as page loading, resizing, scrolling, or unloading.**):**

- load: When the webpage has finished loading.
- resize: When the browser window is resized.
- scroll: When the user scrolls the page.

5) Touch Events (for mobile): Touch events in JavaScript are triggered when a user interacts with a touch-sensitive device, such as a smartphone or tablet. These events allow developers to handle touch gestures and interactions, like tapping, swiping, or pinching.

- touchstart: When a finger touches the screen.
- touchend: When a finger is removed from the screen.

Project 2: Project Overview: Modal Window

A modal window is a user interface component that displays content in a layer above the main application, requiring user interaction before returning to the main page. This project focuses on implementing a Modal Window using HTML, CSS, and JavaScript.

This code implements a Modal Window that:

1. Opens when a user clicks any of the buttons with the class show-modal.
2. Closes when:
   - Clicking the "Close" button.
   - Clicking outside the modal (on the overlay).
   - Pressing the Escape key on the keyboard.

```
'use strict';

// Enables strict mode, which helps catch common JavaScript errors, such as undeclared variables, and enforces better coding practices.



const modal = document.querySelector('.modal');
```

```
// Selects the first HTML element with the class 'modal' and stores it in the variable `modal`.


const overlay = document.querySelector('.overlay');

// Selects the first HTML element with the class 'overlay' and stores it in the variable `overlay`.


const btnCloseModal = document.querySelector('.close-modal');

// Selects the first HTML element with the class 'close-modal' (usually the close button inside the modal) and stores it in `btnCloseModal`.


const btnsOpenModal = document.querySelectorAll('.show-modal');

// Selects all elements with the class 'show-modal' (likely buttons that trigger the modal) and stores them in a NodeList called `btnsOpenModal`.


const openModal = function () {

  // Defines a function named `openModal` that opens the modal.

  modal.classList.remove('hidden');

  // Removes the 'hidden' class from the `modal` element to make it visible.

  overlay.classList.remove('hidden');

  // Removes the 'hidden' class from the `overlay` element to make the background overlay visible.

};


const closeModal = function () {

  // Defines a function named `closeModal` that closes the modal.

  modal.classList.add('hidden');
```

```
  // Adds the 'hidden' class to the `modal` element to hide it.

  overlay.classList.add('hidden');

  // Adds the 'hidden' class to the `overlay` element to hide the background overlay.

};



for (let i = 0; i < btnsOpenModal.length; i++)

  // Iterates over all the buttons in the `btnsOpenModal` NodeList using a for loop.

  btnsOpenModal[i].addEventListener('click', openModal);

// Adds a `click` event listener to each button. When clicked, the `openModal`
function is executed to open the modal.



btnCloseModal.addEventListener('click', closeModal);

// Adds a `click` event listener to the `btnCloseModal` element. When clicked, the
`closeModal` function is executed to close the modal.



overlay.addEventListener('click', closeModal);

// Adds a `click` event listener to the `overlay` element. When clicked, the
`closeModal` function is executed to close the modal.



document.addEventListener('keydown', function (e) {

  // Adds a `keydown` event listener to the entire document. This listens for any key
press.

  // `e` is the event object that contains details about the event (e.g., the key pressed).



  // console.log(e.key);

  // Uncomment this line to log the key pressed to the console for debugging
purposes.
```

```
  if (e.key === 'Escape' && !modal.classList.contains('hidden')) {
    // Checks if the key pressed is 'Escape' AND the modal is not currently hidden.
    closeModal();
    // If the conditions are true, call the `closeModal` function to close the modal.
  }
});
```

The overlay class is typically used to create a semi-transparent layer that appears over the content of a webpage. The overlay is often used in combination with modals, lightboxes, or other popups to dim the background and highlight the modal or popup content.

The hidden class is commonly used to hide elements. This class likely has CSS that sets the element's display to none, making it invisible and not part of the layout. For example:

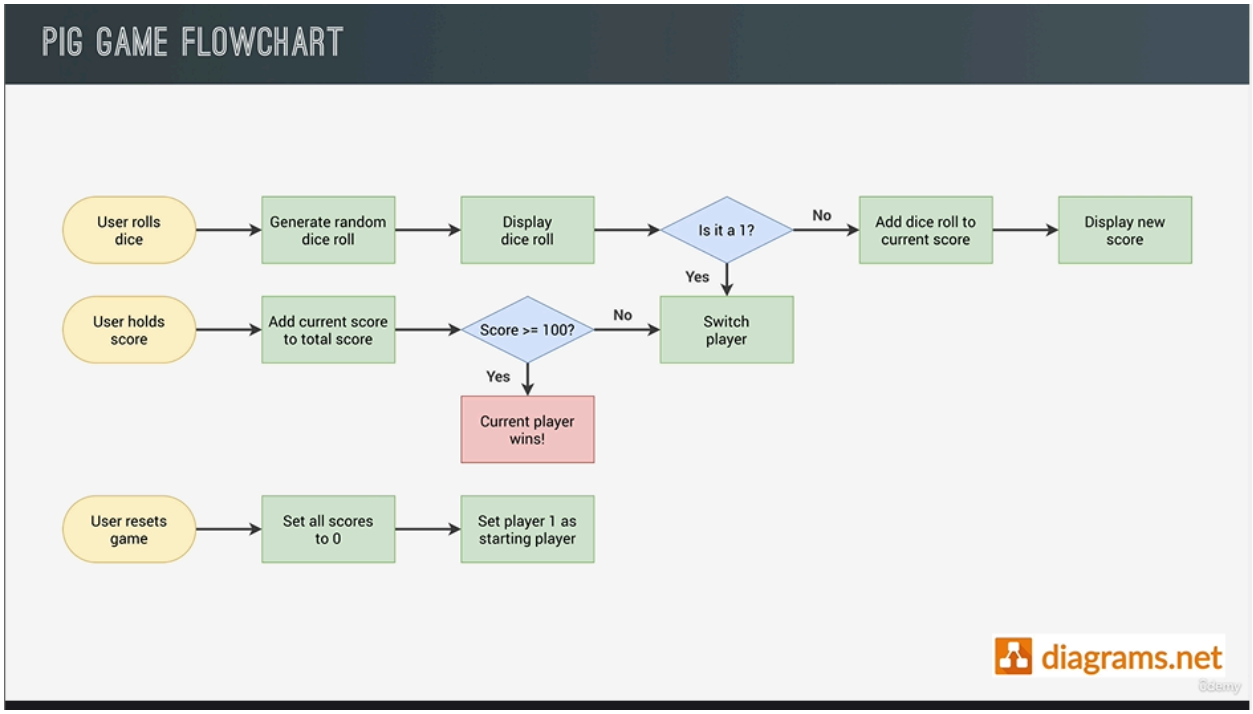
```
.hidden {
  display: none;
}
```

When the hidden class is removed, the overlay becomes visible.

If you want to **show** the element (i.e., make it visible), you can use:

```
.hidden {
  display: block; /* or other appropriate display value, like inline or flex */
}
```

Project 3: Project Overview: Pig Game



About Game: In this game, User Interface (UI) contains user/player that can do three things, they are as follows:

There will be two players in this game. At the start of the game **Player 1** will be the **CurrentPlayer** and **Player 2** will be the **in-active** one.

1. **Roll the dice:** The current player has to roll the dice and then a random number will be generated. If the current player gets **any number other than 1 on the dice** then that number will be added to the current score (**initially the current score will be 0**) and then the new score will be displayed under the Current **Score** section. **Note:** If the current player gets **1 on the dice** then the players will be switched i.e. the current player will become **in-active** and vice-versa.
2. **Hold:** If the current player clicks on **HOLD**, then the Current Score will be added to the **Total Score**. When the active player clicks the Hold button then the **total score is evaluated.** If the **Total Score >= 100** then the current player wins else the players are switched.
3. **Reset:** All the scores are set to 0 and **Player 1** is set as the starting player (current player)

Advanced DOM and Events

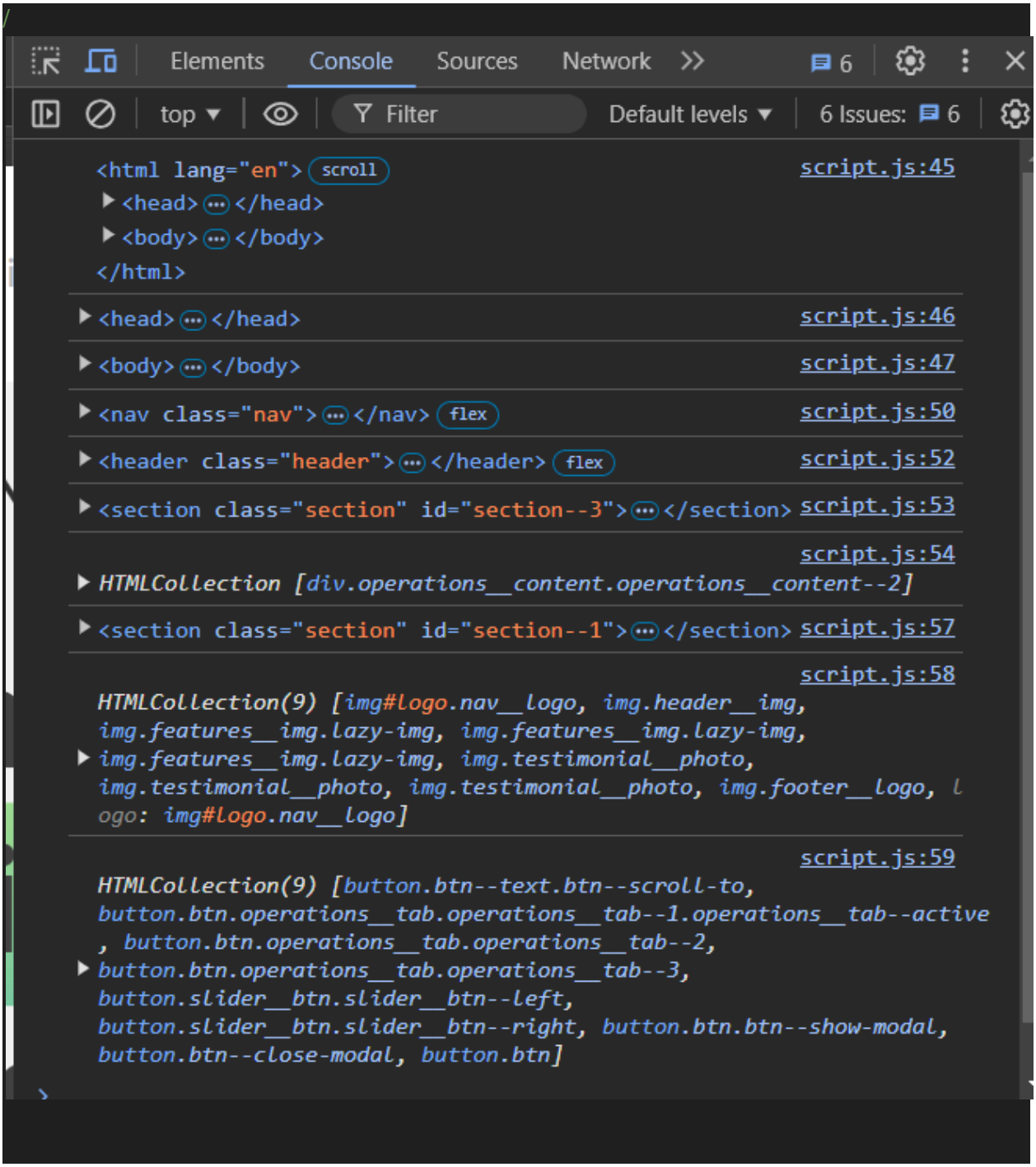
Bankist Marketing App:

i) selecting,creating and deleting elements

Select

```
/select

console.log(document.documentElement);

console.log(document.head);

console.log(document.body);

let doc = document.querySelector('.nav');

console.log(doc);

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

console.log(header);

console.log(document.getElementById('section--3'));

console.log(

  document.getElementsByClassName('operations__content operations__content--2')

);

console.log(document.querySelector('#section--1'));

console.log(document.getElementsByTagName('img'));
```

```
console.log(document.getElementsByTagName('button')); //all buttons
```



Create and insert

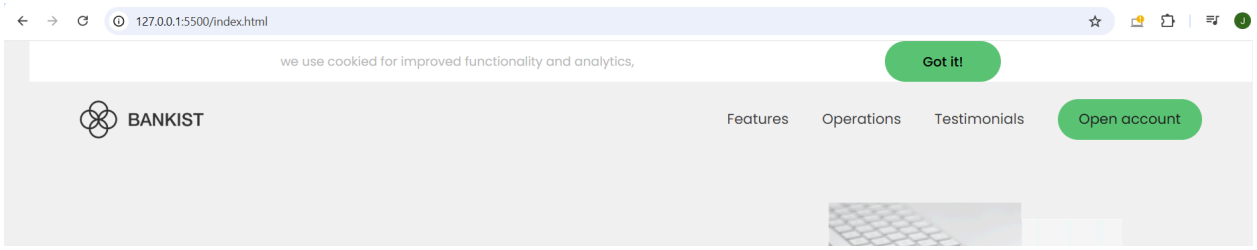
```
//creating and inserting elements


//1.The insertAdjacentHTML() method inserts HTML code into a specified position.
// Create a new element
const message = document.createElement('div'); // Use "div" or any standard HTML element
message.classList.add('cookie-message'); // Add a class to the element
// Add content to the element
message.innerHTML ='we use cookies for improved functionality and analytics,<button
class="btn btn--close-cokkie"> Got it!</button>';
// Optionally, append the element to the DOM (e.g., body or another parent element)
header.prepend(message);
//header.before(message);
//header.after(message);
```

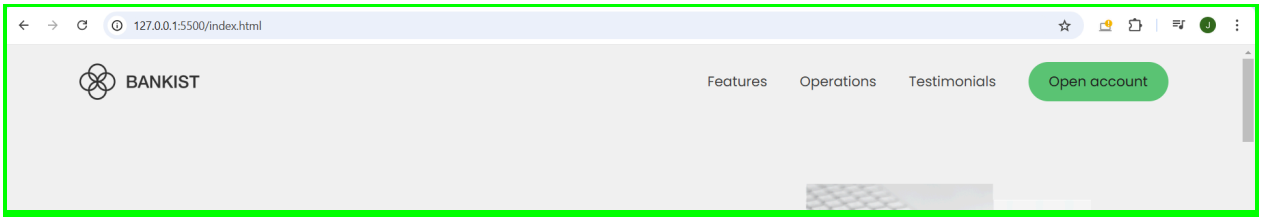
//deleting

```
//deleting

document

 .querySelector('.btn--close-cokkie')

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

```
  message.remove();

  //message.parentElement.removeChild(message);

});
```

If we click on “Got it” , it will vanish.





Styles , Attributes and Classes

```
// Styles, Attributes and Classes

// Styles

message.style.backgroundColor = '#37383d';

message.style.width = '120%';

console.log(message.style.color); //empty

console.log(message.style.backgroundColor); //rgb(55, 56, 61)

console.log(getComputedStyle(message).color); //rgb(187, 187, 187)

console.log(getComputedStyle(message).height); //73px

message.style.height =
```

```
  Number.parseFloat(getComputedStyle(message).height, 10) + 30 + 'px';

document.documentElement.style.setProperty('--color-primary', 'orangered');

// Attributes

const logo = document.querySelector('.nav__logo');

console.log(logo.alt); //Bankist logo

console.log(logo.className); //nav__logo

logo.alt = 'Beautiful minimalist logo';

// Non-standard

console.log(logo.designer); //undefined

console.log(logo.getAttribute('designer')); //ram

logo.setAttribute('company', 'Bankist');

console.log(logo.src); //http://127.0.0.1:5500/img/logo.png

console.log(logo.getAttribute('src')); //img/logo.png

const link = document.querySelector('.nav__link--btn');

console.log(link.href); //http://127.0.0.1:5500/index.html#

console.log(link.getAttribute('href')); //#

// Data attributes

//data attributes  are special attributes and start with data

console.log(logo.dataset.versionNumber); //3.0

// Classes

logo.classList.add('c', 'j');

// logo.classList.remove('c', 'j');

// logo.classList.toggle('c');

// logo.classList.contains('c'); // not includes
```

```
// Don't use

logo.clasName = 'jonas';
```

Smooth scrolling

Smooth scrolling in JavaScript can be implemented using the scrollIntoView, scrollTo, or scrollBy methods, combined with the behavior: "smooth" property. Here's how you can do it:

1. Using scrollIntoView

You can use the scrollIntoView method to scroll to a specific element:

```
document.getElementById("target-element").scrollIntoView({

  behavior: "smooth",

  block: "start", // Align to the top of the viewport

});
```

2. Using window.scrollTo

You can scroll to specific coordinates on the page:

```
window.scrollTo({

  top: 500, // Replace with the desired scroll position in pixels

  left: 0,

  behavior: "smooth", // Enables smooth scrolling

});
```

3. Using window.scrollBy

This scrolls by a relative amount:

```
window.scrollBy({
  top: 200, // Scroll down by 200 pixels
  left: 0,
  behavior: "smooth",
});
```

4.Smooth Scrolling with Event Listeners

For example, smooth scrolling when clicking a link:

```
<a href="#section" id="scroll-link">Go to Section</a>
<section id="section" style="margin-top: 1000px;">Target Section</section>
<script>
  document.getElementById("scroll-link").addEventListener("click", function (e) {
    e.preventDefault();
    document.getElementById("section").scrollIntoView({
      behavior: "smooth",
      block: "start",
    });
```

```
  });

</script>
```

Ex:

```
const btnScrollTo = document.querySelector('.btn--scroll-to');

const section1 = document.querySelector('#section--1');

btnScrollTo.addEventListener('click', function (e) {

 const s1coord = section1.getBoundingClientRect(); //it will vary if change browser layout change

 console.log(s1coord);

 /*

 //DOMRect {x: 0, y: 748, width: 835.2000122070312, height: 1747.5, top: 748,}

 bottom: 2495.5

 height: 1747.5

 left: 0

 right : 835.2000122070312

 top: 748

 width: 835.2000122070312

 x: 0

 y: 748

 */

 section1.scrollIntoView({ behavior: 'smooth' });
```

```
});
```

Type of Events and Event Handler

```
 //event

const h1 = document.querySelector('h1');

h1.addEventListener('mouseenter', function (e) {

 alert('mouse enter operation');

});

//or

const alertH1 = function (e) {

 alert('mouse enter operation');

};

h1.addEventListener('mouseenter', alertH1);



//removing event

setTimeout(() => h1.addEventListener('mouseenter', alertH1), 3000); //after 3 sec it will removed
```

We can use mouse events directly in HTML by adding event attributes to elements. Here's how you can do it for each event:

onclick (for click)

<button onclick="alert('Button clicked!')">Click Me</b

Bubbling and capturing

In JavaScript, bubbling and capturing are two phases of the event propagation model. This model defines how events flow through the Document Object Model (DOM) when they are triggered.

1. Event Propagation Phases

When an event occurs on an element, it goes through three phases:

a. Capturing Phase (Event Capturing)

- The event starts from the root of the DOM tree and travels down to the target element.
- This is also called the **"trickle-down" phase**.

b. Target Phase

- The event reaches the target element itself.

c. Bubbling Phase (Event Bubbling)

- After reaching the target element, the event propagates back up the DOM tree toward the root.
- This is called the **"bubble-up" phase**.

2. How It Works

Consider the following DOM structure:

```
<div id="parent">

  <button id="child">Click Me</button>

</div>
```

If you click the button (#child), the event flows as follows:

- **Capturing phase**: document → html → body → #parent → #child
- **Target phase**: The event triggers on the target element (#child).
- **Bubbling phase**: #child → #parent → body → html → document

—----------

```
  <button id="child">Click Me</button>

</div>
```

Event Bubbling:

- **Definition**: In event bubbling, **the event starts at the target element** (where the event was triggered) and propagates upward through its parent elements, eventually reaching the root element (usually the document).
- **Example**: If you click on a button inside a div, the click event will first be captured by the button, then by the div, then by its parent, and so on.
- **Usage**: By default, events in JavaScript propagate in the bubbling phase.

Example:

```
document.getElementById('btn').addEventListener('click', function() {

    alert('Button clicked!');

});

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

    alert('Parent clicked!');

});

//or

document.getElementById('child').addEventListener('click', function(event) {

    console.log('Child clicked!');

    event.stopPropagation(); // Stops the event from bubbling to the parent.

}, false);
```

```
document.getElementById('parent').addEventListener('click', function() {

    console.log('Parent clicked!');

}, false);
```

In this case, clicking the button will trigger the button's click event first, then the parent's click event, in that order.

Event Capturing (Trickling):

- **Definition**: Capturing, or trickling, is **the phase where the event starts from the root** of the DOM tree (typically document) and goes down to the target element. This happens before the event reaches the target element itself.
- **Example**: The event is first captured by the outermost element and works its way to the target element, which is the last one to capture the event.
- **Usage**: Event capturing is not the default behavior. It must be explicitly enabled by passing a third argument (with the value true) to the addEventListener method.

Example:

```
document.getElementById('parent').addEventListener('click', function() {

    alert('Parent clicked!');

}, true);

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

    alert('Button clicked!');

}, true);
```

In this case, clicking the button will trigger the parent's click event first, then the button's click event, in that order.

true: This value tells the browser to execute the event listener in the **capturing** phase. In this case, the event listener will be triggered as the event moves **down** the DOM tree (from the root to the target element).

false (or omitted): This value tells the browser to execute the event listener in the **bubbling** phase. In this case, the event listener will be triggered as the event moves **up** the DOM tree (from the target element to the root).

Page Navigation

Event Delegation

Event delegation in JavaScript is a technique **where you attach a single event listener to a parent element rather than multiple listeners to individual child elements**. This is particularly useful when dealing with dynamic elements that may be added to the DOM after the page has loaded.

event.target: This property is used to get the element that triggered the event, which is useful when handling events for multiple child elements.

HTML Structure Example:

```
<ul class="nav__links">

  <li class="nav__item"><a href="#section--1" class="nav__link">Features</a></li>

  <li class="nav__item"><a href="#section--2" class="nav__link">Operations</a></li>

  <li class="nav__item"><a href="#section--3" class="nav__link">Testimonials</a></li>

  <li class="nav__item"><a href="#" class="nav__link">Open account</a></li>

</ul>
```

JavaScript (Event Delegation):

```
/ Select the parent element (in this case, the `nav__links` container)
const navLinks = document.querySelector('.nav__links');

// Add event listener to the parent element
navLinks.addEventListener('click', function (event) {
  // Check if the clicked element is a link (i.e., has the class 'nav__link')
  if (event.target.classList.contains('nav__link')) {
    // event.target refers to the element that triggered the event
    const clickedLink = event.target;

    // Get the ID (href attribute) from the clicked link
    const id = clickedLink.getAttribute('href');

    // Select the target section based on the ID (e.g., #section--1)
    const targetSection = document.querySelector(id);

    // Scroll to the target section with smooth behavior
    targetSection.scrollIntoView({
      behavior: 'smooth',
      block: 'start'  // Ensures the section scrolls to the top of the viewport
    });

    // Optionally, you could prevent the default behavior (e.g., jumping to the section)
    event.preventDefault();
  }
```

});

Explanation of Changes:

1. **Getting the id from href**: The code retrieves the href attribute (which contains the ID of the target section like #section--1) from the clicked link.
2. **Selecting the target section**: The document.querySelector(id) selects the target section by using the ID.
3. **Smooth Scrolling**: The scrollIntoView method is used on the target section, and { behavior: 'smooth' } ensures that the page scrolls smoothly to the section. Additionally, { block: 'start' } ensures the section is aligned at the top of the viewport.

Why event.preventDefault():

- Without event.preventDefault(), the page would perform its default action, which is jumping straight to the target section. This is overridden by the smooth scrolling code when you call event.preventDefault().

Why Use Event Delegation?

Event delegation is efficient because it allows you to:

- Use a single event listener for many elements (e.g., all links inside a menu).
- Dynamically add or remove child elements without having to add new event listeners to each one.

DOM Traversing:

DOM (Document Object Model) traversing refers to navigating through and interacting with the elements of an HTML or XML document using JavaScript. You can access, manipulate, and modify the structure, content, and attributes of a document dynamically. Here are the common methods used for DOM traversing:

1. Getting Elements

- document.getElementById(id): Returns the element with the specified ID.
- document.getElementsByClassName(className): Returns a live HTMLCollection of elements with the specified class.
- document.getElementsByTagName(tagName): Returns a live HTMLCollection of elements with the specified tag name.
- document.querySelector(selector): Returns the first element that matches the specified CSS selector.
- document.querySelectorAll(selector): Returns a NodeList of all elements that match the specified CSS selector.

2. Navigating Between Elements

- parentNode: Access the parent of the current element.
- childNodes: Access all child nodes (including text and comment nodes).
- firstChild: Access the first child node.
- lastChild: Access the last child node.
- nextSibling: Access the next sibling node.
- previousSibling: Access the previous sibling node.
- firstElementChild: Access the first child element (ignores text and comment nodes).
- lastElementChild: Access the last child element (ignores text and comment nodes).
- nextElementSibling: Access the next sibling element (ignores text and comment nodes).
- previousElementSibling: Access the previous sibling element (ignores text and comment nodes).

3. Manipulating Element Content and Attributes

- textContent: Get or set the text content of an element.
- innerHTML: Get or set the HTML content of an element.
- setAttribute(attributeName, value): Set an attribute on an element.
- getAttribute(attributeName): Get the value of an attribute.
- removeAttribute(attributeName): Remove an attribute.

4. Modifying Styles

- element.style: Access the inline styles of an element (e.g., element.style.color = 'red';).

5. Event Handling

- element.addEventListener(event, callback): Adds an event listener to an element.
- element.removeEventListener(event, callback): Removes an event listener from an element.

6. Traversing with Traversal Methods

- NodeIterator: An object that can be used to traverse a DOM tree node by node.
- TreeWalker: Provides a way to traverse a DOM tree while filtering the types of nodes to traverse.

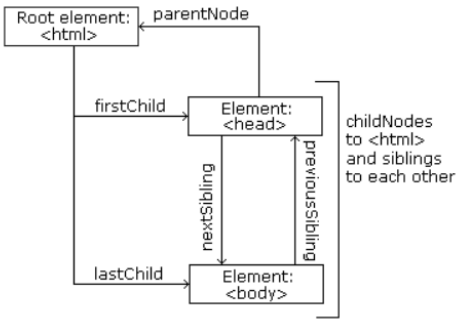
Node Relationships

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```



From the HTML above you can read:

- <html> is the root node
- <html> has no parents
- <html> is the parent of <head> and <body>
- <head> is the first child of <html>
- <body> is the last child of <html>

and:

- <head> has one child: <title>
- <title> has one child (a text node): "DOM Tutorial"
- <body> has two children: <h1> and <p>
- <h1> has one child: "DOM Lesson one"
- <p> has one child: "Hello world!"
- <h1> and <p> are siblings

Navigating Between Nodes

You can use the following node properties to navigate between nodes with JavaScript:

- parentNode
- childNodes[*nodenumber*]
- firstChild
- lastChild
- nextSibling
- previousSibling

closest()

The **closest()** method in JavaScript is used to find the nearest ancestor (including the element itself) of a given element that matches a specified selector. It starts from the current element and traverses upward through the DOM tree, looking for the first element that matches the given selector. If no match is found, it returns null.

Syntax:

```
element.closest(selector);
```

- **selector**: A string representing the selector to match against.
- **Return**: The closest ancestor element (or the element itself) that matches the selector, or null if no matching ancestor is found.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM closest() Method Example</title>
</head>
<body>
    <div class="grandparent">
        <div class="parent">
            <p class="child">I am the child</p>
        </div>
    </div>

    <script>
        const child = document.querySelector('.child');

        // closest() to find the nearest ancestor with class 'parent'
        const parentElement = child.closest('.parent');
        console.log(parentElement);
        // Output: <div class="parent">...</div>
```

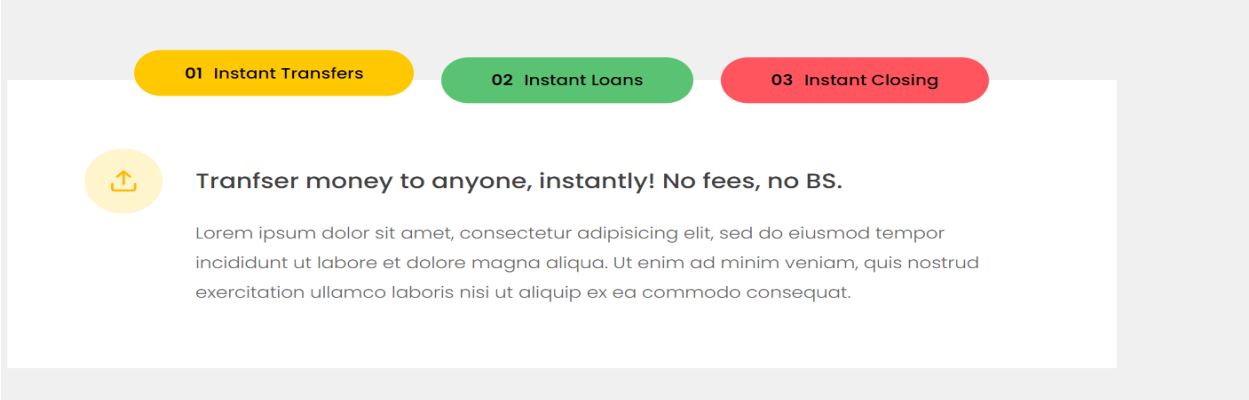
```
        // closest() to find the nearest ancestor with class 'grandparent'
        const grandparentElement = child.closest('.grandparent');
        console.log(grandparentElement);
        // Output: <div class="grandparent">...</div>

        // closest() to find an ancestor with class 'non-existent'
        const nonExistent = child.closest('.non-existent');
        console.log(nonExistent);
        // Output: null (since there's no element with class 'non-existent')
    </script>
</body>
</html>
```

Explanation:

1. **parentElement**: The closest() method starts with the .child element and searches for the closest parent element with the class .parent. It finds and logs the .parent div.
2. **grandparentElement**: Similarly, it searches for the closest parent with the class .grandparent and finds and logs the .grandparent div.
3. **nonExistent**: Since there's no element with the class .non-existent, it returns null.

Tabbed Component

A tabbed component is a UI pattern used to display content in separate sections that can be navigated by clicking on different tabs.



Passing arguments to event handler

//

Scroll event

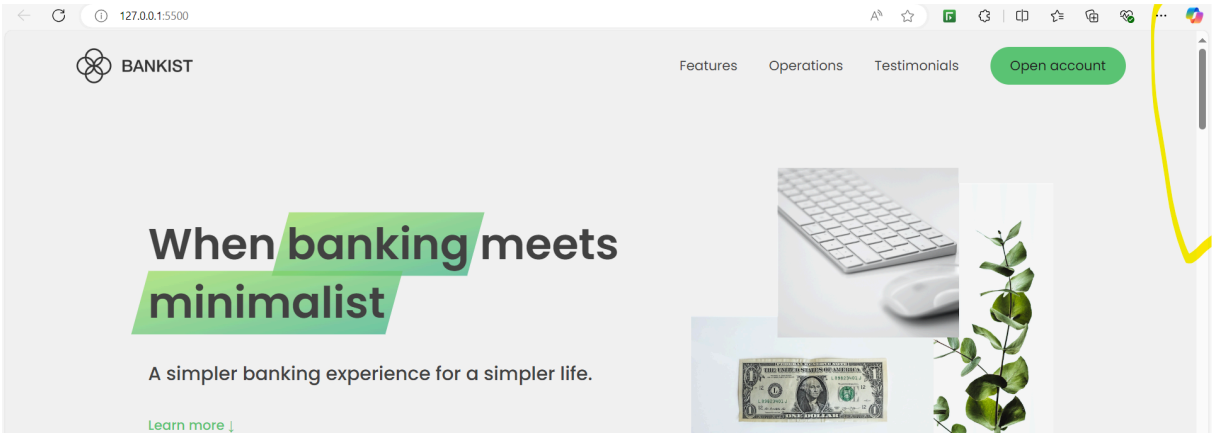
In JavaScript, the scroll event is triggered when the user scrolls an element or the entire page. It can be used to detect when a user scrolls through a document or a specific element (such as a div). You can listen for the scroll event and execute some code in response.

Here's a basic example of using the scroll event:

Example 1: Scroll event on the window

This example listens for scroll events on the entire window.

```
window.addEventListener('scroll', function() {

    console.log('You scrolled the window!');

});
```



Example 2: Scroll event on a specific element

This example listens for scroll events on a specific element (e.g., a div with id="myElement").

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

myElement.addEventListener('scroll', function() {

    console.log('You scrolled the element!');

});
```

sticky class is often used to make an element "stick" to the top or a certain position of the viewport when scrolling. This is commonly used for navigation bars, headers, or other elements that need to remain visible when the user scrolls down.

You can create a sticky effect by using the position: sticky CSS property. The element will stick to the top when you scroll past it.

```
.sticky {

    position: fixed;

    top: 0;
```

```
    width: 100%;

    z-index: 1000; /* Ensure it stays on top of other content */

    box-shadow: 0 4px 2px -2px gray; /* Optional: add a shadow for better visibility */

}
```

The Intersection Observer API:

Intersection Observer is an API that is used to detect the interaction of a target element with its ancestor element or the document viewport. For example, if we want to detect if some element is visible in the viewport we can use this API for that purpose.

or

The Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport(A viewport represents a polygonal (normally rectangular) area in computer graphics that is currently being viewed. In web browser terms, it refers to the part of the document you're viewing which is currently visible in its window (or the screen, if the document is being viewed in full screen mode). Content outside the viewport is not visible onscreen until scrolled into view.)

Historically, detecting visibility of an element, or the relative visibility of two elements in relation to each other, has been a difficult task for which solutions have been unreliable and prone to causing the browser and the sites the user is accessing to become sluggish. As the web has matured, the need for this kind of information has grown. Intersection information is needed for many reasons, such as:

- Lazy-loading of images or other content as a page is scrolled.
- Implementing "infinite scrolling" websites, where more and more content is loaded and rendered as you scroll, so that the user doesn't have to flip through pages.
- Reporting of visibility of advertisements in order to calculate ad revenues.
- Deciding whether or not to perform tasks or animation processes based on whether or not the user will see the result.

Syntax:

The IntersectionObserver is created using:

```
const observer = new IntersectionObserver(callback, options);
```

- **callback**: A function executed whenever the observed elements intersect with the root.
- **options**: An optional object to configure the observer's behavior.

The **options** parameter accepts an object with these properties:

root

- Specifies the element used as the viewport for checking visibility.
- If set to null (default), the browser's viewport is used as the root.
- Can be a specific DOM element, such as a scrollable container.

Examples:

```
root: null // Uses the browser viewport

root: document.querySelector('.scroll-container') // Uses a specific container
```

rootMargin

- A margin around the root in the format of CSS margin values (e.g., "10px", "10px 20px", "10% 5%").
- Expands or shrinks the root bounding box, affecting when the observer triggers.
- Defaults to "0px".
- Can take positive or negative values.

Examples:

```
rootMargin: '0px'       // No margin (default)

rootMargin: '10px 20px' // 10px top/bottom, 20px left/right

rootMargin: '-50px'     // Shrink the root bounding box by 50px on all sides
```

threshold

- Determines at what percentage of the target's visibility the observer's callback should be executed.
- Can be a single number (e.g., 0.5) or an array of numbers (e.g., [0, 0.5, 1]).
- Values range from 0.0 (completely invisible) to 1.0 (completely visible).

Examples:

threshold: 0.5 // Callback triggers when 50% of the target is visible

threshold: [0, 0.5, 1] // Callback triggers at 0%, 50%

Callback Function

The callback is executed whenever the visibility of the observed elements changes. It takes two arguments:

1. entries: An array of IntersectionObserverEntry objects.
2. observer: The IntersectionObserver instance.

Example:

```
const callback = (entries, observer) => {

  entries.forEach(entry => {

    console.log(entry.isIntersecting); // true if the element is in view

  });

};
```

Full Syntax Example

```
const options = {
  root: null, // Use the viewport as the root
  rootMargin: '0px', // No margin
  threshold: 0.5 // Trigger when 50% of the element is visible
};

const callback = (entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      console.log('Element is visible:', entry.target);
      // Stop observing if needed
      observer.unobserve(entry.target);
    }
```

```
  });
};

const observer = new IntersectionObserver(callback, options);

// Start observing an element
observer.observe(document.querySelector('.target'));
```

Methods

1. **observe(target)**
    - Starts observing the visibility of the target element.
    - **target**: A DOM element to observe.

      Example:

      `observer.observe(document.querySelector('.target'));`

2. **unobserve(target)**
    - Stops observing the visibility of the target element.
    - **target**: A DOM element being observed.

      Example:

      `observer.unobserve(document.querySelector('.target'));`

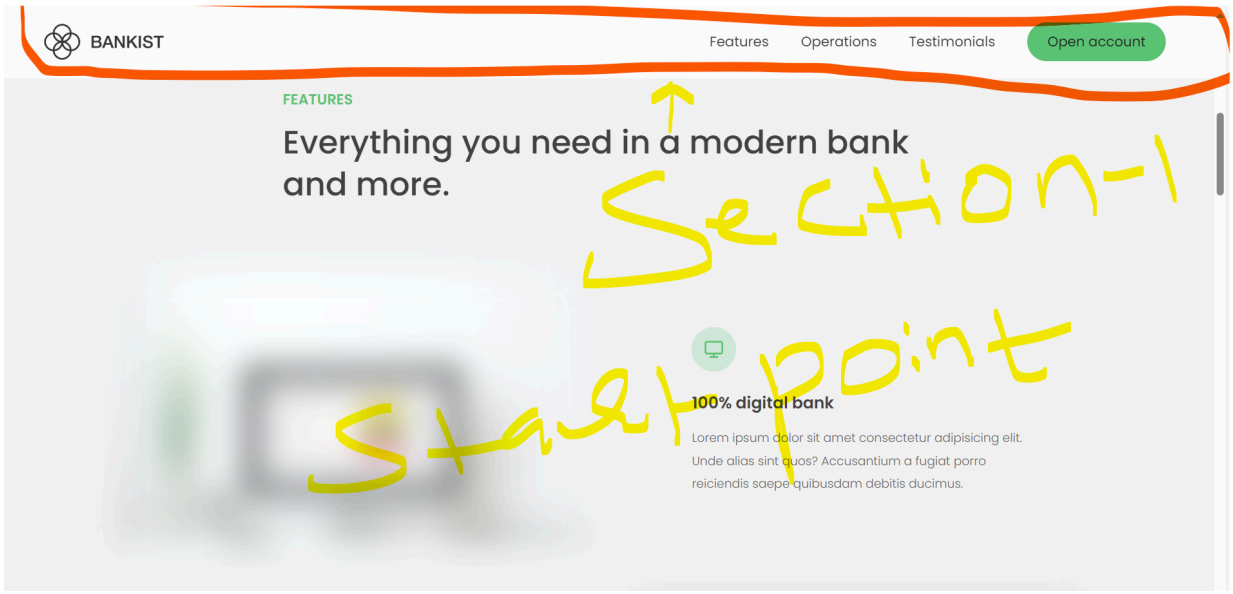
3. **disconnect()**
    - Stops all observation by the observer.

      Example:

      `observer.disconnect();`

4. **takeRecords()**
    - Returns an array of IntersectionObserverEntry objects representing all changes that haven't been delivered yet.

```
const nav = document.querySelector('.nav');

const header = document.querySelector('.header');

const stickNav = function (entries) {

  const [entry] = entries;

  console.log(entry);

  if (!entry.isIntersecting) {

    nav.classList.add('sticky');

  } else {

    nav.classList.remove('sticky');

  }

};

const headerObserver = new IntersectionObserver(stickNav, {

  root: null,

  threshold: 0,

});

headerObserver.observe(header);
```



Revealing blur images while scrolling

```
//reveal section using Intersection Observer api

const allSections = document.querySelectorAll('.section');

const revealSection = function (entries, observer) {

 entries.forEach(entry => {

  if (!entry.isIntersecting) return;

  entry.target.classList.remove('section--hidden'); // Add the revealed class

  observer.unobserve(entry.target); // Stop observing once revealed

 });

};
```

```
const sectionObserver = new IntersectionObserver(revealSection, {

  root: null,

  //rootMargin: '0px',

  threshold: 0.15, // Trigger when 10% of the image is visible

});

allSections.forEach(function (section) {

  sectionObserver.observe(section);

  section.classList.add('section--hidden');

});
```

Lazy Loading

Lazy loading images in JavaScript is a technique to defer loading images until they are about to enter the viewport, improving page performance and user experience. Here's how you can implement it:

Using Native Lazy Loading (loading="lazy")

Modern browsers support native lazy loading via the loading attribute in the <img> tag:

<img src="example.jpg" alt="Example" loading="lazy">

This approach is straightforward and doesn't require JavaScript. It works automatically in supported browsers.

Custom Lazy Loading with JavaScript

If you need more control or are targeting browsers that don't support the loading attribute, use JavaScript.

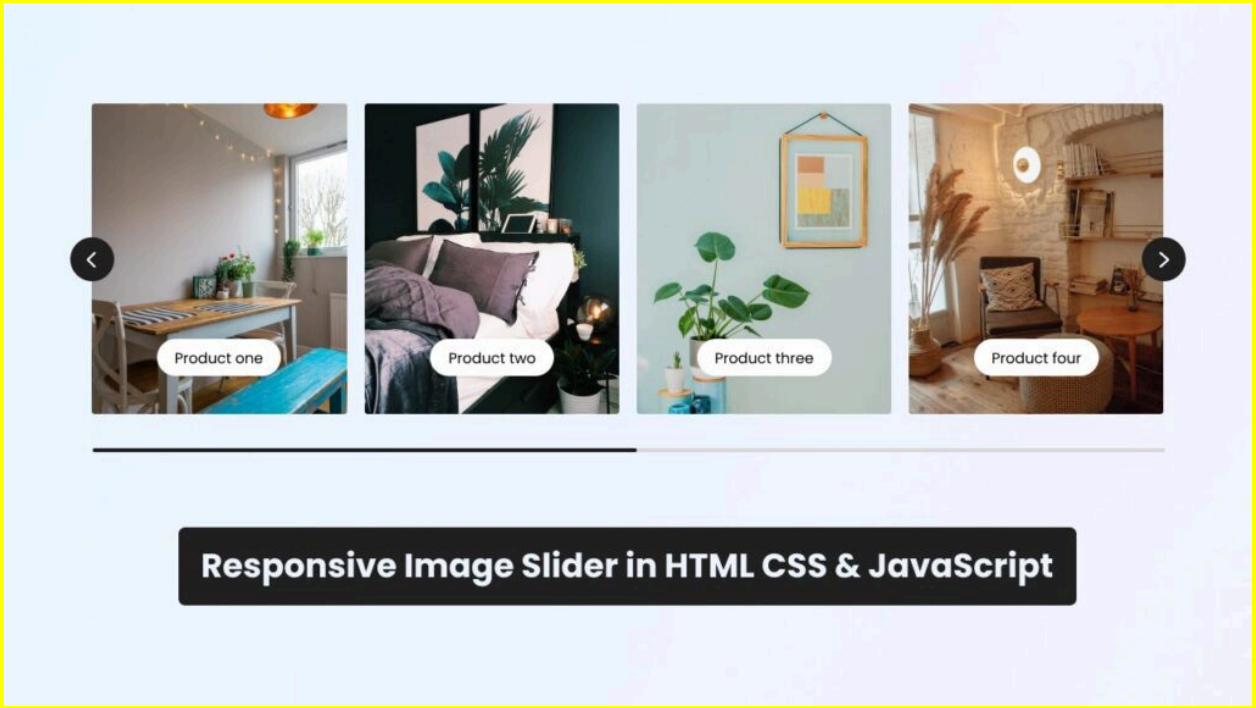
Steps:

1. Add a placeholder image or leave the src attribute empty.
2. Store the actual image URL in a data-src attribute.
3. Use JavaScript to load the image when it enters the viewport.

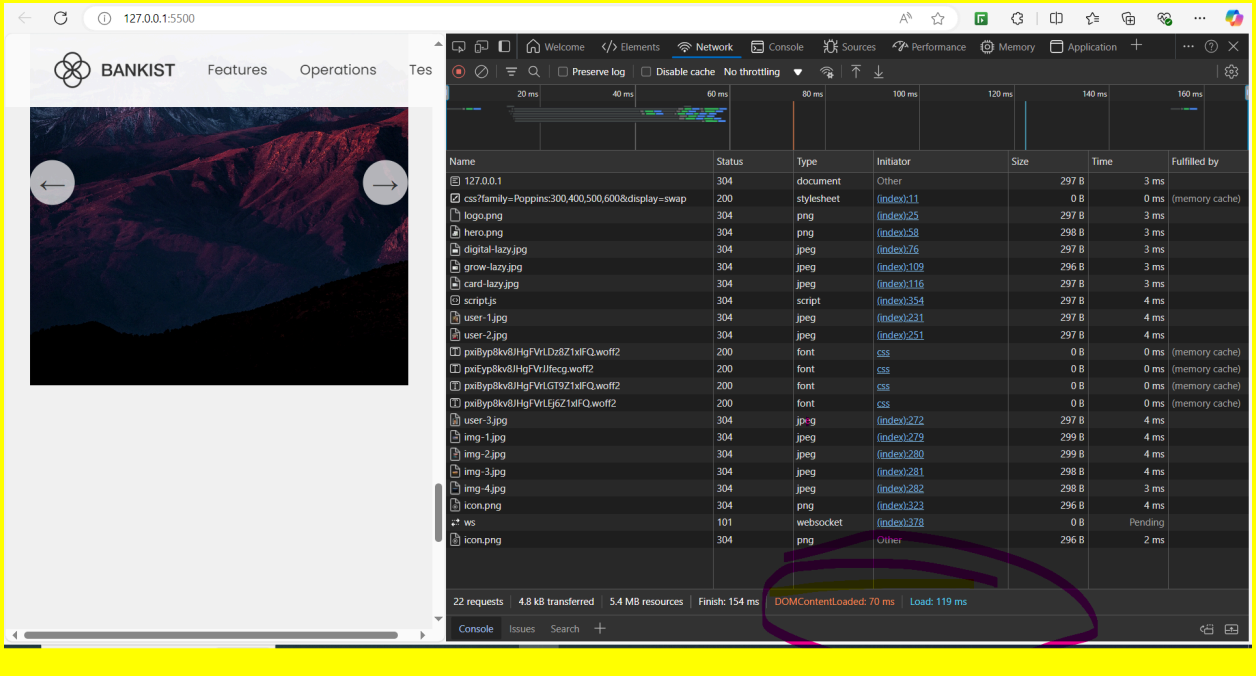
Example:

```
<img data-src="example.jpg" alt="Example" class="lazy-image" width="400" height="300">
```

Slider Component



Life Cycle of DOM



Script Loading Methods

1. Regular Script Loading (Default Behavior)

When a script is loaded without any additional attributes (async or defer), it is loaded and executed **synchronously** as the browser encounters the <script> tag.

- **Default Behavior**:
  - The browser pauses HTML parsing.
  - Download the script.
  - Executes the script immediately before continuing with the rest of the HTML.
- **Impact**:
  - Can block the rendering of the page.
  - If the script takes time to load, the user might experience delays in page rendering.

```
<!-- Regular Script -->
<script src="script.js"></script>
```

2. Async Script Loading

When the async attribute is added to a <script> tag, the script is loaded **asynchronously(**not happening at the same time as something else**)** but is executed **immediately after loading**.

- **Behavior**:
    - The browser continues parsing HTML while the script is downloaded in the background.
    - Once the script is loaded, the browser stops HTML parsing, executes the script, and then resumes parsing.
- **Impact**:
    - Useful for scripts that don't depend on the DOM structure or other scripts.
    - If multiple async scripts are present, they can execute in **random order** depending on which script loads first.

```
<!-- Async Script -->
<script src="script.js" async></script>
```

3. Defer Script Loading

When the defer attribute is used, the script is downloaded **asynchronously** but executed only **after the HTML document has been completely parsed**.

- **Behavior**:
    - The browser continues parsing HTML while downloading the script in the background.
    - Scripts with the defer attribute are executed in the order they appear in the document, **after the parsing of the HTML is complete**.
- **Impact**:
    - Ideal for scripts that depend on the DOM structure or need to execute in a specific order.

```
<!-- Defer Script -->
<script src="script.js" defer></script>
```

Placement of <script> in head vs. body

Scripts in <head>

- **Without async or defer**: Blocks HTML parsing, which can slow down page load.
- **With async**: Loads scripts in parallel but executes them as soon as they’re ready, possibly before the HTML is fully parsed.

- **With defer**: Allows HTML to be fully parsed before script execution, avoiding disruption.

  Example:

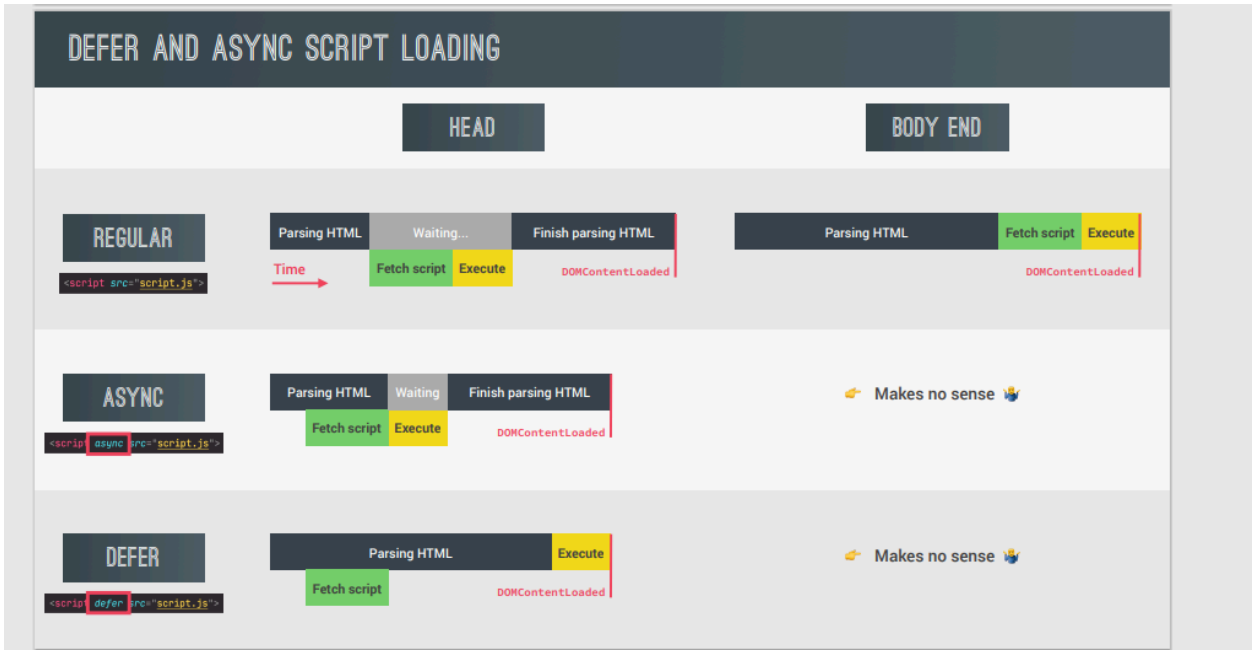
  ```
  <head>
    <script src="script.js" defer></script>
  </head>
  ```

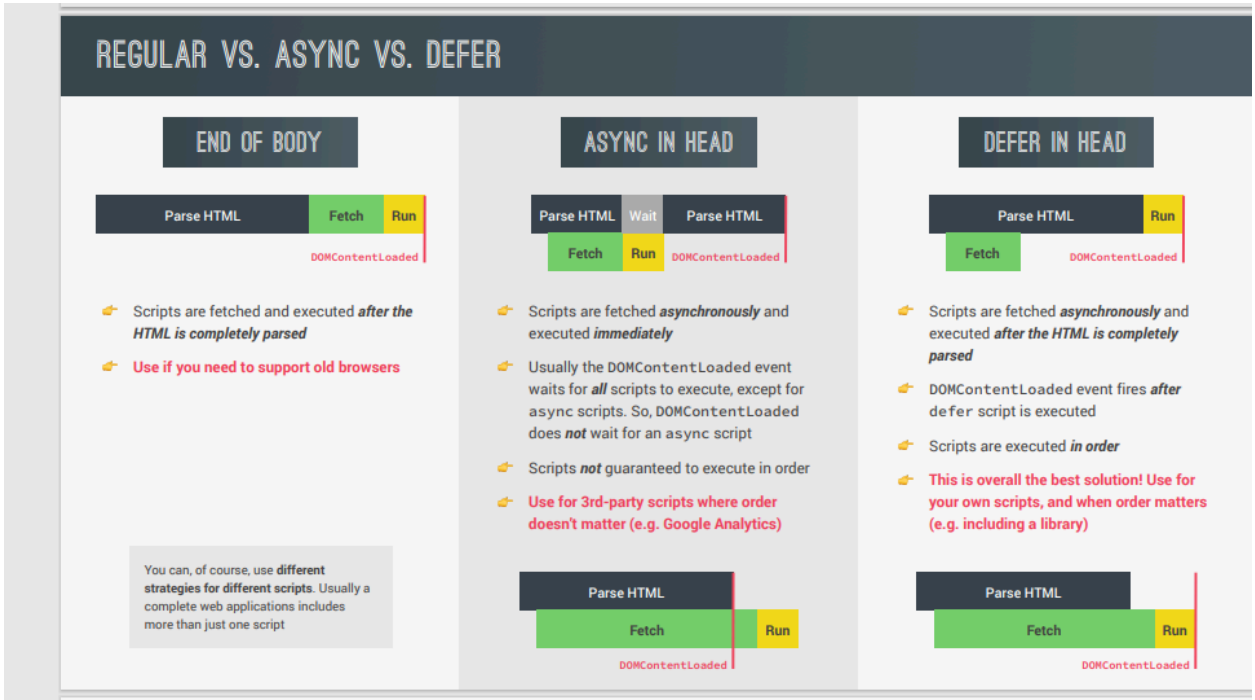
Scripts in <body>

- **Without async or defer**: Executes scripts as they are encountered, which can delay rendering the rest of the page.
- **With async or defer**: Similar behavior to when placed in the <head>.

  Example:

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


DEFER AND ASYNC SCRIPT LOADING
HEAD
BODY END
REGULAR
<script src="script.js">
Parsing HTML
Waiting...
Finish parsing HTML
Time
Fetch script
Execute
DOMContentLoaded
Parsing HTML
Fetch script
Execute
DOMContentLoaded
ASYNC
<script async src="script.js">
Parsing HTML
Waiting
Finish parsing HTML
Fetch script
Execute
DOMContentLoaded
Makes no sense
DEFER
<script defer src="script.js">
Parsing HTML
Execute
Fetch script
DOMContentLoaded
Makes no sense


REGULAR VS. ASYNC VS. DEFER
END OF BODY
Parse HTML
Fetch
Run
DOMContentLoaded
Scripts are fetched and executed after the HTML is completely parsed
Use if you need to support old browsers
You can, of course, use different strategies for different scripts. Usually a complete web applications includes more than just one script
ASYNC IN HEAD
Parse HTML
Wait
Parse HTML
Fetch
Run
DOMContentLoaded
Scripts are fetched asynchronously and executed immediately
Usually the DOMContentLoaded event waits for all scripts to execute, except for async scripts. So, DOMContentLoaded does not wait for an async script
Scripts not guaranteed to execute in order
Use for 3rd-party scripts where order doesn't matter (e.g. Google Analytics)
Parse HTML
Fetch
Run
DOMContentLoaded
DEFER IN HEAD
Parse HTML
Run
Fetch
DOMContentLoaded
Scripts are fetched asynchronously and executed after the HTML is completely parsed
DOMContentLoaded event fires after defer script is executed
Scripts are executed in order
This is overall the best solution! Use for your own scripts, and when order matters (e.g. including a library)
Parse HTML
Fetch
Run
DOMContentLoaded