What are these ?

- ECMAScript
- JavaScript
- ES5
- ES6
- ES7
- ES2015
- ES2016
- ES2017
- ES2018

History of Js

- JavaScript / ECMAScript
- JavaScript was invented by **Brendan Eich** in **1995**.
- It was developed for Netscape 2, and became the ECMA-262 standard in 1997.
- After Netscape handed JavaScript over to ECMA, the Mozilla foundation continued to develop JavaScript for the Firefox browser. Mozilla's latest version was 1.8.5. (Identical to ES5).
- Internet Explorer (IE4) was the first browser to support ECMA-262 Edition 1 (ES1).

History of Js

- The Initial Name of Js is Mocha
- Then livescript
- LiveScript --- Javascript (because of popularity of Java)
- Netscape Navigator becomes so popular because of Js
- Netscape Navigator ----- Javascript
- Internet Explorer ---- Jscript(Copy of Js)
- Role of ECMA and the Name ECMAScript(Standardisation of Web browser.)
- More Info - https://www.w3schools.com/js/js_history.asp

Setting of Environment for Js

- Download VS Code
- https://code.visualstudio.com/download

- Download Extension Live-server/Live-preview

Hello World Program

- console.log("Hello World")

Introduction to Variables

- Variables can store some information
- We can use the information later
- We can change the information later

Declaration of Variables

- var name = "Lokesh"; (you can use either ' or ")

Use of variables

- Console.log(name)

Change value of variables

name = "Aditya"

Rules for Naming of Variables

- Name cannot start with number
- Start with alphabet ,underscore(_) or $ symbol
- You cannot use spaces
- Start with small letter and use of camelCase
- No use of Special character (%,&,*,...)

Let and const keyword

- You can also make variable using let and const keyword

Example

let num = 10

The main Difference Between var,let and const is

var – Can be re-declared and re-assigned

let – Cannot be re-declared but re-assigned

const – Cannot be re-declared and re-assigned

*** block scope and function scope will be discussed later**

String Indexing

- let name = "Amit"

String = A   M   I   T

Index = 0   1   2   3

String length

let length = string_name.length

Note - The spaces are count in the length of string

The Last character is present at index length-1 index

String Methods

- **slice() Method** - extracts a part of the string based on the given stating-index and ending-index and returns a new string.

- **substring() Method** - returns the part of the given string from the start index to the end index. Indexing starts from zero (0).

- **substr() Method** - his method returns the specified number of characters from the specified index from the given string. It extracts a part of the original string.

- **replace()** - replaces a part of the given string with another string or a regular expression. The original string will remain unchanged.

- **replaceAll()** - returns a new string after replacing all the matches of a string with a specified string or a regular expression.

The original string is left unchanged after this operation.

String Methods

- **toUpperCase()**- converts all the characters present in the String to upper case and returns a new String with all characters in upper case.
- **toLowerCase()** - converts all the characters present in the String to lower case and returns a new String with all characters in lower case.
- **concat() Method** - method combines the text of two strings and returns a new combined or joined string.
- **trim() Method** -is used to remove either white spaces from the given string. This method returns a new string with removed white spaces
- **trimStart() Method** - removes whitespace from the beginning of a string. The value of the string is not modified in any manner, including any whitespace present after the string.
- **trimEnd() Method** - removes whitespace from the ending of a string. The value of the string is not modified

  in any manner, including any whitespace present after the string.

String Methods

- **padStart() Method** - pad a string with another string until it reaches the given length. The padding is applied from the left end of the string.
- **padEnd() Method** - pad a string with another string until it reaches the given length. The padding is applied from the right end of the string.
- **charAt() Method** - returns the character at the specified index. String in JavaScript has zero-based indexing.
- **charCodeAt() Method** - eturns a number that represents the Unicode value of the character at the specified index. This method accepts one argument.
- **split() Method** - splits the string into an array of sub-strings. This method returns an array. This method accepts a single parameter character on which you want to split the string.

Datatypes in JavaScript

- String
- Number
- Undefined
- Null
- Boolean
- BigInt
- Symbol

Note – Most Frequently used datatypes are number,string, boolean,undefined, null.

typeof operator in JS

- In JavaScript, the `typeof` operator is used to determine the type of a variable, expression, or value. It returns a string indicating the data type. Here are some common uses:

- let x; console.log(typeof x);

// Output: "undefined"

- let y = 42; console.log(typeof y);

// Output: "number"

Convert String into number in Js

- Use of + or - Operator
- Use of Number() method
- Use of ParseInt() method

Convert Number to String in Js

- Use of "" and +

  Ex - let num = 10 + ""

- Use of String() method

  Ex - let age = String(18)

- Use of toString() method

String Concatenation

- In JavaScript, string concatenation is the process of joining two or more strings together. There are several ways to concatenate strings in JavaScript:
- We can Concatenate String by following Ways
- ***Using + Operator***
- ***Using += Operator***
- ***Using Concat Method()***
- ***Using Template Literals***

String Literals

- In JavaScript, **string literals** are a way to represent strings directly in the source code. They are enclosed by either single quotes (`'`), double quotes (`"`), or backticks (`` ` ``), depending on the type of literal.
- **Template Literals (Backticks, ES6)**: Template literals, enclosed by backticks (`` ` ``), are a more flexible form of string literals introduced in ES6 (ECMAScript 2015). They allow for multi-line strings and variable interpolation using the `${expression}` syntax.

```
let name = "John";

let greeting = `Hello, ${name}! Welcome to the world of JavaScript.`;

console.log(greeting);  // Output: "Hello, John! Welcome to the world of JavaScript."
```

Booleans and Comparison Operator

- In JavaScript, **comparison operators** are used to compare values and return a Boolean result (`true` or `false`). These operators can compare numbers, strings, objects, and other types of data.
- The Comparison Operators are

>, <, >=, <= , ==, != , ===, !==

Q) == V/s ===

== - Compares two values for equality after type conversion (if necessary). This is also called **loose equality**.

=== - Compares two values for equality without type conversion. This is also called **strict equality**.

Examples

- console.log(10 > 5); // true
- console.log(10 === 10); // true
- console.log(10 !== '10'); // true
- console.log('apple' < 'banana'); // true
- console.log(null == undefined); // true (loose equality)
- console.log(null === undefined); // false (strict equality)
- ***Booleans – These are Values which can either be true or false**
- **True, false**

Truthy and Falsy Values In Js

- In JavaScript, **truthy** and **falsy** values refer to how non-Boolean values are evaluated in Boolean contexts (such as conditionals like `if` statements). These values are not strictly `true` or `false`, but when evaluated in a Boolean context, they behave as `true` or `false`.
- Falsy Values = 0, NaN, "", undefined, false, null, -0
- Other all are Truthy values

Example

```
if (!0) {
console.log("0 is falsy"); // This will be printed
}
```

Conditional Statements

- In JavaScript, **conditional statements** allow you to execute different code based on certain conditions. These are essential for controlling the flow of a program. Here are the main types of conditional statements in JavaScript:
- **`if` Statement**
- **`if...else` Statement**
- **`if...else if...else` Statement**
- **Ternary Operator (`? :`)**
- **`switch` Statement**

For loop in JavaScript

- The for loop is the most commonly used loop in JavaScript. It repeats a block of code a specified number of times. It is especially useful when you know in advance how many iterations you want the loop to run.

Syntax

```
for (initialization; condition; increment/decrement){
// Code to be executed
 }
```

***Note** – Later we will read About for-of and for-in loop

While Loop

- The `while` loop executes a block of code as long as the specified condition evaluates to true. The condition is checked before executing the loop, so if the condition is `false` from the start, the code inside the loop may never execute.

Syntax

```
while (condition) {
  // Code to be executed
}
```

Do while Loop

- The `do...while` loop is similar to the `while` loop, but the condition is evaluated **after** the code block has been executed. This guarantees that the code block runs at least once, regardless of whether the condition is true or false.

Syntax:

```
do {
  // Code to be executed
} while (condition);
```

Break and Continue Statements

- **These statements are used to control the flow of loops.**
- **`break`**: Exits the loop immediately when executed.
- **`continue`**: Skips the current iteration and moves to the next iteration of the loop.

Example

```
for (let i = 0; i < 10; i++) {
 if (i === 5) {
   break; // Exits the loop when i is 5
 }
 console.log(i); // Output: 0, 1, 2, 3, 4
}
```

Arrays in Js

- An **array** is a special type of object in JavaScript used to store multiple values in a single variable. Arrays are useful when working with lists or collections of data. Each value in an array has an index (starting from 0), and the values can be of any type (numbers, strings, objects, etc.).

How to Check the typeof array ?

Array.isArray(arr-name)

Note - Array is considered as object because is it reference type of datatypes

Refernce and Primitive Datatypes 🤔🤔😕

Creation of Array

You can create an array in multiple ways:

1. **Using Array Literal Syntax**

This is the most common way to create an array.

```
let fruits = ["Apple", "Banana", "Orange"];
```

2. **Using the `Array()` Constructor**

You can also use the `Array()` constructor, though it's less common.

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

Accessing Array Elements

- You can access elements in an array using their index, with the first element at index 0.
- **Syntax** - arrayName[index];

Example

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits[0]);  // Output: Apple
console.log(fruits[1]);  // Output: Banana
```

Modifying Array Elements

- You can modify any element in an array by directly assigning a new value to a specific index.

Example

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

fruits[1] = "Mango"; *Replaces* "Banana" with "Mango"

console.log(fruits); **Output**: ["Apple", "Mango", "Orange"]

Array Properties

- **`length`**: The `length` property returns the number of elements in an array.

Example

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

console.log(fruits.length); **Output**: 3

Now Understand Primitive V/s Reference Type datatype ⭐😀

Iterating Over Arrays

- **Using `for` Loop:**
- You can loop through an array using a traditional `for` loop.

```
let fruits = ["Apple", "Banana", "Orange"];
for (let i = 0; i < fruits.length; i++) {
 console.log(fruits[i]);
}
```

Output: Apple, Banana, Orange

Other Loops in Array

- While
- Do-while loop
- For-of loop
- For-in loop

Array Methods

- `push()`: Adds one or more elements to the end of an array and returns the new length of the array.
- `pop()`: Removes the last element from an array and returns that element.
- `shift()`: Removes the first element from an array and shifts all other elements to a lower index.
- `unshift()`: Adds one or more elements to the beginning of an array and returns the new length of the array.
- `splice()`: Adds, removes, or replaces elements in an array at a specified index.
- `slice()`: Returns a shallow copy of a portion of an array into a new array, without modifying the original array.
- `concat()`: Merges two or more arrays into a new array without modifying the original arrays.
- `indexOf()`: Returns the first index at which a given element can be found in the array, or `-1` if it is not present.
- `lastIndexOf()`: Returns the last index at which a given element can be found in the array, or `-1` if it is not present.

Array Methods

- **`find()`**: Returns the first element in the array that satisfies the provided testing function.
- **`findIndex()`**: Returns the index of the first element that satisfies the provided testing function.
- **`some()`**: Tests whether at least one element in the array passes the provided function.
- **`every()`**: Tests whether all elements in the array pass the provided function.
- **`join()`**: Joins all elements of an array into a string, with a specified separator.
- **`reverse()`**: Reverses the order of the elements in an array.
- **`sort()`**: Sorts the elements of an array in place and returns the sorted array.
- **includes()**: Checks if an array contains a specified element and returns true or false.

Array Destructuring

- Destructuring allows you to unpack values from an array into distinct variables.

Example

```
let fruits = ["Apple", "Banana", "Orange"];
let [first, second] = fruits;
console.log(first);  // Output: Apple
console.log(second);  // Output: Banana
```

***Other Ways to Destructure Array Element**

Ways to Copy Array

Problem with Direct Copy 🙄🤷‍♂️

Ways to Copy Array

1. Concat() Method
2. Slice() method
3. Spread Operator (...)

Important Points

- *Arrays in JavaScript can hold values of different data types.*
- *Arrays are **zero-indexed**, meaning the first element has an index of 0.*
- *Arrays are dynamic, so their size can change as you add or remove elements.*

Objects in JavaScript

- An **object** in JavaScript is a collection of key-value pairs. Each key is also known as a **property**, and the value can be of any data type (e.g., number, string, array, function, etc.). Objects are used to store and manage related data and functionality in a structured way.
- **Example**

```
let objectName = {
 key1: value1,
 key2: value2,
 ...
};
```

Creation of Object

- You can create an object in multiple ways:
- **Using Object Literal Syntax**
- This is the most common way to define an object in JavaScript.
- **Syntax**

```
let objectName = {
  key1: value1,
  key2: value2,
  ...
};
```

Creation of Object

Using the new `Object()` Syntax

This creates an object using JavaScript’s built-in Object constructor.

```
let objectName = new Object();
objectName.key1 = value1;
objectName.key2 = value2;
```

Accessing the Properties of Object

- You can access the properties of an object in two ways:
- **Dot Notation:**

  objectName.propertyName;
- **Bracket Notation**

  objectName["propertyName"];
- **Why there are two method for Accessing Property ?** 🤔 💭

Modifying Objects And Adding Properties

- **Modifying Object Properties**

You can modify the properties of an object by assigning new values

```
objectName.propertyName = newValue;
```

- . **Adding New Properties**

You can add new properties to an object after it has been created.

```
objectName.newProperty = value;
```

Deleting Properties

- You can remove properties from an object using the `delete` **operator.**

```
delete objectName.propertyName;
```

Checking for Properties in Object

- You can check if a property exists in an object using the `in` operator or the `hasOwnProperty()` method.
- **Using the `in` Operator**:

```
"propertyName" in objectName;
```

- **Using the `hasOwnProperty()` Method**:

```
objectName.hasOwnProperty("propertyName");
```

Object Methods

- **`Object.keys()`**: Returns an array of an object’s property names (keys).
- **`Object.values()`**: Returns an array of an object’s property values.
- **`Object.entries()`**: Returns an array of key-value pairs for the object.
- **`Object.assign()`**: Copies the values of all enumerable properties from one or more source objects to a target object.
- **`Object.freeze()`**: Freezes an object, preventing new properties from being added or existing properties from being modified.

Object Methods

- **`Object.seal()`**: Seals an object, preventing new properties from being added, but allows modification of existing properties.
- **`Object.create()`**: Creates a new object with the specified prototype object and properties.
- **`Object.hasOwn()`**: Determines whether an object has a specified property as a direct property of that object.

Iterating Over Objects

- You can loop through the properties of an object using the **`for...in`** loop.

```
for (let key in objectName) {
  console.log(key + ": " + objectName[key]);
}
```

Nested Objects

- An object can contain another object as a property, creating nested objects.
- **Example**

```
let objectName = {
 nestedObject: {
  key1: value1,
  key2: value2
 }
};
```

Destructuring of Objects

- **Object destructuring** is a feature in JavaScript that allows you to extract values from objects and assign them to variables in a more concise way. It simplifies code by unpacking properties of an object into individual variables.
- **<u>Basic Syntax</u>**

```
let { key1, key2 } = objectName;
```

Destructuring of Objects

- **Examples**

```
let person = { name: "John", age: 30 };
let { name, age } = person;
console.log(name);  // Output: John
console.log(age);   // Output: 30
```

- **Renaming Variables**

You can also rename the destructured variables by using a colon :

```
let { key1: newVariableName1, key2: newVariableName2 } = objectName;
```

Destructuring of Objects

- You can assign default values to variables in case the corresponding property in the object is **`undefined`**.

Syntax

let { **key1 = defaultValue1, key2 = defaultValue2** } = ***objectName***;

Example

```
let person = { name: "John" };

let { name, age = 25 } = person;

console.log(name);  // Output: John

console.log(age);   // Output: 25
```

Spread Operator in Objects

- **Cloning objects**: To create a copy of an object without modifying the original.

```
let newObject = { ...oldObject };
```

- **Merging objects**: To combine properties from multiple objects into one.

```
let mergedObject = { ...object1, ...object2 };
```

- **Adding/modifying properties**: To easily add or change properties in an object.

```
let updatedObject = { ...oldObject, newProperty: newValue };
```

Array of Objects

- An **array of objects** is an array where each element is an object. This is commonly used in JavaScript to store multiple objects, especially when managing collections of data, such as user profiles, products, or items.
- Objects inside Array
- Very useful for real world Application
- We can Get the element by indexing and we are also loop through it

Nested Destructuring

```
const users = [
  {userId: 1,firstName: 'harshit', gender: 'male'},
  {userId: 2,firstName: 'mohit', gender: 'male'},
  {userId: 3,firstName: 'nitish', gender: 'male'},
]
const [{firstName: user1firstName, userId}, , {gender: user3gender}] = users;
console.log(user1firstName);
console.log(userId);
console.log(user3gender);
```

Functions In JavaScript

A **function** in JavaScript is a block of code designed to perform a specific task. Functions allow you to encapsulate logic, making your code reusable, organized, and easier to maintain.

Defining a Function

There are different ways to define functions in JavaScript:

- **Function Declaration**

A named function that can be called anywhere in the code after its declaration.

- **Function Expression**

A function can also be stored in a variable. This is known as a function expression.

- **Arrow Functions (ES6)**

A shorter syntax for function expressions using the => arrow.

Function Declaration

Syntax

```
function functionName(parameters) {
  // function body
}
```

Example

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

Function Expression

Syntax

```
let functionName = function(parameters) {
  // function body
};
```

Example

```
let greet = function(name) {
  return `Hello, ${name}!`;
};
```

Arrow Function

Syntax

```
let functionName = (parameters) => {
  // function body
};
```

Example

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

<u>**Note:-**</u>

- ***<u>If You Function code is One line you can ignore curly bracket {} in Arrow function</u>***
- ***<u>If You have only one parameter in function then you can ignore () brackets</u>***

Others things to read in functions

- Calling of functions
- Parameters and Arguments
- Default Parameters
- Return statements
- Immediately Invoked Function Expression (IIFE)

Calling of Functions

- To execute a function, you "call" or "invoke" it by using its name followed by parentheses. If the function has parameters, you pass them inside the parentheses.
- greet("John"); **Output**: *Hello, John!*

Function Parameters and Arguments

- **Parameters** are placeholders in the function definition that accept values when the function is called.
- **Arguments** are the actual values passed to the function during the function call.

```
function add(a, b) {
  return a + b;
}
add(5, 3);  Output: 8
```

Default Parameters

- You can assign default values to function parameters. If no value is passed for a parameter, the default value will be used.

Example

```
function greet(name = "Guest") {
 return `Hello, ${name}!`;
}
greet(); Output: Hello, Guest!
```

Rest Parameter in Functions

- The rest parameter in JavaScript allows a function to accept an indefinite number of arguments as an array. It is represented by three dots (`...`) followed by a name for the array.

Syntax

```
function functionName(...rest) {
 // Function body
}
```

Key Features

- The rest parameter must be the **last** parameter in the function's parameter list.
- It allows you to gather all remaining arguments into a single array.

Example of Rest Parameter

```
function displayInfo(name, ...hobbies) {
  console.log(`Name: ${name}`);
  console.log('Hobbies:');
  hobbies.forEach(hobby => console.log(hobby));
}
displayInfo('Alice', 'Reading', 'Hiking', 'Cooking');
```

Return Statement

- Functions can return a value using the `return` statement. Once `return` is executed, the function stops running.

```
function multiply(a, b) {
  return a * b;
}
let result = multiply(4, 5);  Output: 20
```

Immediately Invoked Function Expression (IIFE)

- An **IIFE** is a function that runs immediately after it is defined. It's used to avoid polluting the global scope.

Example

```
(function() {
  console.log("This function runs immediately!");
})();
```

Functions as First-Class Citizens

- In JavaScript, functions are treated as **first-class citizens**, meaning:
- They can be passed as arguments to other functions.
- They can be returned from other functions.
- They can be assigned to variables.

Passing Function as an Argument

```
function greet(name) {
   console.log("Hello, " + name);
}
function processUserInput(callback) {
   const name = "Lokesh Singh";
   callback(name);
}
processUserInput(greet);
```

*** Give some more Examples to clear the concept**

Function Returning a function

```
function add(a) {
  return function(b) {
    return a + b;
  };
}
let addFive = add(5);
console.log(addFive(3));
```

Output: 8

*** Give some more Examples to clear the concept**

Higher Order Functions

- A **higher-order function** is a function that takes another function as an argument or returns a function.

```
 function greet(greeting) {
  return function (name) {
      console.log(`${greeting}, ${name}!`);
      };
 }
const sayHello = greet("Hello");
sayHello("John"); **Outputs**: "Hello, John!"
```

Common Example of High Order Functions(HOF)

- **`map()`:**

Creates a new array by applying a function to each element of an existing array.

- **`filter()`:**

Creates a new array by selecting elements from an existing array that pass a certain test.

- **`reduce()`:**

Iterates over an array and accumulates a single value based on a provided function.

- **`forEach()`:**

Executes a function for each element in an array.

Map Method in Js

- <u>**Syntax**</u>
- *array*.map(*function(currentValue, index, arr)*)
- <u>**Example**</u>

```
const persons = [
{firstname : "Malcom", lastname: "Reynolds"}, {firstname : "Kaylee", lastname: "Frye"}, {firstname : "Jayne", lastname: "Cobb"}
];
const getfullname = (data) => data.firstname + " " + data.lastname;
const ans = persons.map(getfullname);
console.log(ans)
```

Filter Method in Js

- **Syntax**
- *array*.filter(*function(currentValue, index, arr)*)
- **Example**

```
const persons = [
 {firstname : "Malcom", lastname: "Reynolds"}, {firstname : "Kaylee", lastname: "Frye"}, {firstname : "Jayne", lastname: "Cobb"}
];
const getfullname = (data) => data.firstname == "Jayne";
const ans = persons.filter(getfullname);
Console.log(ans)
```

ForEach Method in Js

- array.forEach(function(currentValue, index, arr))
- Return Value – undefined

Example

```
let sum = 0;
const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);

function myFunction(item) {
  sum += item;
}
```

Reduce Method in Js

- array.reduce(function(total, currentValue, currentIndex, arr), initialValue)
- Return value - The accumulated result from the last call of the callback function.

Example

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
 console.log(sum); // Output: 10
```

Sort Method in Js

- The `sort` method in JavaScript is used to sort the elements of an array in place and returns the sorted array. By default, the `sort` method sorts the elements as strings in ascending order.

Syntax

array.sort([compareFunction])

Parameters

compareFunction (optional): A function that defines the sort order. If omitted, the elements are sorted as strings.

Default Behaviour of Sort

- When sorting without a compare function, the `sort` method converts each element to a string and compares their sequences of UTF-16 code unit values.
- **Example 1: Default Sorting**

const fruits = ['banana', 'apple', 'orange', 'mango'];

fruits.sort();

console.log(fruits); ***Output****: ['apple', 'banana', 'mango', 'orange']*

Sorting Numbers

- When sorting numbers, you need to provide a compare function because the default behavior sorts them as strings.

const numbers = [10, 2, 5, 1, 9];

<u>Sort numbers in ascending order</u>

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

console.log(numbers); // **Output: [1, 2, 5, 9, 10]**

<u>Sort numbers in descending order</u>

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

console.log(numbers); **Output: [10, 9, 5, 2, 1]**

Sorting of objects

You can also sort an array of objects based on a specific property.

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 20 }
];
```

Sort users by age

```
users.sort((a, b) => a.age - b.age);
console.log(users);
```

Key Points of Sorting

- The `sort` method modifies the original array and does not create a new one.
- Always provide a compare function when sorting numbers or objects to ensure the correct order.
- The `sort` method is not stable; the order of equal elements may not be preserved.

The `sort` method is a powerful tool for organizing data in arrays according to specific criteria!

Some Method in Js

- The `some` method tests whether at least one element in the array passes the test implemented by the provided function. It returns `true` if it finds an element that satisfies the condition; otherwise, it returns `false`.

Syntax

array.some(function(value, index, arr))

Example

const numbers = [1, 2, 3, 4, 5]; // Check if there is at least one even number

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

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

Every Method of Js

- The `every` method tests whether all elements in the array pass the test implemented by the provided function. It returns `true` if all elements satisfy the condition; otherwise, it returns `false`.

Syntax

array.every(function(value, index, arr))

<u>Example</u>

const ages = [15, 18, 21, 30]; // **Check if all are adults** (18 or older)

const allAdults = ages.every(age => age >= 18);

console.log(allAdults); **Output**: false

Key Difference Between Some and Every method of Js

- **`some`:** Returns `true` if at least one element meets the condition; otherwise, it returns `false`.
- **`every`:** Returns `true` only if all elements meet the condition; otherwise, it returns `false`.

These methods are quite useful for validating conditions in arrays without needing to write loops explicitly!

Find Method of Js

- The find() method returns the value of the first element that passes a test.
- The find() method executes a function for each array element.
- The find() method returns undefined if no elements are found.
- The find() method does not execute the function for empty elements.
- The find() method does not change the original array.

Syntax

```
array.find(function(currentValue, index, arr))
```

Example of Find

const numbers = [4, 9, 16, 25];

<u>Find the first number greater than 10</u>

const firstGreaterThanTen = numbers.find(num => num > 10);

console.log(firstGreaterThanTen); **Output**: 16

Example of find() with Objects

```
const users = [
  { id: 1, name: 'Alice', age: 25 },
  { id: 2, name: 'Bob', age: 30 },
  { id: 3, name: 'Charlie', age: 35 }
];
```

Find the user with age 30

```
const user = users.find(user => user.age === 30);
console.log(user);
```

Output: { id: 2, name: 'Bob', age: 30 }

Splice method

- The `splice` method in JavaScript is used to change the contents of an array by adding or removing elements. It modifies the original array and returns an array containing the removed elements (if any).

Syntax

array.splice(start, deleteCount, item1, item2, ...)

Parameters

- **start:** The index at which to start changing the array.
- **deleteCount:** The number of elements to remove from the array starting at the `start` index. If set to `0`, no elements will be removed.
- **item1, item2, ... (optional):** The elements to add to the array, starting at the `start` index.

Removing Element Using Splice

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

<u>Remove 2 elements starting from index 1</u>

const removed = numbers.splice(1, 2);

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

console.log(removed); **Output**: [2, 3]

Adding Elements

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

<u>Add 'orange' and 'kiwi' starting at index 1</u>

fruits.splice(1, 0, 'orange', 'kiwi');

console.log(fruits); **Output**: ['apple', 'orange', 'kiwi', 'banana', 'cherry']

Replacing Elements

const colors = ['red', 'green', 'blue'];

<u>Replace 1 element at index 1 with 'yellow'</u>

const replaced = colors.splice(1, 1, 'yellow');

console.log(colors); **Output**: ['red', 'yellow', 'blue']

console.log(replaced); **Output**: ['green']

Use of Negative Indexing

const animals = ['cat', 'dog', 'rabbit', 'bird'];

<u>Remove the last element</u>

const removed = animals.splice(-1, 1);

console.log(animals); **Output**: ['cat', 'dog', 'rabbit']

console.log(removed); **Output**: ['bird']

Key Points of Splice

- The `splice` method modifies the original array and returns an array of removed elements.
- It can be used for adding, removing, or replacing elements in an array.
- Negative indices count backward from the end of the array.

The `splice` method is a versatile way to manipulate arrays directly, making it useful for various operations!

Fill Method of Js

- The `fill()` method in JavaScript is used to fill all the elements of an array from a start index to an end index with a static value. This method modifies the original array and returns the modified array.

Syntax

array.**fill**(value, start, end);

- **value**: The value to fill the array with.
- **start** (optional): The index to start filling from (default is `0`).
- **end** (optional): The index to stop filling (default is `array.length`).

Example of fill method

- **UseCase - 1**

```
const myArray = new Array(10).fill(0);
console.log(myArray);
```

- **UseCase - 2**

```
const myArray = [1,2,3,4,5,6,7,8];
myArray.fill(0,2,5);
console.log(myArray);
```

Iterables in Js

In JavaScript, **iterables** are objects that can be iterated over, meaning their elements can be accessed one by one. The most common built-in iterables in JavaScript are arrays, strings, maps, sets, and more.

Built-in Iterables in JavaScript

- **Arrays**
- **Strings**
- **Maps**
- **Sets**
- **Typed arrays**
- **The arguments object** (inside functions)
- **NodeList** (e.g., DOM elements)

Array like Objects

In JavaScript, array-like objects are objects that resemble arrays but don't have all the properties and methods of true arrays. Here are some key points about them:

Characteristics of Array-like Objects

- **Length Property**: They usually have a `length` property, similar to arrays.
- **Indexed Elements**: They can be indexed like arrays (e.g., `obj[0]`, `obj[1]`).
- **No Array Methods**: They do not have access to standard array methods like `.push()`, `.pop()`, `.forEach()`, etc.

Example of Array like Objects

- **Arguments Object**:

In functions, `arguments` is an array-like object containing the values of the arguments passed to the function.

```
function example() {
   console.log(arguments.length); // number of arguments
   console.log(arguments[0]); // first argument
}
example(1, 2, 3);
```

Example of Array like Objects

- **NodeList**:
- Methods like `document.querySelectorAll()` return a NodeList, which is array-like.

- **Example**

```
const items = document.querySelectorAll('div');
console.log(items.length); // number of div elements
console.log(items[0]); // first div element
```

Example of Array like Objects

- **HTMLCollection**:

Accessing elements via `document.getElementsByTagName()` returns an HTMLCollection.

- **Example**

```
const divs = document.getElementsByTagName('div');
console.log(divs.length);
```

Converting Array-like Objects to Arrays

- You can convert array-like objects into actual arrays using methods like:
- **Array.from()**:

```
const argsArray = Array.from(arguments);
```

- **Spread Syntax**:

```
const argsArray = [...arguments];
```

- **Array.prototype.slice**:

```
const argsArray = Array.prototype.slice.call(arguments);
```

Set in JavaScript

- In JavaScript, a `Set` is a built-in object that allows you to store unique values of any type, whether primitive values or object references. Here are the key features and functionalities of Sets:
- **Key Features**
- **Uniqueness**: A `Set` automatically ensures that all values are **unique**. If you try to add a duplicate value, it will be ignored.
- **Order**: Values in a `Set` are **ordered**. The order of insertion is **maintained**.
- **Dynamic Size**: A `Set` can grow and shrink dynamically as values are **added** or **removed**.

Creating a Set

- You can create a Set using the Set constructor:

const **mySet** = new Set();

- You can also initialize a Set with an array or any iterable:

const **mySet** = new Set([1, 2, 3, 4, 5]);

Basic Operations of Set

- **Add Value to a set**

mySet.add(1);

mySet.add(2);

mySet.add(2); **Ignored, since 2 is already present**

- **Checking for Values**:

console.log(mySet.has(1)); true

console.log(mySet.has(3)); false

Basic Operations of Set

- **Removing Values**:

  mySet.delete(2); Removes 2

- **Clearing the Set**:

  mySet.clear(); Removes all elements

- **Getting the Size**:

  console.log(mySet.size); **Returns the number of unique values**

Iterating Over a Set

- You can iterate through the values of a `Set` using the **`forEach`** method or a **`for...of`** loop:

```
mySet.forEach(value => {
  console.log(value);
});

for (const value of mySet) {
  console.log(value);
}
```

Converting a Set into an Array

- You can convert a `Set` back to an array using the **spread** operator or **`Array.from():`**

- const myArray = [...mySet]; **Using spread operator**
- const myArray2 = Array.from(mySet); **Using Array.from()**

Map in JavaScript

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

How to Create a Map

You can create a JavaScript Map by:

- Passing an Array to new Map()
- Create a Map and use Map.set()

The new Map() Method

- You can create a Map by passing an Array to the new Map() constructor:

- ***Example***

Create a Map

```
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

The set() Method

- You can add elements to a Map with the set() method:
- **Create a Map**
  const fruits = new Map();
- **Set Map Values**
  fruits.set("apples", 500);
  fruits.set("bananas", 300);
  fruits.set("oranges", 200);
- The set() method can also be used to change existing Map values:

  Example - fruits.set("apples", 200);

The other map method Method

- **get method() -** get the value of given key

  fruits.get("apples");   // Returns 500

- **has method –** check the key exists or not

  console.log(fruits.has("oranges")); // Output: true

- **delete method** – removes the entries from the map

  fruits.delete("bananas");

- **Size method** – return the size of Map

  console.log(fruits.size); **Output**: Number of entries

Iterating Over a Map

You can iterate over a Map using `forEach` or a `for...of` loop:

forEach loop

```
fruits.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});
```

Or using for...of

```
for (const [key, value] of fruits) {
  console.log(`${key}: ${value}`);
}
```

Converting Map into Array

- **Converting to Array**

You can convert a Map to an array of entries:

```
const entriesArray = Array.from(fruits); // [[key1, value1], [key2, value2]]
```

Difference Between Map and Objects

Object	Map
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be Strings (or Symbols)	Keys can be any datatype
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

Cloning of Object

- We can clone a Object Using Spread Operator as we Studied previously but we can also clone object using **Object**.**assign**(objname)

```
const obj = {
  key1: "value1",
  key2: "value2"
}
const obj1 = {'key69': "value69",...obj};
const obj2 = Object.assign({'key69': "value69"}, obj);
```

Optional Chaining in Js

- Optional chaining is a feature in JavaScript that allows you to safely access deeply nested properties of an object without having to check if each reference in the chain is valid. It simplifies your code by reducing the need for multiple checks for **`null`** or **`undefined`**.

Syntax

The optional chaining operator is `?.`. Here's how it works:

Optional Chaining Example - 1

```
const user = {
  profile: {
    name: "Alice"
  }
};
console.log(user.profile?.name); // "Alice"
console.log(user.profile?.age);  // undefined
console.log(user.address?.city);  // undefined
```

Optional Chaining Example - 2

```
const **user**  = {
   firstName: "harshit",
   address: {houseNumber: '1234'}
}

console.log(user?.firstName);
console.log(user?.address?.houseNumber);
```

Understanding Script Tag Placement

- The placement of a script tag within an HTML document can significantly impact the execution order of your JavaScript code. There are two primary locations: within the `<head>` section and within the `<body>` section.

Head Placement

- **Early Execution:** Scripts in the <head> section are executed before the page content is rendered, allowing for early initialization of global variables, functions, or essential page components.
- **Potential Rendering Delays:** Large or complex scripts in the <head> can block page rendering, leading to a perceived delay in loading.
- **Use Cases:** Ideal for scripts that define global variables, functions, or components that are essential for the page's functionality and need to be loaded before other content.

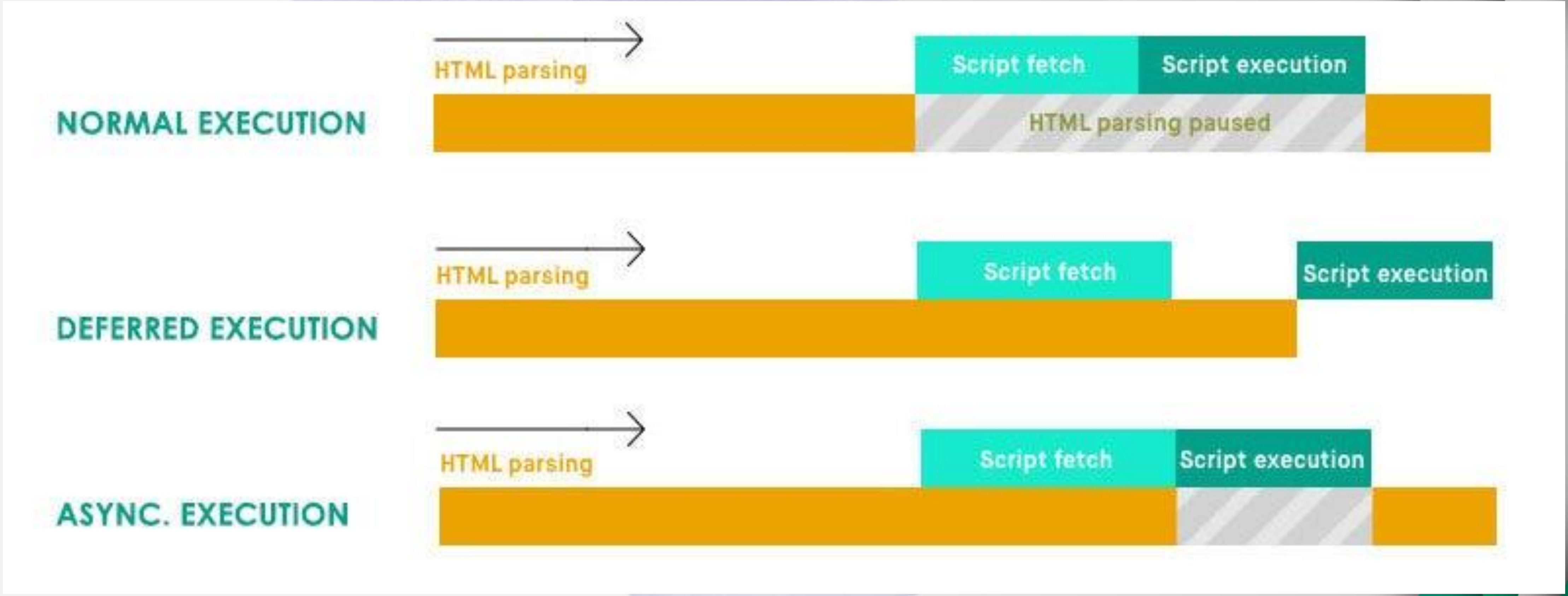
Body Placement

- **Delayed Execution:** Scripts in the <body> section are executed after the page content has been parsed and rendered, providing a more responsive user experience as the page's content is visible while the script is executing.
- **Interaction with Page Elements:** This placement is suitable for scripts that interact with elements on the page or handle user events, as they can wait for those elements to be loaded before executing.
- **Use Cases:** Ideal for scripts that handle user interactions, dynamic content updates, or other tasks that require the page elements to be already present.

Key Differences

Feature	Head Placement	Body Placement
Execution Timing	Before page rendering	After page rendering
Impact on Page Rendering	Can delay rendering	Generally less impact
Use Cases	Global variables, functions, essential scripts	Page interactions, event handlers

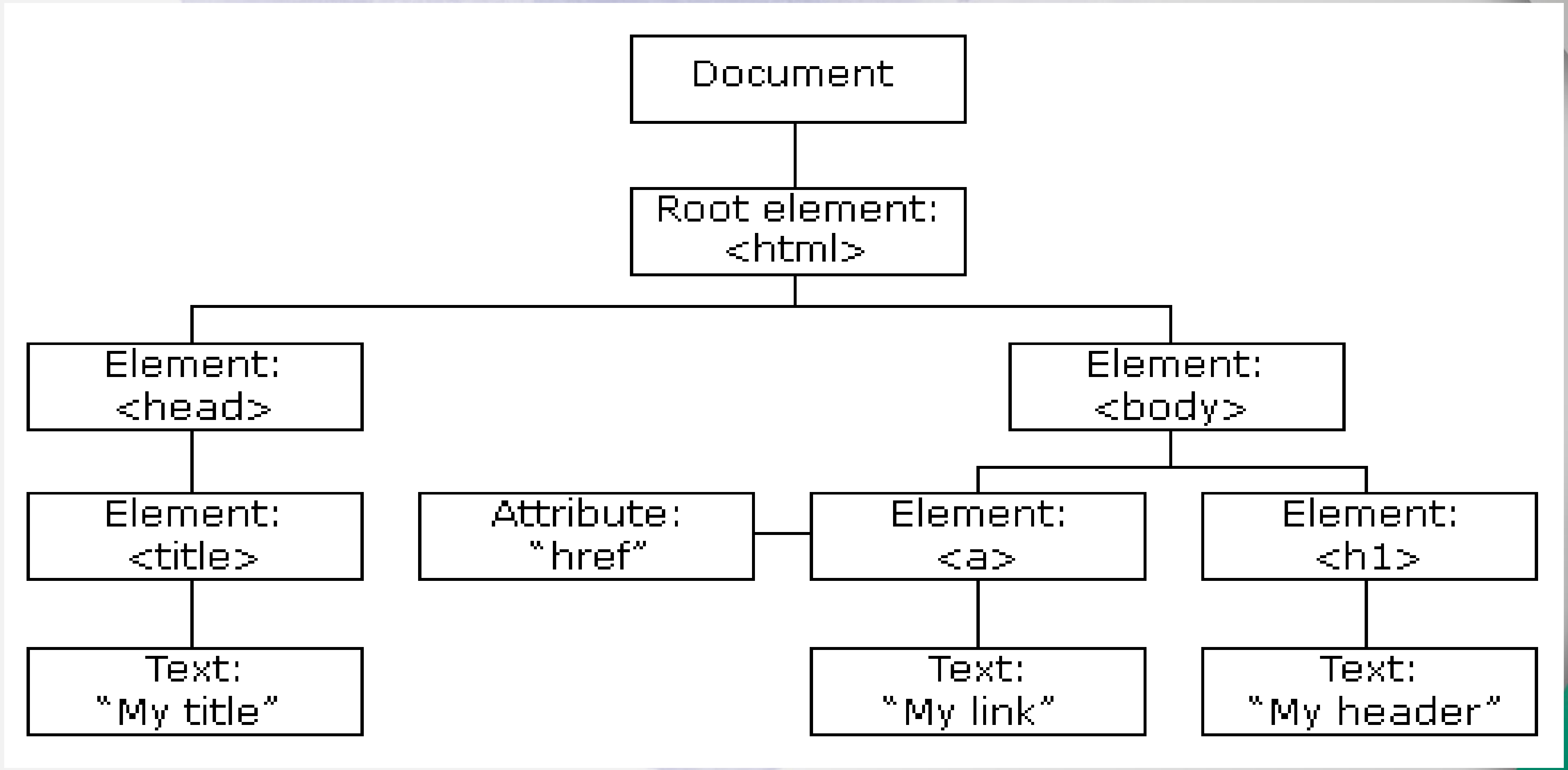
Script Execution Different Scenarios



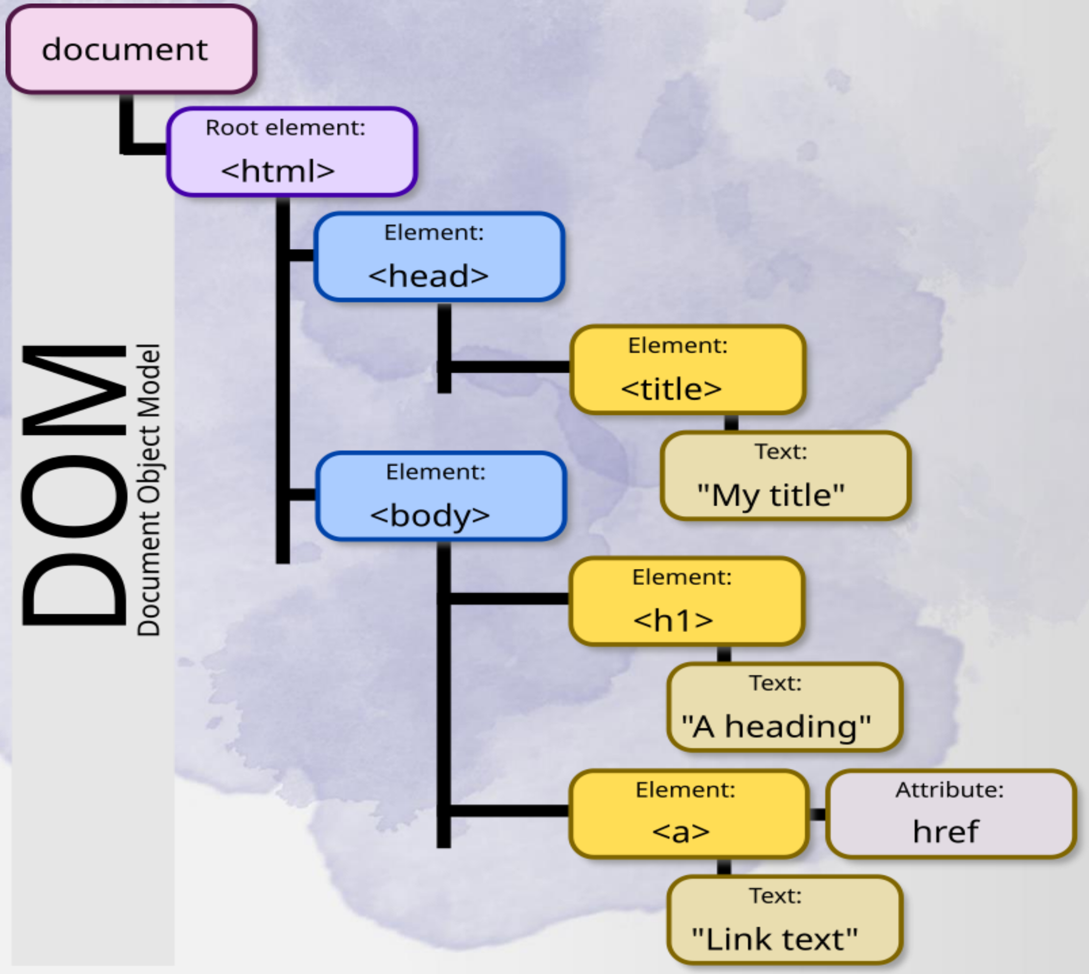
DOM – Document Object Model

- **Definition:** The DOM is a programming interface that represents an HTML document as a tree structure of nodes. Each node represents an element, attribute, text, or comment within the document.
- **Purpose:** It provides a way for JavaScript to interact with HTML elements, modify their content, attributes, and styles, and handle events.

DOM Tree


Document
Root element: <html>
Element: <head>
Element: <body>
Element: <title>
Attribute: "href"
Element: <a>
Element: <h1>
Text: "My title"
Text: "My link"
Text: "My header"

DOM Tree


document
Root element:
<html>
Element:
<head>
Element:
<title>
Text:
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribute:
href
Text:
"Link text"
DOM
Document Object Model

Core Concepts

- **Nodes:** The fundamental building blocks of the DOM, representing elements, attributes, text, comments, and document fragments.
- **Tree Structure:** The DOM organizes nodes in a hierarchical tree structure, where the root node is the `document` object.
- **Properties and Methods:** Nodes have properties and methods that allow you to access and manipulate their content, attributes, and relationships with other nodes.

Document Object

- **Definition:** The `document` object is the root node of the DOM tree. It represents the entire HTML document.
- **Properties and Methods:** The `document` object provides numerous properties and methods to access and manipulate various aspects of the document, including:
- **`getElementById()`:** Retrieves an element by its ID.
- **`getElementsByTagName()`:** Retrieves a collection of elements by their tag name.
- **`getElementsByClassName()`:** Retrieves a collection of elements by their class name.

Document Object methods

- **`querySelector()`:** Retrieves the first element matching a specified CSS selector.
- **`querySelectorAll()`:** Retrieves a collection of elements matching a specified CSS selector.
- **`createElement()`:** Creates a new element.
- **`createTextNode()`:** Creates a new text node.
- **`appendChild()`:** Adds a new child node to an element.
- **`removeChild()`:** Removes a child node from an element.
- **`insertBefore()`:** Inserts a new child node before an existing child node.

Document Object methods

- **`replaceChild()`:** Replaces an existing child node with a new child node.
- **`write()`:** Writes content to the document.
- **`writeln()`:** Writes content to the document with a newline character.
- **`cookie`:** Accesses and manipulates cookies.
- **`location`:** Accesses and manipulates the current URL.
- **`history`:** Accesses and manipulates the browser's history.

getElementByID()

- Retrieves an element from the HTML document based on its unique ID attribute.

Syntax: document.**getElementById**(id);

- IDs must be unique within an HTML document.
- The `getElementById()` method is case-sensitive.
- If no element with the specified ID is found, the method returns `null`.
- Once you have a reference to an element, you can access its properties, modify its attributes, and handle events associated with it.

querySelector() in Js

- Retrieves the first element in the document that matches a specified CSS selector.
- **Syntax** - document.querySelector(selector);
- CSS selectors can be simple or complex, allowing you to target elements based on various criteria, such as tag name, class, ID, attributes, and relationships.
- `querySelector()` only returns the first matching element. If you need to select multiple elements, use **`querySelectorAll()`**.
- The **`querySelector()`** method is case-sensitive.
- If no matching element is found, the method returns **`null`**.

Change Text Using Js

`textContent`

- **Definition**: This property sets or gets the text content of an element and all its descendants. It returns the text as plain text, without any HTML tags.
- **Usage**: It is faster than `innerText` and does not trigger a reflow.

`innerText`

- **Definition**: This property sets or gets the visible text content of an element. It respects the styling of the page, meaning it will only return text that is visible to the user.
- **Usage**: It triggers a reflow and is generally slower than `textContent`.

Change the Styling Using Js

- **Directly Modifying the `style` Attribute:**
- Access the element's `style` property.
- Set individual CSS properties using camelCase notation.

```
const element = document.getElementById("myElement");
element.style.backgroundColor = "blue";
element.style.color = "white";
```

Get and Set Attribute in Js

- **`getAttribute(attributeName)`:** Retrieves the value of the specified attribute.
- **`setAttribute(attributeName, attributeValue)`:** Sets the value of the specified attribute.

```
const element = document.getElementById("myElement");
```

- **Get the value of the "title" attribute**

```
const title = element.getAttribute("title");
```

- **Set the value of the "class" attribute**

```
element.setAttribute("class", "newClass");
```

Key Points

- **Key Points:**
- Attributes are case-sensitive.
- If the attribute doesn't exist, **`getAttribute()`** returns **`null`**.
- Setting an attribute that already exists updates its value.
- You can use **`removeAttribute()`** to remove an attribute.

Get Multiple HTML element

Type of Collection:

- **NodeList**: A collection of nodes that can include elements, text nodes, and comments. It can be returned by methods like **`querySelectorAll()`**.
- **HTMLCollection**: A specialized collection of only HTML elements. It is returned by methods like **`getElementsByClassName()`** or **`getElementsByTagName()`**.

Live vs. Static:

- **NodeList**: Can be either live (e.g., returned by `childNodes`) or static (e.g., returned by **`querySelectorAll()`**).
- **HTMLCollection**: Always live, meaning it updates automatically when the DOM changes.

Difference Between NodeList and HTMLcollection

- Both are Array like Object (have indexing and length property)
- Apply Loop on both
- Simple for Loop
- *For..of* Loop
- forEach loop

- Convert both in Array using **Array**.**from()**

InnerHTML Porperty

- Gets or sets the HTML content of an element.

<u>Syntax</u>

- element.innerHTML; // **Gets** the **HTML** content
- element.innerHTML = newHTML; // **Sets** the **HTML** content

Play-with DOM

- getroot()
- ChildNodes
- ParentNodes
- childElementCount
- HasChildNodes
- LastChild
- Firstchild
- Nextsibling
- PreviousSibling
- ParentElement
- NextElementSibling
- PreviousElementSibling

classList Property in JavaScript

- Manages the class attributes of an element.
- **Syntax** - element.**classList**; // Accesses the classList object
- **`add(className)`:** Adds a class to the element.
- **`remove(className)`:** Removes a class from the element.
- **`toggle(className)`:** Toggles a class on or off, depending on whether it's already present.
- **`contains(className)`:** Checks if the element has the specified class.
- **`item(index)`:** Retrieves the class at the specified index.
- **`length`:** Returns the number of classes on the element.

Key Points

- `classList` is a dynamic property that reflects the current class attributes of the element.
- You can use these methods to dynamically add, remove, or toggle classes based on conditions or user interactions.
- Multiple classes can be applied to an element, separated by spaces.
- `classList` provides a convenient way to manage styles without directly manipulating the **`className`** attribute.

Create Your Own Methods and this Keyword

```
let obj1 = {
  name: "John",
  age: 25,
  greet: function () {
    console.log(`Hello, my name is ${this.name} and my age is
${this.age}`);
  },
};
obj1.greet();
```

What is this here ? 🙄 🤔

Definition:

- `this` refers to the context in which the current code is executing. Its value is determined based on how the function is called.

Important Points:

- Dynamic and can change.
- Not assigned a value until the function where it is used is called.
- In a browser, `this` refers to the global `window` object.
- In Node.js, it refers to the global object.

Method Context (`this` in Methods)

- When a method is called on an object, `this` refers to the object itself.

Calling a function for Various Objects and setting it a value of a key(method)

```
function PrintData() {
 console.log(` Hello, my name is ${this.name} and my age is ${this.age}` );
 console.log(this);
}
let person1 = {
 name: "Brian",
 age: 30,
 about: PrintData,
};
let person2 = {
 name: "John",
 age: 25,
 about: PrintData,
};
```

Call, Apply and Bind Method in Js

The `call()`, `apply()`, and `bind()` methods in JavaScript allow you to explicitly set the value of `this` inside a function. These methods are especially useful when borrowing methods from one object and using them in the context of another.

Purpose:

- All three methods allow setting the value of `this` explicitly within a function, but they differ in how they pass arguments and when the function is executed.

Key Points:

- `call()` – Calls the function immediately with arguments passed individually.
- `apply()` – Calls the function immediately with arguments passed as an array.
- `bind()` – Returns a new function with a bound `this` value; doesn't call the function immediately.

Call Syntax - **functionName**.call(thisArg, arg1, arg2, arg3, ...)

Apply Syntax - **functionName**.apply(thisArg, **[arg1, arg2, arg3, ...]**)

Bind Syntax - **functionName**.bind(thisArg, arg1, arg2, arg3, ...)

Example Object

```
function about(hobby, role) {
  console.log(`My name is ${this.firstName + " " + this.lastName}, hobby is ${hobby}`);
  console.log(`My Role is ${role}`);
}
const user = {
  firstName: "Lokesh",
  lastName: "Singh",
  age: 25,
};
const user2 = {
  firstName: "Aniket",
  lastName: "Singh",
  age: 28,
};
```

Call Method

- The `call()` method calls a function with a given `this` value and arguments provided individually.
- **Use Case:** Borrowing a method from one object and using it for another.

You can call the function like this

- user.about("playing","Trainer")

Using Call Method

- about.call(user,"playing","Trainer")
- about.call(user2, "Dancing", "Developer");

Apply Method

- The **`apply()`** method is similar to **`call()`**, but it takes arguments as an array.
- **Use Case:** Passing an array of arguments instead of listing them individually.

Example

```
about.apply(user,["Playing","Coding Trainer"])
about.apply(user2,["Dancing","App Developer"])
```

Bind Method

- The **`bind()`** method creates a new function where the value of `this` is permanently bound to the first argument passed to **`bind()`**. It doesn't invoke the function immediately.
- **Use Case:** When you need a function to be called later with a specific `this` value.

Example

- const user1Data = about.bind(user, "Playing", "Coding Trainer");
- const user2Data = about.bind(user2, "Dancing", "Developer");
- user1Data()
- user2Data()

Comparison: call() vs apply() vs bind()

Feature	call()	apply()	bind()
Invocation	Calls the function immediately	Calls the function immediately	Returns a new bound function (not called immediately)
Arguments	Passed individually	Passed as an array	Passed individually when function is invoked later
Usage	When arguments are known	When arguments are in an array	When you need to preset this for later use

__proto__ or [[prototype]]

- The __proto__ property in JavaScript is a way to access or set the prototype of an object. Every object in JavaScript has an internal link to another object called the "prototype," which allows for property inheritance. The __proto__ property gives access to this internal link.

Example

```
let obj1 = {
 firstName: "Lokesh",
 lastName: "Singh",
};
```

Creating Obj2 from Obj1

```
let obj2 = Object.create(obj1);
```

Synchronous and Asynchronous

Synchronous:

- Code is executed line by line, and each statement waits for the previous one to finish before executing.

Asynchronous:

- Code allows certain operations (like fetching data or waiting for a timer) to happen in the background, freeing up the main thread for other tasks. The program doesn’t wait for the completion of tasks before moving to the next one.

Synchronous Code Execution

How it works:

- In synchronous execution, tasks are executed one after another in a blocking manner. The next line of code is only executed after the current line completes.

Example

- console.log('Start');
- console.log('Middle');
- console.log('End');

Error Handling

- **What is Error Handling?** Error handling refers to the process of responding to and managing errors that arise during the execution of a program.

- **Why is it Important?**
  - Ensures the program doesn’t crash unexpectedly.
  - Provides feedback to users when something goes wrong.
  - Helps developers debug issues.

Types of Errors in JavaScript

- **Syntax Errors**
- Occurs when there is a mistake in the code’s syntax.

Example: `console.log('Hello)` (missing closing quote)

- **Runtime Errors**
- Occurs when the script is executed, and the code cannot be executed as intended.

Example: Trying to access an undefined variable.

- **Logical Errors**
- The code runs without issues but produces incorrect results.

Example: Incorrect calculations or conditions.

JavaScript Error Objects

- JavaScript has built-in error objects that represent different types of errors:
- **Error** – Generic error object.
- **SyntaxError** – Thrown when there’s a syntax error in the code.
- **ReferenceError** – Occurs when a variable that doesn't exist is referenced.
- **TypeError** – Thrown when a variable is of the wrong type.
- **RangeError** – Happens when a number is out of range.
- **Custom Errors** – You can create your own error types.

Error Handling with `try...catch`

- The `try...catch` block is used to handle errors in JavaScript.

```
try {
  let result = 10 / 0;
  console.log(result);
} catch (error) {
  console.log('An error occurred: ' + error.message);
}
```

Using `finally`

- The `finally` block is executed regardless of whether an error was thrown or not. It is typically used for clean-up actions.

```
try {
 // Code that may throw an error
} catch (error) {
 // Code to handle the error
} finally {
 // Code to always execute (e.g., closing a file)
}
```

Throwing Errors with `throw`

- You can manually throw an error using the `throw` statement.
- Syntax - **throw** new Error('Custom error message');

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero is not allowed.');
  }
  return a / b;
}
try {
  divide(4, 0);
} catch (error) {
  console.log(error.message);
}
```

How Js Code Executes(Synchronous)

Before Execution of Code there is 2 steps which is done.

- Early error cheching(Before execution of code the program must be error free)
- Scope determination of variables.

After above 2 steps are done correclty then comes the execution part

- Creation of Global Execution Context(GEC) which has two phases
  - Global memory
  - Code Execution phase
- Now GEC added to Stack for code execution

Asynchronous Code Execution

How it works:

- In asynchronous execution, tasks are executed independently, and the program does not wait for the completion of a task before moving on. Once an asynchronous task is finished, its result is processed later.

```
console.log('Start');
setTimeout(() => {
 console.log('Delayed message');
}, 2000);
console.log('End');
```

SetTimeout()

Definition: Executes a function **once** after a specified delay.

Syntax:

```
let timeoutId = setTimeout(function, delay);
```

- `function`: The function to execute.
- `delay`: Time in milliseconds before execution (1 second = 1000 milliseconds).

SetTimeout()

Definition: Executes a function **once** after a specified delay.

Syntax:

```
let timeoutId = setTimeout(function, delay);
```

- `function`: The function to execute.
- `delay`: Time in milliseconds before execution (1 second = 1000 milliseconds).

```
setTimeout(() => {
  console.log('Hello, after 3 seconds!');
}, 3000);
```

SetInterval()

Definition: Executes a function **repeatedly** after a specified delay.

Syntax:

```
let intervalId = setInterval(function, delay);
```

- `function`: The function to execute.
- `delay`: Time in milliseconds between execution (1 second = 1000 milliseconds).

```
setInterval(() => {
  console.log('Hello, after every 2 seconds!');
}, 2000);
```

clearTimeout()

Definition: Cancels a `setTimeout()` before it executes.

Syntax:

```
clearTimeout(timeoutId);
```

`timeoutId`: The ID returned by `setTimeout()` when it was created.

```
let timeoutId = setTimeout(() => {
  console.log('This will not run!');
}, 5000);

clearTimeout(timeoutId); // Cancels the timeout
```

clearInterval()

Definition: Stops a `setInterval()` from executing further.

Syntax:

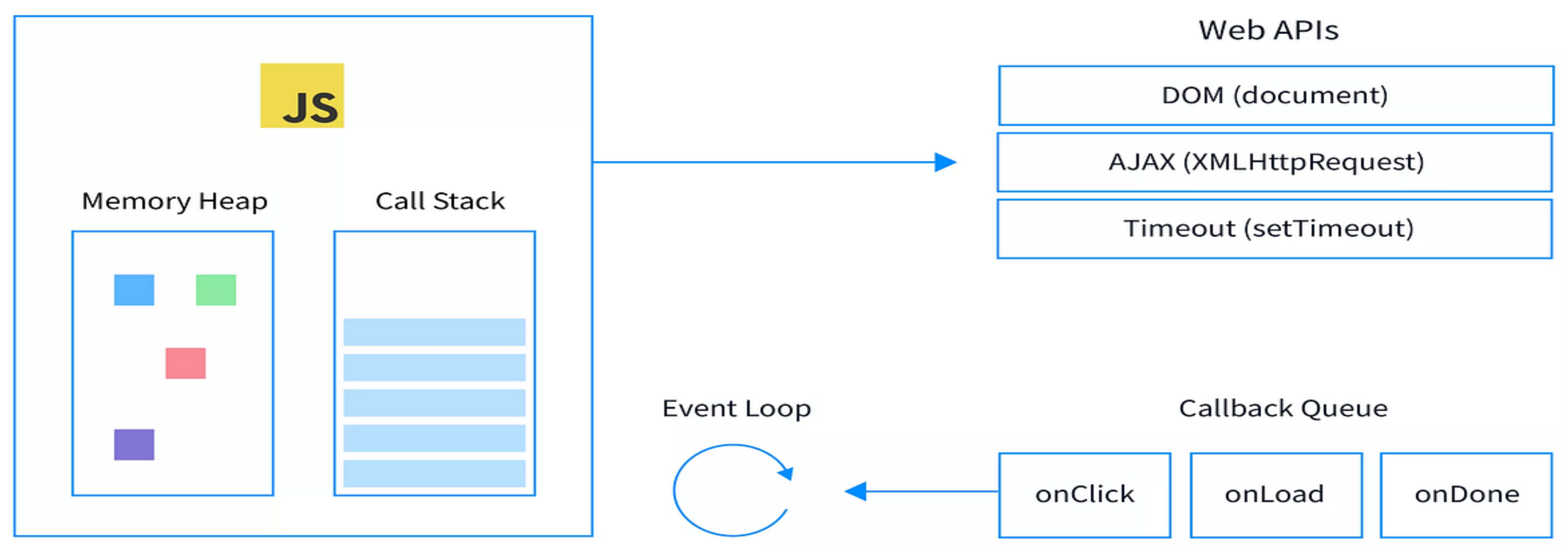
```
clearInterval(timeoutId);
```

`timeoutId`: The ID returned by `setInterval()` when it was created.

clearInterval() - example

```
let count = 0;
let intervalId = setInterval(() => {
  console.log('Running...');
  count++;
  if (count === 5) {
    clearInterval(intervalId); // Stops the interval after 5 runs
  }
}, 1000);
```

Js Asynchronous Code Execution



JavaScript Promises

- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- A value that I don't know yet. Will be known in future

A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

Creation and consumption of Promise

-
  ```
  let myPromise = new Promise(function(myResolve, myReject) {
  // "Producing Code" (May take some time)

    myResolve(); // when successful
    myReject();  // when error
  });

  // "Consuming Code" (Must wait for a fulfilled Promise)
  myPromise.then(
    function(value) { /* code if successful */ },
    function(error) { /* code if some error */ }
  );
  ```

Promise Example - 2

```
let bucket = ["paneer", "masala", "", "rice", "tea", "haldi"];


let paneerBiryani = new Promise((resolve, reject) => {
if (bucket.includes("salt") &&bucket.includes("masala") &&bucket.includes("rice"))
{
    resolve({ fullfill: "Come and Eat" });
  } else {
    reject("Can't do it");
  }
});
```

Promise Consumption Method - 1

```
- Promise
.then((value)=>{})
.catch((value)=>{})
-  Example:
paneerBiryani.then((value)=>{
    console.log(value);
})
.catch((value)=>{
    console.log(value);
})
```

Promise Consumption Method

- `promise.then(()=>{},()=>{})`

```
paneerBiryani.then((value)=>{
    console.log(value);
},(value)=>{
    console.log(value);
})
```

Function Returning Promise

```
let paneerBiryani = () => {
let bucket = ["paneer", "masala", "salt", "rice", "tea", "haldi"];
 return new Promise((resolve, reject) => {
if (bucket.includes("salt") && bucket.includes("masala") && bucket.includes("rice")) {
       resolve({ fullfill: "Come and Eat" });
     } else {
       reject("Can't do it");
     }
   });
 };
```

Consumption

```
paneerBiryani().then(()=>()).catch((error) => ());
```

Promise and SetTimeOut

```
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function() { myResolve("I love You !!");
}, 3000);
});

myPromise.then(function(value) {
  console.log(value);
});
```

Promise.resolve() and Promise.reject()

```
Promise.resolve(5)
  .then((val) => {
    console.log(val,"Hello");
  })
  .catch((val) => {
    console.log(val);
  });
```

Promise Chaining

`Promise Chaining -> .then() method Always returns a Promise so we can use .then method over it.`

- **Promise chaining** is a technique to execute multiple asynchronous operations **one after another**.
- Each `.then()` passes its result to the next `.then()` in the chain.

Why Use Promise Chaining?

- **Readable Code**: Avoids callback hell.
- **Sequential Execution**: Ensures operations run in order.
- **Error Handling**: Simplifies error management using `.catch()`.

How promise chaining works ?

- A promise is created and resolved.
- The `.then()` block processes the result and can return a new promise.
- The next `.then()` processes the result of the previous one.

Example

- Imagine a task:
- Boil water.
- Add tea leaves.
- Serve tea.

Code Example:

```
boilWater()
  .then((water) => addTeaLeaves(water))
  .then((tea) => serveTea(tea))
  .catch((error) => console.error("Something went wrong:", error));
```

Key Points

- **Each `.then()` returns a promise.**
  - This allows chaining the next `.then()` block.
- **Error Handling in Chains:**
  - Use `.catch()` to handle errors from any step in the chain.

What is an API?

API stands for **Application Programming Interface**.

It allows different software applications to communicate with each other.

Simple Example:

- Ordering food in a restaurant:
    - **Menu**: API (it tells what is available and how to ask for it).
    - **Order**: Request to the kitchen (server).
    - **Food**: Response from the server.

Why Are APIs Important?

- **Connect Services**: APIs connect web applications, databases, and devices.
- **Share Data**: Access data like weather, maps, or social media posts.
- **Simplify Development**: Developers use existing APIs instead of building features from scratch.

What is the Fetch Method?

- The `fetch()` **method** is a built-in JavaScript function(and Promise) for making API requests.
- **Key Features:**
- Works with **APIs** to fetch data from servers.
- Returns a **Promise** (handles asynchronous operations).
- Supports modern syntax and is easy to use.

How Does fetch() Work?

- **Make a Request**: Send a request to an API using `fetch(url)`.
- **Get a Response**: The server responds with data.
- **Process the Data**: Use `.then()` to handle the response.

Benefits of `fetch()`

- **Asynchronous**: Non-blocking; doesn’t freeze the app while waiting.
- **Flexible**: Supports various HTTP methods like `GET`, `POST`, `PUT`, and `DELETE`.
- **Modern Syntax**: Works with Promises for clean and readable code.

Code Example

// Fetch data from a public API

```
fetch('https://api.example.com/data')
  .then((response) => response.json()) // Convert response to JSON
  .then((data) => console.log(data))   // Process the data
  .catch((error) => console.error('Error:', error)); // Handle errors
```

When to Use APIs and Fetch

- To access third-party services (e.g., Weather APIs, Google Maps).
- To communicate with your backend server.
- To dynamically update content on a webpage.

Real-Life Use Cases

- **Weather App**: Fetch real-time weather data.
- **E-commerce**: Retrieve product details from a server.
- **Social Media**: Display posts from APIs like Facebook or Twitter.

Async/Await in JavaScript

What is Async/Await?

- **`async/await`** is a modern way to handle asynchronous code in JavaScript.
- It makes code easier to read and write compared to Promises or callbacks.

Why Use Async/Await?

- **Simplifies Syntax**: Looks like synchronous code, even for async operations.
- **Readable Code**: Avoids nested `.then()` chains in Promises.
- **Better Error Handling**: Use `try/catch` for errors.

How Does Async/Await Work?

1. `async` **Keyword**

- Declares a function as asynchronous.
- Automatically returns a Promise.

2. `await` **Keyword**

- Pauses execution until the Promise resolves.
- Must be used inside an `async` function.

Code Example: Without Async/Await

Fetching user data from an API using Promises:

```
fetch('https://api.example.com/user')
  .then((response) => response.json())
  .then((user) => console.log(user))
  .catch((error) => console.error(error));
```

Code Example: With Async/Await

The same task becomes cleaner and easier:

```
async function fetchUser() {
  try {
    const response = await fetch('https://api.example.com/user');
    const user = await response.json();
    console.log(user);
  } catch (error) {
    console.error('Error:', error);
  }
}
fetchUser();
```

Key Benefits of Async/Await

- **Sequential Execution**: Code execution flows top to bottom.
- **Easier Debugging**: Debugging async code is as simple as synchronous code.
- **Improved Error Handling**: Use `try/catch` instead of `.catch()` for cleaner code

Real-Life Analogy

- Imagine placing an order at a restaurant:
    - **Order**: `await` the server to deliver food.
    - Once food arrives, continue enjoying the meal!

Best Practices

- Always use try/catch to handle errors.
- Avoid blocking with too many await calls; use them efficiently.
- Combine with Promise.all for parallel async operations when needed.

Comparison: Promises vs Async/Await

Feature	Promises	Async/Await
Syntax	Chained .then() and .catch()	Cleaner and more readable
Readability	Can become complex with nested .then() calls	Synchronous-like flow; easier to understand
Error Handling	Requires .catch() for errors	Use try/catch blocks for better clarity
Debugging	Slightly harder to trace errors in chained calls	Easier due to synchronous-like structure
Code Length	Often longer due to chaining and callbacks	Shorter and more concise
Parallel Execution	Use Promise.all() explicitly	Works with Promise.all() for parallel tasks
Use Case	Suitable for simple async tasks or quick chaining	Better for complex workflows with multiple steps
Example	Example with .then() chaining:	Example with async/await: