**Javascript**

JavaScript is a High level programming language that is primarily used to enhance the interactivity and dynamic behavior of websites.

JavaScript is a lightweight, cross-platform, single-threaded, and High level interpreted compiled programming language. It is also known as the scripting language for webpages.

### .**Lightweight**

* **Meaning:** JavaScript is considered lightweight because it has a simple syntax and doesn't require a large amount of system resources to run.

### **Cross-Platform**

* **Meaning:** JavaScript is cross-platform, meaning it can run on various operating systems and devices. Whether it's a Windows PC, a Mac, a Linux system, or a mobile device, JavaScript code can be executed as long as the environment has a JavaScript engine, such as web browsers (Chrome, Firefox, Safari)

### **Single-Threaded**

* **Meaning:** JavaScript operates on a single thread, meaning it can perform one operation at a time per thread. This is often referred to as the "JavaScript event loop," where JavaScript handles operations sequentially. While JavaScript is single-threaded, it can handle asynchronous tasks (like network requests, timers, etc.) using callbacks, promises, and the async/await syntax, allowing it to perform tasks without blocking the main thread.

### **High-Level and Easy to Write**

* Scripting languages are usually high-level, meaning they are closer to human language and easier to read, write, and understand. They are often used to write code quickly and efficiently without needing complex syntax or understanding the underlying hardware.

### **Interpreted Compiled**

* **Meaning:** JavaScript is often referred to as an interpreted language because traditionally, it was executed directly by the browser without prior compilation. modern JavaScript engines (like V8 in Chrome) use Just-In-Time (JIT) compilation, which compiles JavaScript code to machine code at runtime, improving performance. Thus, it's both interpreted (in a traditional sense) and compiled (due to JIT).

### **Scripting Language for Webpages**

* **Meaning:** JavaScript was initially created as a scripting language for the web. Its primary use was to add interactivity to HTML pages. Over time, it has evolved significantly but remains the core technology for client-side web development.

**The History of JavaScript**

**1. Origins and Creation (1995)**

* **Who Introduced JavaScript:** JavaScript was created by Brendan Eich, a programmer working at Netscape Communications Corporation.
* **Development Timeline:** Brendan Eich developed the first version of JavaScript in just 10 days in May 1995.
* **Company:** Netscape, the company that hired Eich, was a pioneer in the early web browser market. At that time, Netscape was in direct competition with Microsoft to dominate the browser space.

**2. Early Naming and Initial Implementation**

* **Initial Name - Mocha:** The language was initially called *Mocha*. This name was chosen by Marc Andreessen, one of the co-founders of Netscape.
* **Renaming to LiveScript:** Before its official launch, the language was renamed *LiveScript*. This name was intended to emphasize its dynamic capabilities in web pages, distinguishing it from static HTML.
* **Final Renaming to JavaScript:** In December 1995, Netscape changed the language’s name to *JavaScript*. This was part of a marketing collaboration with Sun Microsystems. The name JavaScript was chosen to leverage the popularity of Java at the time, even though JavaScript and Java are fundamentally different languages.

**3. Relationship with Java and the Role of Sun Microsystems**

* **Java vs. JavaScript:** Despite the name similarity, Java and JavaScript serve different purposes and have distinct syntaxes. Java is a statically typed, compiled language, whereas JavaScript is a dynamically typed, interpreted language.
* **Sun Microsystems’ Involvement:** Sun Microsystems, the creators of Java, and Netscape agreed to rename LiveScript to JavaScript to associate the new scripting language with Java. This move helped market JavaScript as a complementary technology to Java, which was popular among developers at that time.

**4. Microsoft's Response: Introduction of JScript**

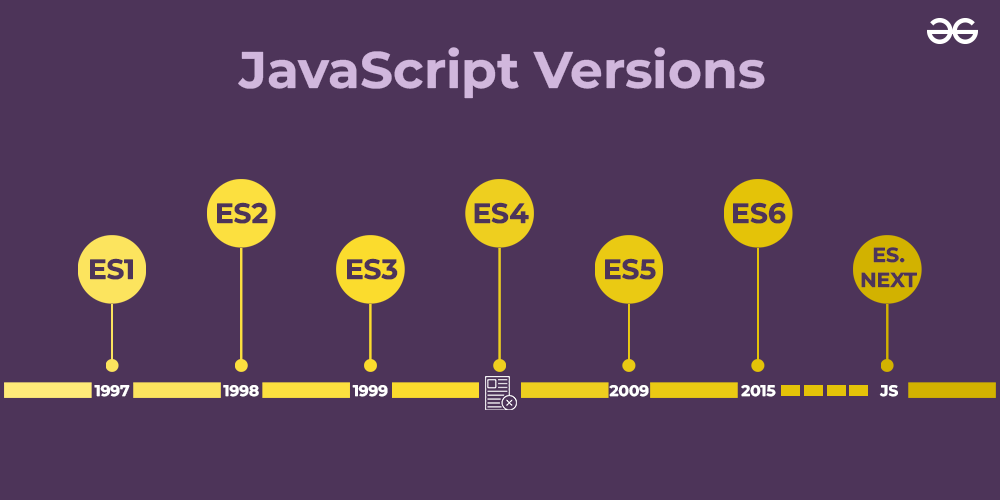
* **Launch of JScript:** In response to JavaScript’s growing popularity, Microsoft developed its own implementation of JavaScript called *JScript* in 1996.
* **Compatibility:** JScript was designed to be compatible with JavaScript but included certain proprietary features and enhancements. Microsoft integrated JScript into Internet Explorer 3.0, their web browser, to compete with Netscape Navigator.
* **Browser Wars:** This period marked the beginning of the first browser wars, with Netscape and Microsoft vying for dominance in the web browser market.

**5. Standardization: ECMAScript**

* **Why Standardization was Necessary:** The existence of different implementations (JavaScript by Netscape and JScript by Microsoft) created inconsistencies and compatibility issues across browsers.
* **Submission to ECMA:** In November 1996, Netscape submitted JavaScript to the European Computer Manufacturers Association (ECMA) for standardization.
* **Creation of ECMAScript:** In June 1997, ECMA published the first edition of ECMAScript, a standardized specification for JavaScript. ECMAScript provided a standard that all scripting languages like JavaScript and JScript could conform to, ensuring greater compatibility and consistency across different web browsers.
* **Role of ECMA-262:** ECMAScript is often referred to as ECMA-262, which is the name of the standard specification document. All major browsers began aligning their JavaScript engines to follow the ECMAScript standard.

**6. Evolution of JavaScript and ECMAScript Versions**

* **ECMAScript 2 and 3:** After the first edition, ECMAScript 2 (June 1998) and ECMAScript 3 (December 1999) were released, with ES3 becoming widely implemented and serving as the base for JavaScript development for many years.
* **ECMAScript 4:** There was an attempt to create ECMAScript 4 with major enhancements, but it was abandoned due to disagreements within the industry.
* **ECMAScript 5 (ES5):** Released in 2009, ES5 included important features such as strict mode, JSON support, and more. ES5 was a significant improvement over ES3 and was widely adopted.
* **ECMAScript 6 (ES6/ECMAScript 2015):** Released in 2015, ES6 introduced major enhancements, including arrow functions, classes, modules, template literals, let and const keywords, promises, and many more. ES6 marked a major evolution in JavaScript, making it more powerful and easier to work with.



**7. Modern JavaScript: ECMAScript Evolution and Community Involvement**

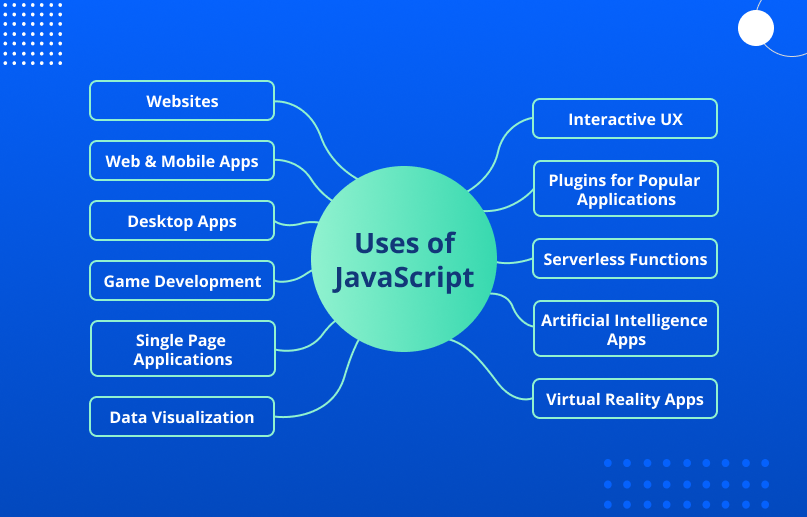
* **Annual Release Cycle:** Since ECMAScript 2015, the TC39 committee, which oversees the evolution of ECMAScript, adopted an annual release cycle. Each year, a new version of ECMAScript is released with incremental updates and new features.
* **ES7 (ES2016), ES8 (ES2017), ES9 (ES2018), etc.:** These versions introduced features like the exponentiation operator, async/await, Object.values/Object.entries, and more.
* **Community and Open-Source Influence:** The JavaScript community, including developers and companies, actively contributes to the evolution of JavaScript through proposals, feedback, and implementation in various environments (browsers, Node.js, etc.).

**8. JavaScript's Role Today**

* **Ubiquity:** JavaScript is a core technology of the World Wide Web, alongside HTML and CSS. It runs on all modern web browsers and is essential for creating interactive web applications.
* **Beyond the Browser:** JavaScript’s usage has extended beyond the browser, thanks to environments like Node.js. It is used for server-side programming, mobile app development (e.g., React Native), desktop apps (e.g., Electron), and more.
* **Frameworks and Libraries:** JavaScript has a rich ecosystem of frameworks and libraries like React, Angular, Vue, and jQuery, which make it easier to build complex and responsive web applications.

Some of the common uses of JavaScript are:

1. **Web Development:** JavaScript is primarily used for creating interactive and dynamic web pages and web applications. It can be used for tasks such as form validation, animations, user interface enhancements, and more.
2. **Mobile App Development:** JavaScript can be used to create mobile applications that can run on both iOS and Android devices using frameworks such as React Native and Ionic.
3. **Game Development:** JavaScript is used to create web-based games, such as HTML5 games, that can be played on any device with a web browser.
4. **Server-Side Development:** JavaScript can be used to create server-side applications using Node.js, which is a platform that allows developers to run JavaScript on the server side.
5. **Desktop Application Development:** JavaScript can be used to create desktop applications using frameworks such as Electron, which allows developers to build cross-platform desktop applications using web technologies.
6. **Internet of Things (IoT):** JavaScript can be used to program IoT devices, such as sensors and smart home devices, using frameworks such as Johnny-Five and Cylon.js.



## JavaScript: the Swiss army knife trap | by Tiago Sansão | Medium

## **How JavaScript Works: JS Engine & Runtime environment**

## JavaScript in the Browser:

## When you visit a webpage, your browser (like Chrome, Firefox, or Safari) loads the HTML and CSS to structure and style the page.

## If the page includes JavaScript (either directly in the HTML file or linked as a separate .js file), the loads and runs that JavaScript code using its built-in JavaScript engine (like V8 in Chrome or SpiderMonkey in Firefox).

## JavaScript Engine:

## What It Is: A JavaScript engine is a program inside the browser that reads and executes JavaScript code. Every major web browser has a built-in JavaScript engine.

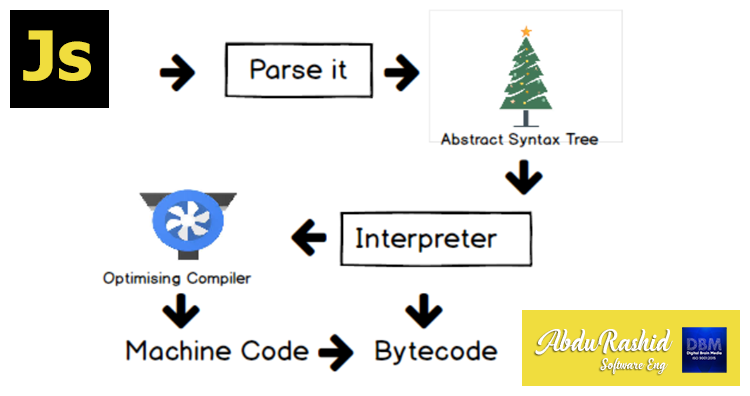
## Examples: V8 (Chrome, Node.js), SpiderMonkey (Firefox), Chakra (Edge), and JavaScriptCore (Safari).

## Execution Process:

## Parsing: The JavaScript engine first reads (or "parses") the JavaScript code. It checks the syntax to make sure everything is written correctly.

## Compilation: Unlike some languages that are compiled ahead of time (like C++), JavaScript is often compiled just before or even during its execution, in a process called Just-In-Time (JIT) compilation.

## Execution: After parsing and compiling, the engine runs the JavaScript code. It turns the code into machine-readable instructions that the computer's processor can understand and execute.

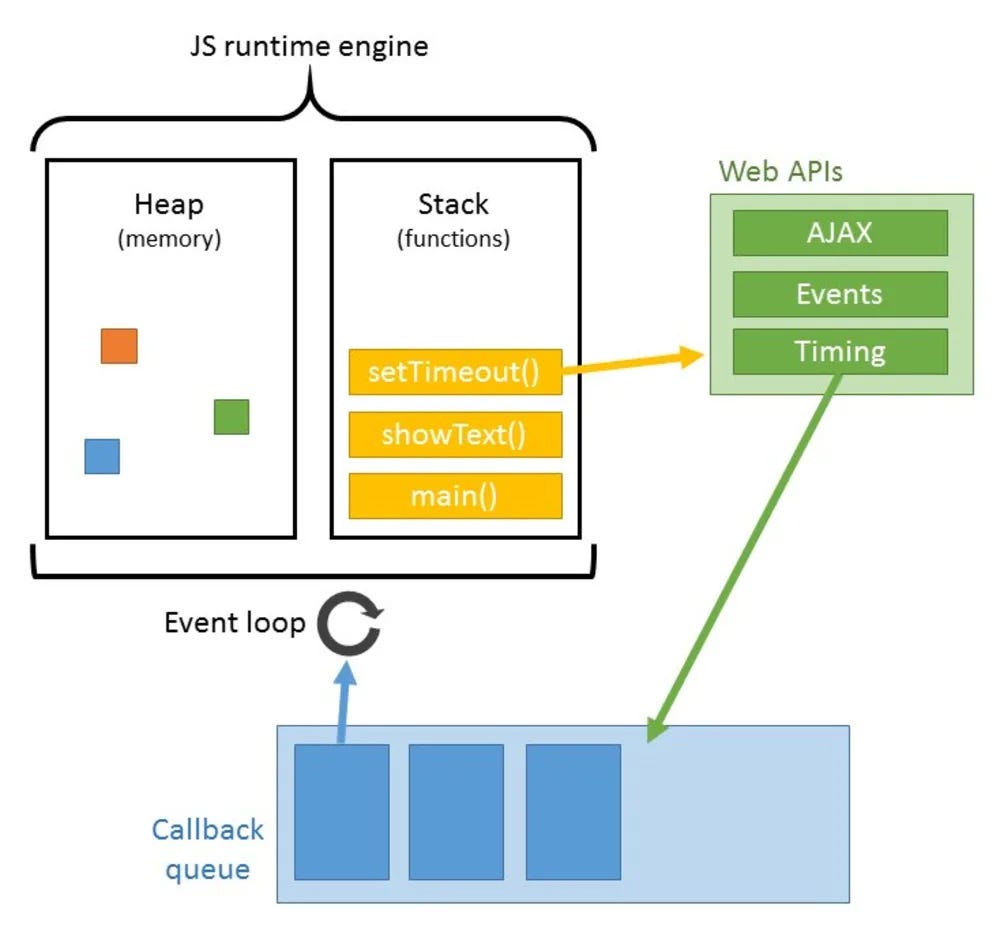


## Event Loop and Asynchronous Behavior:

## Single-Threaded: JavaScript runs in a single thread, meaning it can only do one thing at a time.

## Event Loop: To handle multiple tasks (like user inputs, timers, or fetching data from a server), JavaScript uses an event loop. The event loop constantly checks if there are tasks (events) to be executed. If there are, it runs them one by one.

## Asynchronous Tasks: JavaScript can perform certain tasks asynchronously, meaning it can start them, then continue with other work, and come back to finish the task once it’s done. Examples include making network requests, reading files, or waiting for user interactions.



## Interacting with the Webpage (DOM):

## DOM (Document Object Model): The DOM is a structured representation of the HTML elements on a webpage. JavaScript can interact with the DOM to change the content, structure, and style of the webpage dynamically.

## How It Works: JavaScript can select elements on the page (like buttons or input fields), change their properties (like text, color, or visibility), and respond to events (like clicks or form submissions).

## **How to Add JavaScript Code to a Html doc ?**

JavaScript code can be added to an HTML web page in three ways:

1. **Internal JavaScript** contains all JavaScript code within the body of the HTML script using the script> tag.

# <script>

# **alert("Hello PrepBytes!");**

# </script>

2) **Inline JavaScript** refers to the practice of embedding JavaScript code within HTML event attributes, such as onclick, onload, onsubmit, etc. When the event is triggered

# <!DOCTYPE html>

# <html>

# <head>

# <title>Inline JavaScript Example</title>

# </head>

# <body>

# **<button onclick="alert('Hi, PrepBytes!')">Click</button>**

# </body>

# </html>

3) **External JavaScript**, on the other hand, involves storing the code in a separate .js file and it can be linked using the < script> tag with the "src" attribute in the < body> section of the HTML file. The greet() function is defined in the script.js file and is called when the button is clicked in the HTML file. The src attribute specifies the path to the external JavaScript file.

it allows you to reuse the script in multiple HTML pages

# <!DOCTYPE html>

# <html>

# <head>

# <title>External JavaScript Example</title>

# </head>

# <body>

# <button onclick="greet()">Click me</button>

# **<script src="script.js"></script>**

# </body>

# </html>

**Variables**

Variables in JavaScript are containers for storing data values. These values can be numbers, strings, arrays, objects, or any other data type. Think of variables as labeled boxes where you can store information to be used later in your code.

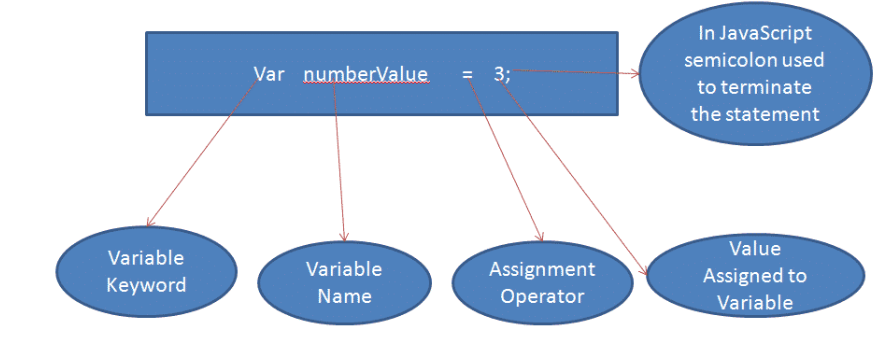
1. **Declaration**: Variables in JavaScript are declared using the **var**, **let**, or **const** keyword.
   * **var** has been traditionally used but has some scope-related issues.
   * **let** is block-scoped and is preferable for variable declaration.
   * **const** is also block-scoped but its value cannot be reassigned once it’s set.

      //variable declared with var keyword followed by a name & assigned a value using assignment operator

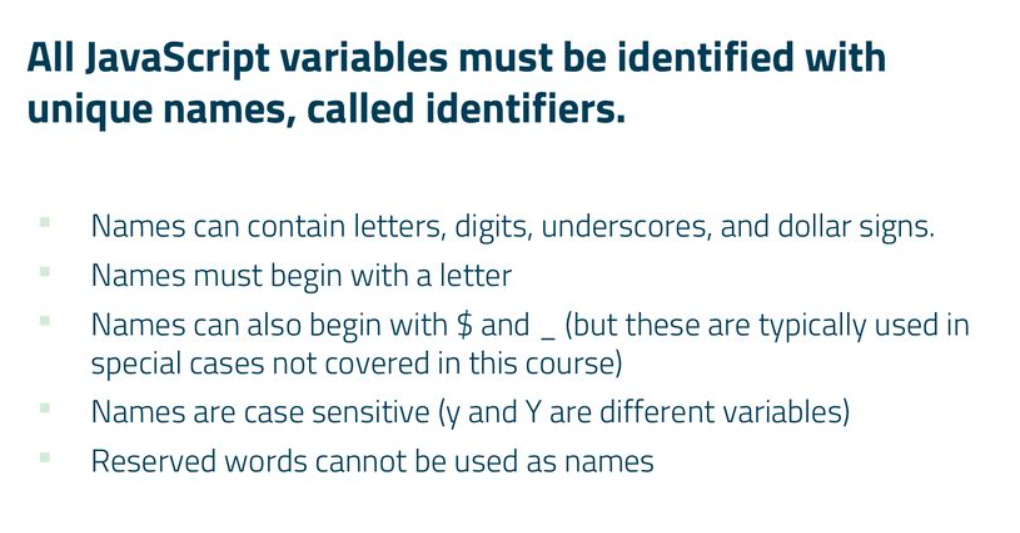
      var x = 10;

      let y = 20;

      const PI = 3.14;



1. **Naming Convention**: Variable names in JavaScript can contain letters, digits, underscores, and dollar signs. They must begin with a letter, underscore, or dollar sign



      var myVariable = 5;

      var \_myVariable = 10;

      var $myVariable = 15;

**Case Sensitivity**: JavaScript variable names are case-sensitive, meaning **myVariable** and **MyVariable** are treated as different variables.

      var myVariable = 5;

      var MyVariable = 10;

      console.log(myVariable); // Outputs: 5

      console.log(MyVariable); // Outputs: 10

1. **Data Types**: JavaScript variables can hold various data types including numbers, strings, objects, arrays, functions, etc.

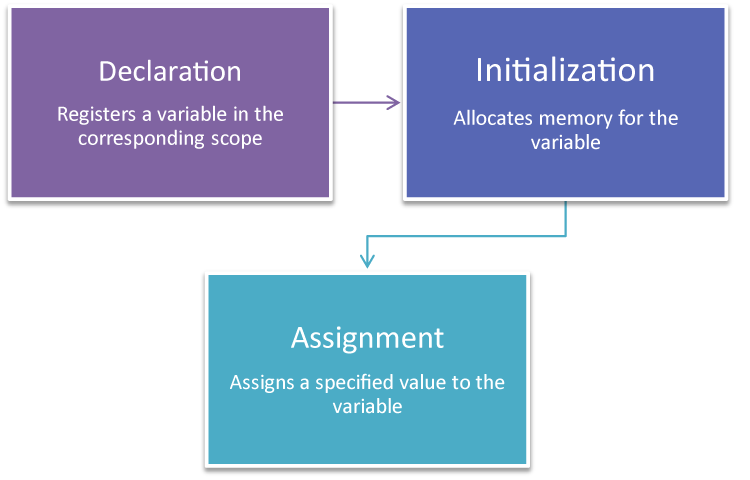
      var num = 5;

      var str = "Hello";

      var arr = [1, 2, 3];

      var obj = { name: "John", age: 30 };

1. **Dynamic Typing**: JavaScript is dynamically typed, meaning you don't have to specify the data type of a variable when declaring it. The data type of the variable is determined automatically at runtime.
2. **Variable terms**



**1. Declaration**

* **What it is**: Declaring a variable means telling the programming environment that a variable exists. This is essentially the act of defining the variable's name
* **Example**:

let myVariable;

In this example, myVariable is declared but not yet initialized with a value.

**2. Initialization**

* **What it is**: Initialization is the process of assigning an initial value to a variable at the time of declaration. This is when you first give the variable a specific value.
* **Example**:

let myVariable = 10;

Here, myVariable is both declared and initialized to the value 10.

**3. Assignment**

* **What it is**: Assignment refers to giving a variable a value, regardless of whether it’s being assigned for the first time or re-assigned to a new value after initialization.
* **Example**:

let myVariable;

myVariable = 10; // Assignment after declaration

myVariable = 20; // Re-assignment

In the first line, myVariable is declared. In the second line, it is assigned the value 10. In the third line, it is re-assigned the value 20.

1. **Scope**: Variables in JavaScript have function or block scope, depending on how they are declared.
   * Variables declared with **var** are function-scoped. They are accessible anywhere within the function they are declared in.
   * Variables declared with **let** or **const** are block-scoped. They are only accessible within the block they are declared in.
2. **Hoisting**: Variable declarations are hoisted to the top of their scope during the compilation phase, but their assignments remain where they are.

      myVar=5;

      console.log(myVar); // Output: 5

      var myVar;  //----This declaraation is moved to top even before the code execution—Hoisting

**Hoisting**

**In general terms Hoisting means ?**

Hoisting means lifting up. For example, when we hoist our flag or when a crane lifts something, it's hoisting(lifting up) that thing,

**Technically ?**

**1)** **Hoisting** is a behavior where the declarations of the variables and functions are moved to the top even before the execution.

Only the declarations are hoisted, not the initializations.

**2) var Declarations**:

 Variables declared with var are hoisted to the top of their scope.

 The variable is hoisted and initialized with undefined

console.log(myVar);   // Output: undefined

var myVar = 10;

console.log(myVar);   // Output: 10

In the example above, var myVar is hoisted to the top, but the assignment myVar = 10 is not. So, the first console.log outputs undefined.

**3) Function Declarations**:

Functions declared using the function keyword are hoisted entirely, both the declaration and the definition.

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

function myFunction() {

  console.log("Hello, world!");

}

**4)** **let and const Declarations**:

\* Variables declared with let and const are also hoisted, but they are not initialized.

\* These variables are in a "temporal dead zone" from the start of the block until the declaration is encountered.

console.log(myLet);   // ReferenceError: Cannot access 'myLet' before initialization

let myLet = 20;

console.log(myLet);   // Output: 20

In the example above, let myLet is hoisted, but not initialized. Accessing it before initialization results in a ReferenceError.

console.log(myConst); // ReferenceError: Cannot access 'myConst' before initialization

const myConst = 30;

console.log(myConst); // Output: 30

Similarly, const myConst is hoisted but not initialized. Accessing it before initialization results in a ReferenceError.

How variable works in different cases

**Variables :-** variables are containers used to store the data, we can declare variables with var, let and const

Programs on var in different cases.

//case - 1

//var a is "undefined" in the console because "value is not defined"

var a;

console.log(a);

//case - 2

//In output it shows "not defined" because "variable is not at all defined" in the program like this-> var b;

console.log(b);

//case - 3

//in output it shows "10" because value and variable both are defined

var c=10;

console.log(c);

//case - 4

//it shows the output "undefined" because assignment is not hoisted

var d;

console.log(d);

d=10;

//case - 5

//it shows the output "10" because of hoisting, In hoisting declarations are moved to top

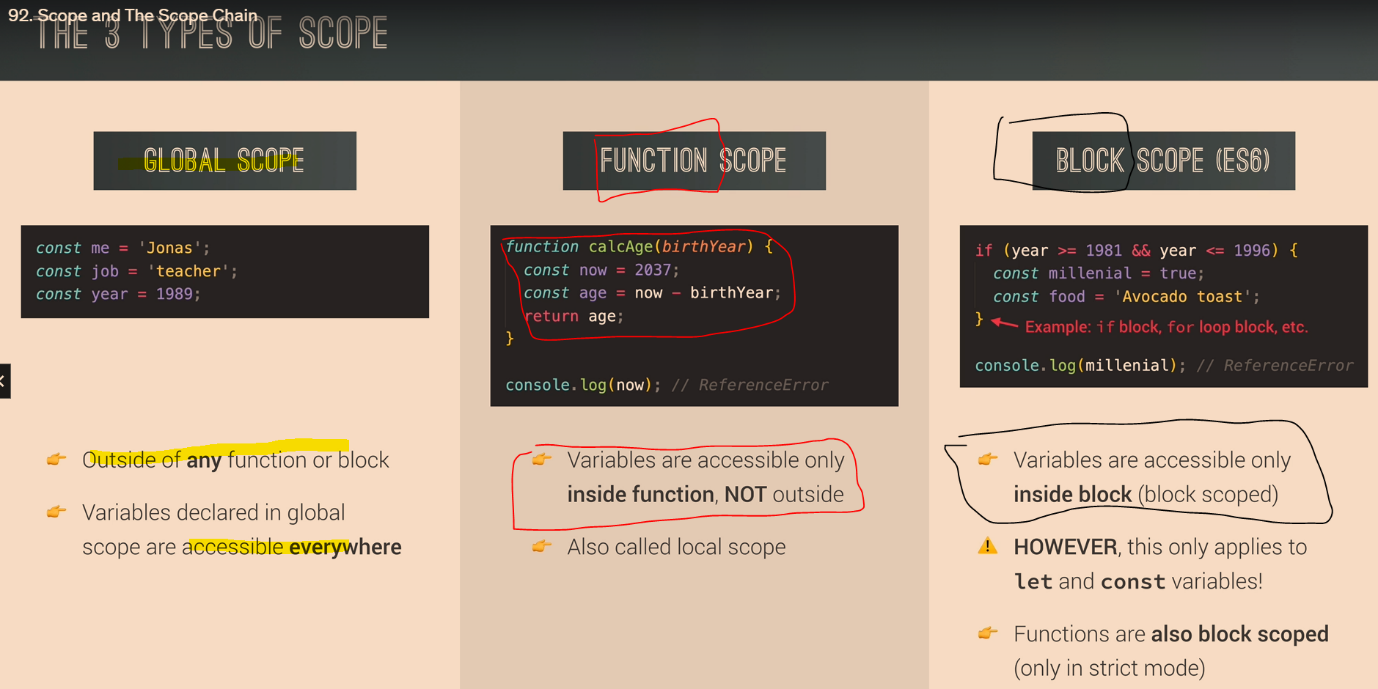
e=10;

console.log(e);

var e;

**Scopes in javascript**:

A scope in JavaScript defines the accessibility or life or visibility of variables and functions. When you declare a variable or a function, its scope determines where it can be accessed from within your code.



**Global Scope**:

Variables declared Globally (outside any function) have Global Scope. Global variables can be accessed from anywhere in a JavaScript program.

var a=10

{

  var x = 2;

}

// x can be used here

**Block Scope**:

Variables declared within a block cannot be accessed from outside of the block.

{

  let x = 2;

}

// x can NOT be used here

**Local Scope**:

Variables declared inside a function have local scope. They can only be accessed within that function. They cannot be accessed from the outside.

function myFunction() {

  var a = "skills";

  // code here CAN use a

}

// code here can NOT use a

**VAR LET CONST**

variable declared with var have global scope can be redeclared and reassigned:-

1) Variables declared with var inside a { } block can be accessed from outside the block because it doesn’t have a block scope it has global scope

{

  var x = 2;

}

// x CAN be used here

//because it has a global scope

2) variable declared with var can be redeclared

var x = "front end";

var x = 0;

//Variables defined with var can be redeclared.

3) variable declared with var can be re-assigned

var x;

x = 0;

x=10;

variable declared with let have block scope cannot be redeclared and can be reassigned:-

1) Variables declared inside a { } block cannot be accessed from outside the block:

{

  let x = 2;

}

// x can NOT be used here

2) Variables defined with let can not be redeclared.

let x = "John Doe";

let x = 0;

//output – identifier has already been declared

3) Variable defined with let can be reassigned.

let x;

x = 0;

x=20;

//output- 20

variable declared with const have block scope cannot be redeclared and cannot be reassigned:-

1) Variables declared inside a { } block cannot be accessed from outside the block:

{

 const x = 2;

}

// x can NOT be used here

2) Variables defined with const can not be redeclared.

Const x = "coding”;

Const x = 0;

output – identifier has already been declared

3) Variable defined with const cannot be reassigned.

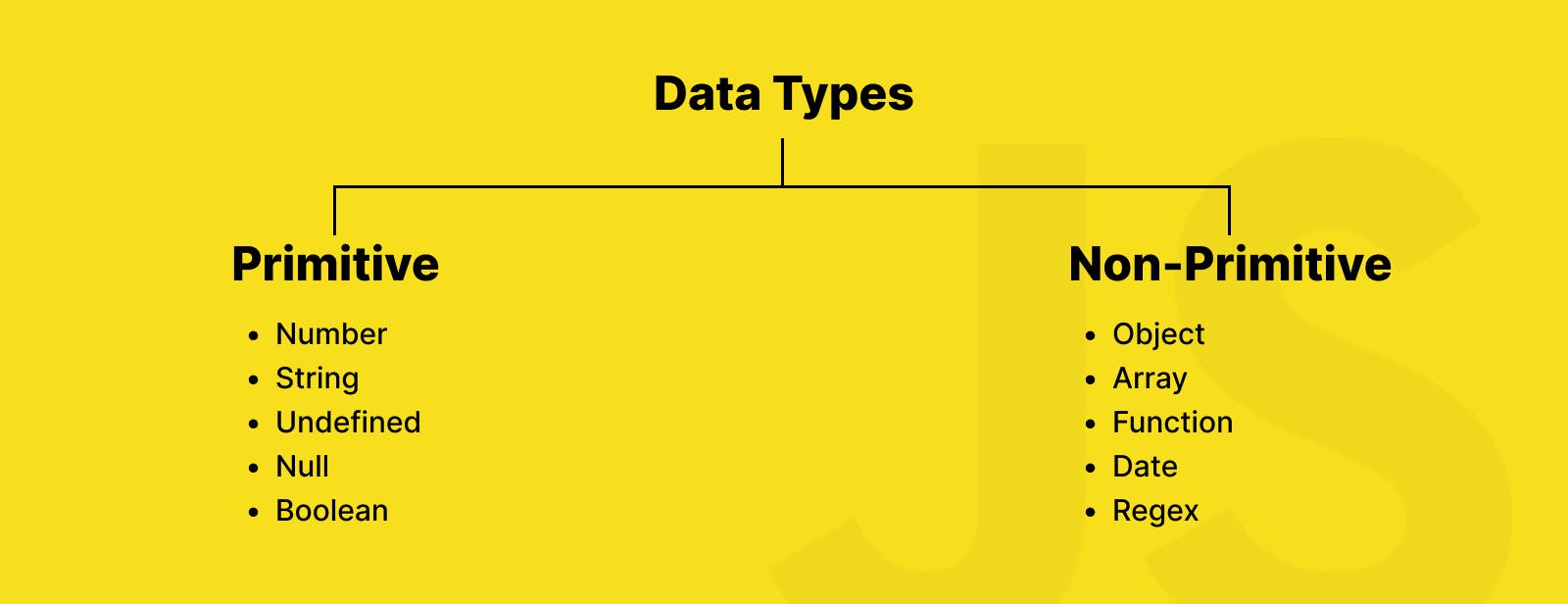
Const x;

x = 0;

x=20;

//output-missing initializer in const declaration

**Data types**



1. Primitive Data Types:primitive data types are the fundamental building blocks used to represent single values. Primitive data types are directly stored in stack memory and are immutable, meaning their values cannot be changed after they are created.
   * **Number**: Represents numeric values, including integers and floating-point numbers.

        let integer = 42;          // Integer

        let float = 3.14;          // Floating-point number

        let negative = -7;         // Negative number

        let infinity = Infinity;   // Infinity

        let notANumber = NaN;      // NaN (result of an invalid number operation)

* + **String**: Represents textual data, enclosed in single or double quotes.

        let singleQuoteString = 'Hello, world!';

        let doubleQuoteString = "Hello, world!";

        let templateLiteral = `Hello, ${name}!`; // Template literals allow embedded expressions

* + **Boolean**: Represents a logical value, **true** or **false**.

        let isJavaScriptFun = true;

        let isTired = false;

* + **Undefined**: Represents a variable that has been declared but has not been assigned a value.

    let notAssigned; // `notAssigned` is undefined

        console.log(notAssigned); // undefined

* + **Null**: Represents an intentional absence of any object value.

        let emptyValue = null;

1. Composite Data Types or Non primitive:Non-primitive data types, also known as reference types, are more complex data structures that can hold multiple values and have methods and properties.non-primitive data types are mutable, meaning their values can be changed after they are created, and they are stored and accessed by reference rather than by value.
   * **Array**: Represents an ordered collection of elements, accessed by index, starting from zero.

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

        let mixedArray = [1, 'hello', true, { key: 'value' }];

* + **Object**: Represents a collection of key-value pairs, where keys are strings and values can be any data type, including other objects.

         let person = {

        name: 'John',

        age: 30,

        isEmployed: true

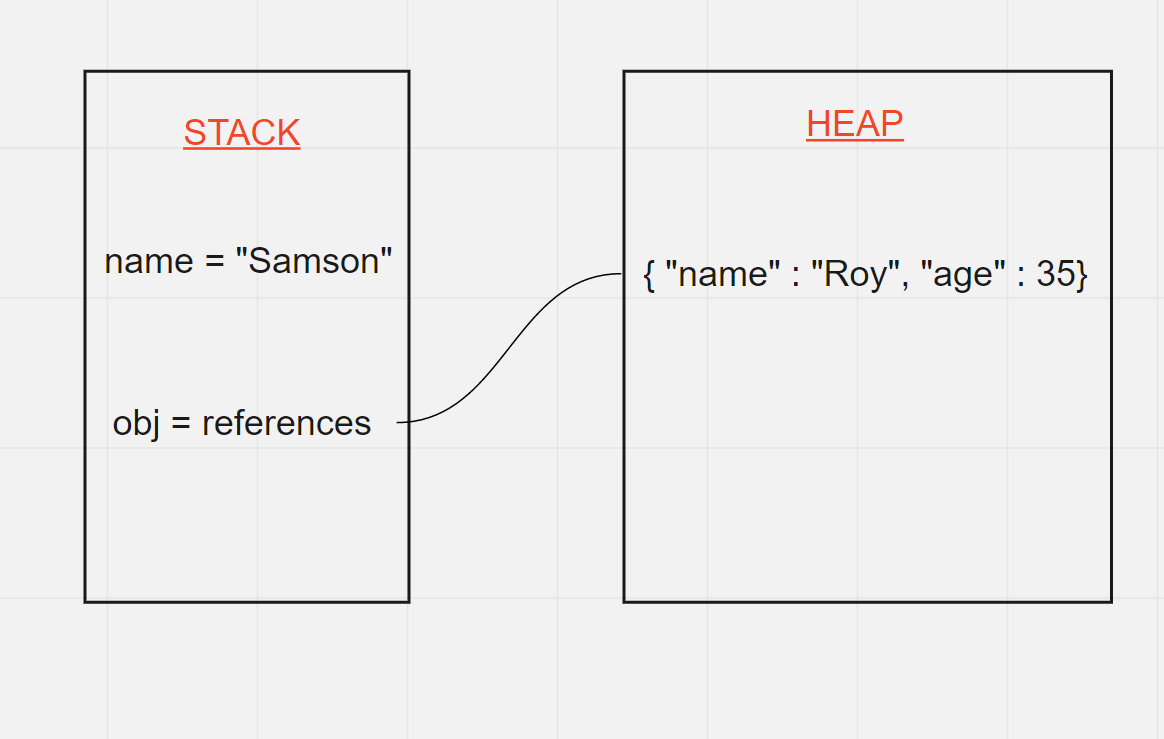
        };

* + **Function**: Represents a reusable block of code that can be executed by invoking it.

    let sayHello = function(name) {

        return `Hello, ${name}!`;

    };



**Null & Undefined**

1. **null**:
   * Think of **null** as an intentional empty value. It's used when you want to say, "This variable intentionally has no value."
   * For example, if you have a variable **person** but don't yet know the person's details, you might set **person** to **null**.
2. **undefined**:
   * **undefined** means a variable has been declared but hasn't been assigned a value yet.
   * It's the default value of variables that haven't been initialized.
   * For example, if you declare a variable **x** but don't assign any value to it, **x** is **undefined** by default.

Bigint

Integers are safely handled within a specific range due to the use of **double-precision floating-point format** (64-bit) based on the IEEE 754 standard. Here's how it works:

**Safe Integer Limit in JavaScript:**

* The safe range for integers is between:
  + **- (2^53 - 1)** (or **-9,007,199,254,740,991**)
  + **2^53 - 1** (or **9,007,199,254,740,991**)

These limits are defined by two constants in JavaScript:

* Number.MAX\_SAFE\_INTEGER (which is **9,007,199,254,740,991**)
* Number.MIN\_SAFE\_INTEGER (which is **-9,007,199,254,740,991**)

Numbers beyond this range may lose precision when stored or manipulated, causing rounding errors.

# var number = 9007199254740991; // Last safe number

# console.log(number); // 9007199254740991 (Correct)

# var numberTooBig = 9007199254740992; // First unsafe number

# console.log(numberTooBig); // 9007199254740992 (Okay, looks fine)

# var anotherBigNumber = 9007199254740993; // Uh oh!

# console.log(anotherBigNumber); // 9007199254740992 (Wrong, it got rounded)

To handle numbers beyond the safe integer limit without losing precision, JavaScript provides the BigInt type:

# var bigNumber = BigInt("888888888888888888888");

# console.log(bigNumber); // 888888888888888888888n

Symbol

**Symbol** data type is a unique and immutable (unchangeable) value, introduced in ES6 (ECMAScript 2015). Unlike other data types like strings, numbers, or objects, a Symbol is primarily used to create unique identifiers. This makes it particularly useful for avoiding naming conflicts in your code, especially when dealing with properties in objects

const uniqueId = Symbol('id');

let user = {

name: "Alice",

age: 25,

[uniqueId]: 12345 // Using the symbol as a key

};

console.log(user); // { name: 'Alice', age: 25, [Symbol(id)]: 12345 }

console.log(user[uniqueId]); // 12345

**Operators:**

Operators are symbols used to perform operations on variables and values

**1 ) Arithmetic Operators**: Used to perform arithmetic operations

      let x = 10;

      let y = 5;

      let addition = x + y; // Addition

      let subtraction = x - y; // Subtraction

      let multiplication = x \* y; // Multiplication

      let division = x / y; // Division

let expo=x\*\*y //Exponential

      let modulus = x % y; // Modulus (remainder)

      let increment = x++; // Increment

      let decrement = y--; // Decrement

**2 ) Assignment Operators**: Used to assign values to variables.

      let x = 10;

      x += 5; // Equivalent to x = x + 5

      x -= 5; // Equivalent to x = x - 5

      x \*= 5; // Equivalent to x = x \* 5

      x /= 5; // Equivalent to x = x / 5

**3 ) Comparison Operators**: Used to compare values. They return a boolean value - true or false.

      let a = 10;

      let b = 5;

      console.log(a > b); // Greater than

      console.log(a < b); // Less than

      console.log(a >= b); // Greater than or equal to

      console.log(a <= b); // Less than or equal to

      console.log(a === b); // Equal to (strict equality)

      console.log(a !== b); // Not equal to (strict inequality)

**4 ) Logical Operators**: Used to combine or manipulate boolean values.

      let p = true;

      let q = false;

      console.log(p && q); // AND

      console.log(p || q); // OR

      console.log(!p); // NOT

**//logical and**

## **console.log(true && true); // true**

## **console.log(true && false); // false**

## **console.log(false && true); // false**

## **console.log(false && false); // false**

**//logical or**

## **console.log(true || true); // true**

## **console.log(true || false); // true**

## **console.log(false || true); // true**

## **console.log(false || false); // false**

**5 ) Ternary Operator (Conditional Operator):** Used to assign a value to a variable based on a condition.

## **condition ? expressionIfTrue : expressionIfFalse;**

      let age = 20;

      let status = age >= 18 ? "Adult" : "Minor";

      console.log(status); // Output: 'Adult'

When nesting ternary operators, you place one ternary operation inside another. This is useful when you have multiple conditions to evaluate in a compact way. Here's an example of how a nested ternary operator works:

# **let result = condition1 ? value1 : condition2 ? value2 : value3;**

**Breakdown of Syntax:**

1. **First condition** (condition1):
   * If true, it will return value1.
   * If false, it will check the next condition (condition2).
2. **Second condition** (condition2):
   * If true, it will return value2.
   * If false, it will return value3.

**6 ) Nullish Coalescing Operator (??)**

It is a logical operator that returns its right-hand operand when its left-hand operand is **null** or **undefined**, and otherwise returns its left-hand operand. It's useful for providing default values in expressions without overriding valid falsy values like 0, NaN, or ''.

Syntax

let result = expression1 ?? expression2;

      //case - 1

      let name1 = null;

      let defaultName1 = name1 ?? "yes it is null or undefined";

      console.log(defaultName1); // Output: 'yes it is null or undefined'

      //case - 2

      let name2 = undefined;

      let defaultName2 = name2 ?? "yes it is null or undefined";

      console.log(defaultName2); // Output: 'yes it is null or undefined'

      //case - 3

      let name3 = "hello world";

      let defaultName3 = name3 ?? "yes it is null or undefined";

      console.log(defaultName3); // Output: hello world

Practical Applications

Handling Optional Function Parameters:

      function hello(a){

        var b=a ?? "Dear";

        console.log("hello " + b);

      }

      hello();// we are not any value but still we are getting value because of Nullish Coalescing Operator

      hello("teja")

**7 ) Optional Chaining Operator (?.)**

It is a powerful tool that allows for safe navigation through nested object properties, functions, and arrays. It prevents runtime errors that occur when accessing properties of null or undefined objects, returning undefined instead of throwing an error.

**Syntax**

let result = object?.property;

let result = object?.[property];

let result = object?.method?.();

      // wihout using optional chaining operator

      var obj = {

        name: "tej",

        state: {

          name: "ap",

        },

      };

      console.log(obj.obj.state); //it throws an error

      // with using optional chaining operator

      var obj = {

        name: "tej",

        state: {

          name: "ap",

        },

      };

  console.log(obj.obj?.state); //undefined instead of error

**8 ) Bitwise operator**

**What are bitwise operators?**

Bitwise operators are used to perform operations at the **bit** level, where a **bit** is the smallest unit of data in a computer, represented as either a **0** or a **1**. Every number in a computer is stored in binary (which is a series of 0s and 1s).

**AND (&)**

Compares two numbers **bit by bit**. If both bits are 1, the result is 1. Otherwise, it’s 0.

Think of it like a "both must be true" rule.

101 (binary for 5) & 011 (binary for 3) --------

001 (binary for 1)// Result: 1

* Sets each bit to 1 if both corresponding bits are 1.

**let a = 5; // 0101**

**let b = 3; // 0011**

**let result = a & b; // 0001 (1 in decimal)**

**OR (|)**

* Compares two numbers **bit by bit**. If **either** bit is 1, the result is 1. If both are 0, the result is 0.
* Think of it like "if either is true."

101 (binary for 5) | 011 (binary for 3) --------

111 (binary for 7)// Result: 7

* Sets each bit to 1 if at least one of the corresponding bits is 1.

**let a = 5; // 0101**

**let b = 3; // 0011**

**let result = a | b; // 0111 (7 in decimal)**

**Bitwise XOR (^)**

* Compares two numbers **bit by bit**. If the bits are different, the result is 1. If they are the same, the result is 0.
* Think of it like "one or the other, but not both."

101 (binary for 5) ^ 011 (binary for 3) --------

110 (binary for 6)// Result: 6

* Sets each bit to 1 if only one of the corresponding bits is 1.

**let a = 5; // 0101**

**let b = 3; // 0011**

**let result = a ^ b; // 0110 (6 in decimal)**

**9** **) String operators**

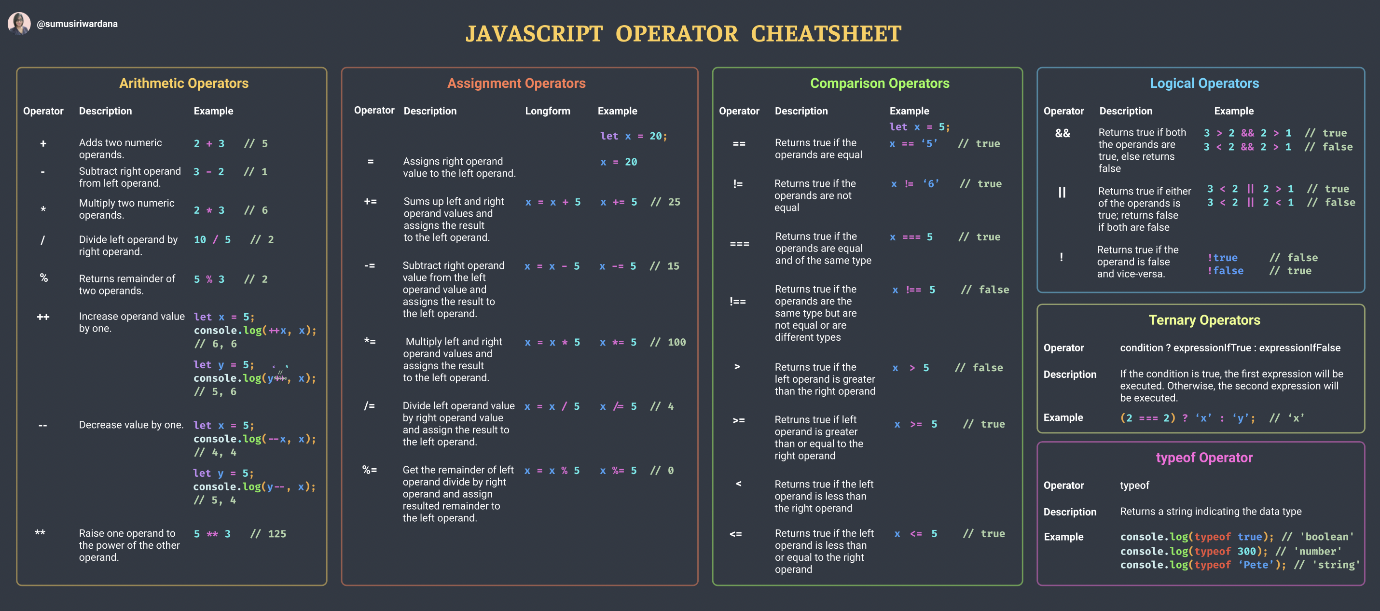
It is used to perform operations on string values. The primary string operator is the concatenation operator (+), but other operations involving strings include template literals, comparison operators, and methods available on string objects.

**Concatenation Operator (+)**

The concatenation operator combines two or more strings into a single string.

let greeting = "Hello, " + "world!";

console.log(greeting); // Outputs: "Hello, world!"



**Type Coercion**

Type coercion refers to the automatic or implicit conversion of values from one data type to another. This process happens in the background during operations involving values of different types.

**Types of Type Coercion**

**Implicit Coercion**: This occurs automatically when JavaScript encounters an operation involving different data types.

**Explicit Coercion**: This is when you manually convert a value from one type to another using functions or methods.

**Explicit Type Conversion**

JavaScript type conversion, allowing you to convert values from one data type to another.

1. **String()**: Converts a value to a string.

let num = 123;

let str = String(num);

console.log(str); // Output: "123"

1. **Number()**: Converts a value to a number.

let str = "123";

let num = Number(str);

console.log(num); // Output: 123

3. **Boolean()**: Converts a value to a boolean.

let num = 0;

let bool = Boolean(num);

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

**In JavaScript, values are categorized as either "truthy" or "falsy"**

**Falsy Values:**

1. **false**: The boolean value false itself.
2. **0**: The number zero.
3. **""**: Empty string.
4. **null**: The absence of any value.
5. **undefined**: A variable that has not been assigned a value or a property that does not exist.
6. **NaN**: Not-a-Number.

**Truthy Values:**

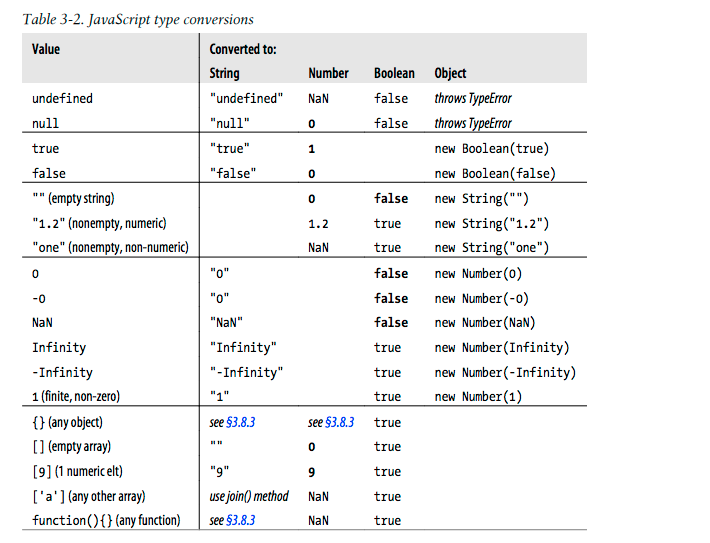
1. **true**: The boolean value true itself.
2. **Non-zero numbers**: Any number other than 0 (including negative numbers and decimals).
3. **Non-empty strings**: Any string with at least one character.
4. **Non-empty arrays**: Arrays with at least one element.
5. **Objects**: Any object (including functions and arrays) is truthy, even if it's empty.
6. **Functions**: Any function is truthy, even if it doesn't return anything.

      //program to find falsy and truthy values

      var a = [];

      var b = a ? true : false;

      console.log(b);



**Conditional statements**

Conditional statements allows you to execute different blocks of code based on specified conditions.

**1. if Statement:**

The **if** statement executes a block of code if a specified condition is true.

Syntax:

## **if (condition) {**

## **// Code to execute if condition is true**

## **}**

      let x = 10;

      if (x > 0) {

        console.log("x is positive");

      }

**2. if...else Statement:**

The **if...else** statement executes one block of code if a specified condition is true and another block if the condition is false.

## **if (condition) {**

## **// Code to execute if condition is true**

## **} else {**

## **// Code to execute if condition is false**

## **}**

      let x = -5;

      if (x > 0) {

        console.log("x is positive");

      } else {

        console.log("x is non-positive");

      }

**3)if...else if...else Statement:**

The **if...else if...else** statement allows you to specify multiple conditions and execute different code blocks based on the outcome of those conditions.

## **if (condition1) {**

## **// Code to execute if condition1 is true**

## **} else if (condition2) {**

## **// Code to execute if condition2 is true**

## **} else {**

## **// Code to execute if none of the conditions are true**

## **}**

      let x = -5;

      if (x > 0) {

        console.log("x is positive");

      } else if (x < 0) {

        console.log("x is negative");

      } else {

        console.log("x is zero");

      }

**Switch statements**

A switch statement in JavaScript is a control flow statement that allows you to execute a block of code among many options based on the value of an expression.

# **switch (expression) {**

# **case value1:**

# **// Code to run if expression === value1**

# **break;**

# **case value2:**

# **// Code to run if expression === value2**

# **break;**

# **// More cases...**

# **default:**

# **// Code to run if no case matches**

# **}**

**Key Points**

1. **Expression Evaluation**: The **expression** inside the switch statement is evaluated once.
2. **Case Matching**: The result of the expression is compared with the values specified in each **case** clause using strict equality (**===**).
3. **Code Execution**: If a match is found, the code block associated with that **case** is executed.
4. **Break Statement**: The **break** statement is used to terminate the switch statement. If omitted, execution will continue to the next **case** clause (fall-through behavior).
5. **Default Case**: The **default** clause is optional and executes if no matching **case** is found. It acts like the **else** in an if-else structure.

switch (grade) {

  case 'A': console.log('Excellent');

    break;

  case 'B':

  case 'C':console.log('Well done');

    break;

  case 'D':console.log('You passed');

break;

  case 'F':console.log('Better try again');

    break;

  default:console.log('Invalid grade');

}

**Loops**

Loops in JavaScript are control structures that allow you to repeat a block of code multiple times. They are essential for performing repetitive tasks, such as iterating over arrays, processing data, or implementing certain algorithms.

* **for loop**: A **for** loop is used to execute a block of code a number of times. It consists of three optional expressions enclosed in parentheses, separated by semicolons:

for (initialization; condition;updation) {

// code to be executed

}

* **Initialization**: Executes once at the beginning of the loop.
* **Condition**: Evaluated for each iteration. If true, the loop continues; if false, the loop terminates.
* **Updation**: Executed after each iteration. Typically used to update the loop counter.

      // Print numbers from 1 to 5

      for (let i = 1; i <= 5; i++) {

        console.log(i);

      }

* **while loop**: A **while** loop repeats a block of code while a specified condition is true. It has the following syntax:

while (condition) { // code to be executed }

* The condition is evaluated before each iteration. If it returns true, the loop continues; otherwise, it stops.

      // Print numbers from 1 to 5 using a while loop

      let i = 1;

      while (i <= 5) {

        console.log(i);

        i++;

      }

* **do...while loop**: Similar to the **while** loop, but it always executes its block of code at least once, even if the condition evaluates to false. It has the following syntax:

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

* The block of code is executed first, then the condition is evaluated. If true, the loop continues; if false, it stops.

      // Print numbers from 1 to 5 using a do...while loop

      let j = 1;

      do {

        console.log(j);

        j++;

      } while (j <= 5);

**What are Nested Loops?**

1.Nested loops are loops within loops. They are useful for iterating over multi-dimensional arrays or performing complex tasks that require multiple levels of iteration.

**2. Basic Structure**

The basic structure of a nested loop in JavaScript is as follows:

**JavaScript**

for (let i = 0; i < outerLimit; i++) {

for (let j = 0; j < innerLimit; j++) {

// Code to execute

}

}

**3. Multiplication Table**

Using nested loops to create a multiplication table:

for (let i = 1; i <= 10; i++) {

let row = "";

for (let j = 1; j <= 10; j++) {

row += (i \* j) + "\t";

}

console.log(row);

}

* **for...in loop**: Used to iterate over the properties of an object,array, string. It iterates over enumerable properties of an object, in an arbitrary order.

Syntax:

      for (ref in strname){

        console.log(ref);//indexes

      }

for (variable in object) { // code to be executed }

      // Iterate over the properties of an object

      const person = {

        name: "John",

        age: 30,

        gender: "male",

      };

      for (let prop in person) {

        console.log(prop + ": " + person[prop]);

      }

1) **Iterates over Properties**:

* The for...in loop iterates over all enumerable properties of an object.

2) **Order Not Guaranteed**:

* The order of iteration is not guaranteed. It's generally the order in which properties were defined, but this can vary.

3) **Use with Objects**:

* Typically used for objects, not arrays, because it iterates over property names (keys) rather than values.
* **for...of loop**: Introduced in ES6, it iterates over iterable objects such as arrays, strings, maps, sets, etc.

Syntax:

      for (ref of strname){

        console.log(ref);//values

      }

for (variable of iterable) { // code to be executed }

* It provides a more concise syntax compared to the traditional **for** loop for iterating over arrays and other iterable objects.

      // Iterate over elements of an array

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

      for (let num of numbers) {

        console.log(num);

      }

1) **Iterates over Values**:

* The for...of loop iterates over the values of an iterable object.
* This loop does not work with objects unless they implement the iterable protocol.

2) **Use with Arrays and Other Iterables**:

* Commonly used with arrays, strings, maps, sets, and other iterable objects.
* **Nested loops**: You can nest loops inside one another to perform more complex iterations. For example, you can use a **for** loop inside another **for** loop to iterate over a two-dimensional array.

**How continue and break behaves in loops**

continue and break are used to control the flow of loops, such as for, while, and do...while loops. They are not used directly within conditional statements like if, else, or switch. However, they can be used inside loops that contain conditional statements to influence the loop's behavior based on certain conditions.

**continue Statement**

The continue statement is used to skip the current iteration of a loop and move on to the next iteration. When the continue statement is encountered, the loop's current iteration is terminated, and control is passed to the next iteration of the loop.

      for (let i = 0; i < 10; i++) {

        if (i % 2 === 0) {

          continue; // Skip even numbers

        }

        console.log(i); // This will only log odd numbers

      }

In this example, the continue statement skips the even numbers, so the console.log(i) statement only logs the odd numbers from 1 to 9.

**break Statement**

The break statement is used to terminate the entire loop immediately. When the break statement is encountered, the loop is exited, and control is passed to the statement following the loop.

      for (let i = 0; i < 10; i++) {

        if (i === 5) {

          break; // Exit the loop when i is 5

        }

        console.log(i); // This will log numbers 0 to 4

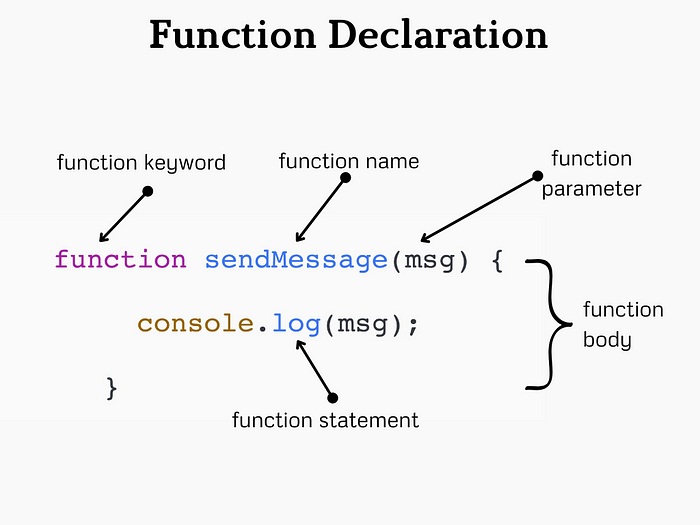
      }

In this example, the break statement causes the loop to terminate when i is equal to 5, so console.log(i) only logs the numbers from 0 to 4.

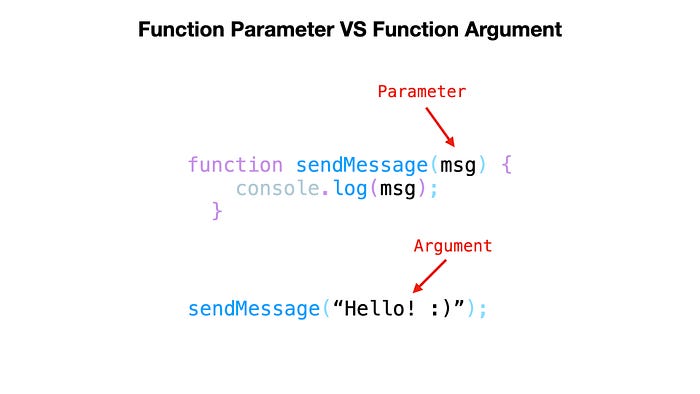
**Function:-**

**A function is a reusable block of code that is used to perform a specific task when something invokes it.**

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().The code to be executed, by the function, is placed inside curly brackets: {}



Function parameters are listed inside the parentheses () in the function definition. Function arguments are the values received by the function when it is invoked. Inside the function, the arguments (the parameters) behave as local variables.



Named functions can be hoisted

Ex 1:-

        function myfun(a){

            return console.log(a);

            alert("it will not execute because it was in void")

        }

        myfun(20);

above program shows the output of 20 and alert will not be showed because it was returned after the statement of return means it was in voids

Points needs to be noted in functions

1) function name stores function definition

2) log off the function calling stores return value

3) statement after return will not execute because It was in void

4) Named function can only hoisted

5) function definition act as value because in js functions are first class functions

**Anonymous function:-**

Anonymous function is a function that is defined without a name

*var* anonfun=function(){

            return "this is anonymous function";

        }

        console.log(anonfun());

above program prints “this is anonymous function” in the output.it is similar to the named function but the difference is hoisting is not applied,it is also called as function expression.

**Arrow function:-**

Arrow function is a concise way of writing function in shorter way.

        var arrowfun=()=>{

            return "this is arrow function";

        }

        console.log(arrowfun());

above program prints “this is arrow function” in the output. It is a also have same functionality , it is also not hoisted.

Function with default parameters

        function hello(a="this is a function ",b="with default parameters"){

            return a+b;

        }

        console.log(hello());

above program shows the output “this is a function with default parameters” .Above program takes default parameters to print the output because we haven’t gave the parameters while calling the function.

**Immediately Invoked Function Expression**

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

Following shows the syntax

(function(){

*//code goes here*

})()

        (function(){

            console.log("self invoking function invoked by itself") ;

        })();

**Callback Function**

Callback function is a function definition passed into a another function as an argument which is then invoked inside the outer function to complete some kind of task.

        function hello1(){

            return "hello1 function is triggered by main function";

        }

        function hello2(j){

            var a=j();

         return a;

        }

        console.log(hello2(hello1));

in following program hello1 is passed as an argument to a hello2 function .hello2 starts execution line by line. hello2 stores the hello1 function in j . In the next statement variable invokes the function stores the ouput in it and displays the output in a console “hello1 function is triggered by main function”.

**Global execution context:**

Global Execution Context is the first context that gets created when the JavaScript engine starts executing code.

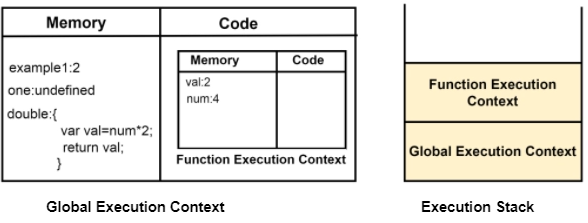
**Key Components**

**1. Memory Allocation (Creation Phase)**:

* During this phase, the engine sets up the memory for variables and functions.
* **Variables** declared with **var** are hoisted and initialized with **undefined**.
* **Function declarations** are hoisted and their definitions are stored in memory.
* Variables declared with **let** and **const** are also hoisted but are not initialized. They remain in a temporal dead zone until they are assigned a value

**2. Code Execution (Execution Phase)**:

* + The JavaScript engine executes the code line by line.
  + Variables declared with **var** are assigned their values.
  + Functions are available to be called.
  + Variables declared with **let** and **const** are assigned values when their declaration is encountered in the code.



**How it works on the functions**

When a function is invoked, a new **Execution Context(function execution context)** is created specifically for that function. This context is separate from the Global Execution Context but follows similar principles.

* Each function invocation creates a new execution context.
* Variables declared inside a function are local to that function and are not accessible outside it.
* The scope chain allows inner functions to access variables from their parent functions and the global context.

**Closures**

A closure is a function that has access to its own scope, the scope of the outer function, and the global scope. This means a closure can remember and access variables from its outer function even after that function has finished executing.

When an inner function have access to the variable which is declared in outer function even after the execution

      function ahello() {

        var a = "variable inside a outer function";

        function ahi() {

          var b = "varaible inside inner function";

          console.log(a);

console.log(b);

        }

        ahi();

      }

      ahello();

**Scope Chaining in JavaScript**

**Scope** in JavaScript refers to the context in which variables, functions, and objects are accessible. JavaScript has three types of scope:

1. **Global Scope:** Variables declared outside of any function or block are in the global scope. They are accessible from anywhere in the code.
2. **Local Scope:** Variables declared within a function or block are in the local scope. They are only accessible within that function or block.
3. **Block Scope:** Variables declared inside a block can’t able to access outside of the function

**Scope Chain**:

* When a variable is accessed, JavaScript looks for it in the current scope.
* If the variable is not found, it looks in the outer scope.
* This process continues until it reaches the global scope.
* If the variable is not found in any scope, it results in a ReferenceError

  var globalVar = "I am global";

function outerFunction() {

    var outerVar = "I am outer";

    function innerFunction() {

        var innerVar = "I am inner";

        console.log(innerVar);     // Output: I am inner

        console.log(outerVar);     // Output: I am outer

        console.log(globalVar);    // Output: I am global

    }

    innerFunction();

    console.log(outerVar);         // Output: I am outer

    console.log(globalVar);        // Output: I am global

    // console.log(innerVar);      // Error: innerVar is not defined

}

outerFunction();

console.log(globalVar);            // Output: I am global

// console.log(outerVar);          // Error: outerVar is not defined

// console.log(innerVar);          // Error: innerVar is not defined

**Lexical Scoping:**

* JavaScript uses lexical scoping, meaning that the scope of a variable is determined by its position in the source code.
* Inner functions have access to variables declared in their outer functions (but not vice versa).

**Strings**

\*collection of words or characters enclosed in a single or double quotes

\*Strings are immutable can’t be change the values directly we can get the values of string but we cant change.

     //iterate the string using for loop

      var a ="hello";

      for(i=0;i<5;i++){

        console.log(a[i]);

      }

**String.length**

It is a method used to find the length of a string

The **.length** property returns the number of characters in the string, including spaces, punctuation marks, and special characters.

let str = "Hello, world!";

console.log(str.length); // This will output 13

**charAt()**

The charAt() method is used to return the character at a specified index (position) within a string.

let str = "Hello";

console.log(str.charAt(0)); // Output: "H"

console.log(str.charAt(1)); // Output: "e"

console.log(str.charAt(4)); // Output: "o"

**at()**

The at() method allows you to directly access a character at a specific position within a string, similar to charAt() but also takes negative values.

let str = "Hello";

console.log(str.at(0)); // Output: "H"

console.log(str.at(1)); // Output: "e"

console.log(str.at(-1)); // Output: "o"

**charCodeAt()**

the charCodeAt() method returns the Unicode value (integer between 0 and 65535) of the character at a specified index in a string.

let str = "Hello";

console.log(str.charCodeAt(0)); // Output: 72

console.log(str.charCodeAt(1)); // Output: 101

console.log(str.charCodeAt(4)); // Output: 111

**slice(start,end)**

the slice() method is used to extract a section of a string and return it as a new string. It doesn't modify the original string. This method takes two parameters: the start index and the end index (optional).

let str = "Hello, world!";

console.log(str.slice(0, 5)); // Output: "Hello"

console.log(str.slice(7)); // Output: "world!"

console.log(str.slice(-6)); // Output: "world!"

console.log(str.slice(7, -1)); // Output: "world"

console.log(str.slice(0)); // Output: "Hello, world!"

console.log(str.slice(-1)); // Output: "!"

**substring(start, end)**

The **substring()** method is used to extract a portion of a string and return it as a new string. It is similar to the **slice()** method, but there are differences in how negative indices are handled because negative values are considered as **zero**

let str = "Hello, world!";

console.log(str.substring(0, 5)); // Output: "Hello"

console.log(str.substring(7)); // Output: "world!"

console.log(str.substring(7, 12)); // Output: "world"

console.log(str.substring(-6)); // Output: "Hello, world!"

console.log(str.substring(7, -1)); // Output: "Hello, world"

**substr()**

In JavaScript, the substr() method is used to extract a portion of a string, starting from a specified index and extending for a specified length of characters. This method is different from substring() in that the second parameter specifies the length of the extracted substring rather than the end index.

let str = "Hello, world!";

console.log(str.substr(0, 5)); // Output: "Hello"

console.log(str.substr(7)); // Output: "world!"

console.log(str.substr(7, 5)); // Output: "world"

console.log(str.substr(-6)); // Output: "world!"

console.log(str.substr(7, -1)); // Output: ""

**toUpperCase()**

The toUpperCase() method is used to convert all characters in a string to uppercase letters.

let str = "Hello, world!";

let a= str.toUpperCase();

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

**toLowerCase()**

The toLowerCase() method is used to convert all characters in a string to lowercase letters.

let str = "Hello, WORLD!";

let a = str.toLowerCase();

console.log(a); // Output: "hello, world!"

**concat()**

The concat() method is used to concatenate two or more strings.

let str1 = "Hello";

let str2 = "world";

let str3 = "!";

let result = str1.concat(", ", str2, str3);

console.log(result); // Output: "Hello, world!"

**trim()**

The trim() method is used to remove whitespace from both ends of a string.

let str = "   Hello, world!   ";

let trimmedStr = str.trim();

console.log(trimmedStr); // Output: "Hello, world!"

**repeat()**

The repeat() method is used to construct and return a new string by concatenating the string on which it is called a certain number of times.

let str = "Hello";

let repeatedStr = str.repeat(3);

console.log(repeatedStr); // Output: "HelloHelloHello"

**Split()**

the split() method is used to split a string into an array.

let str = "Hello, world!";

let parts = str.split(", ");

console.log(parts); // Output: ["Hello", "world!"]

let characters = str.split("");

console.log(characters); // Output: ["H", "e", "l", "l", "o", ",", " ", "w", "o", "r", "l", "d", "!"]

**replace()**

replace() method used to replace the current occurences of substring within a string with another string

syntax: **string.replace(searchValue, replaceValue)**

    let originalString = "Hello, world!,world";

    let newString = originalString.replace("world", "universe");

    console.log(newString); // Output: Hello, universe!,world

Above program replaces only the first match to replace all matches use a regular expression with /**g** flagset

let newString = originalString.replace(**/world/g**, "red fox");

    let originalString = "Hello, world!,world";

    let newString = originalString.replace(/world/g, "universe");

    console.log(newString); // Output: Hello, universe!,universe

**Replace** method is case sensitive writing World will not consider

To replace case insensitive, use a **regular expression** with an /i

    let originalString = "Hello, world!, World";

    let newString = originalString.replace(/world/ig, "universe");

    console.log(newString); // Output: Hello, universe!,universe

**replaceAll()**

it is a method to replace a substring with another sting but it doesn’t compatible in all browsers

**string search methods**

**indexOf() and lastIndexOf**

    let str= "Hellow, world!";

    let newString = str.indexOf("w")//output 5 because it checks from the starting

    let newString1= str.lastIndexOf("w")//output 8 because it checks from the ending

Both indexOf(), and lastIndexOf() return -1 if the text is not found

    let newString1= str.lastIndexOf("w",5)//output 8 because it checks from the ending

if the second parameter is 5, the search starts at position 5, and searches

**search()**

the search() method is used to search for a specified substring within a string. It returns the index of the first occurrence of the specified substring, or -1 if the substring is not found. It can take regular expressions also

let str = "Hello, world!";

let index = str.search("world");

console.log(index); // Output: 7

two methods, indexOf() and search(), are not **equal because**

search() method cannot take a second start position argument.

**match()**  
The match() method in JavaScript is used to search a string for a specified pattern (regular expression), and returns an array containing the matches, or null if no matches are found.

It can take regular expression and it can print the values in an array

    let text = "The rain in SPAIN stays mainly in the plain";

    text.match(/ain/gi);    //4 [ain,AIN,ain,ain]

**includes()**

the includes() method returns true if a string contains a specified value.

let text = "Hello world, welcome to the universe.";

text.includes("world");//true

**Template literals**

Template literals allow you to embed expressions and variables directly within the string using **${}**. This makes **string interpolation** more intuitive and readable.

    let a = 5;

    let b = 10;

    let result = `The sum of ${a} and ${b} is ${a + b}.`;

    // result is "The sum of 5 and 10 is 15."

**Arrays**

Array is a special type of object used to store multiple values in a single variable. Arrays allow you to group data and perform various operations on them, such as adding, removing, or modifying elements.

**Features of array**

1. Order collection
2. Homogenous or heterogenous-(mixed datatypes)
3. Muttable
4. Dynamic size
5. Multi dimensional array

**Creating Arrays:**

You can create an array using square brackets **[]** and separating the elements with commas.

let arr = [1, 2, “html”, “Css”];

**Accessing Elements:**

You can access elements of an array using square brackets and the index of the element. Remember that array indices start at 0.

console.log(arr[0]); // Output: 1

console.log(arr[3]); // Output: css

**Modifying Elements:**

You can modify elements in an array by assigning a new value to a specific index.

arr[1] = 'js';

console.log(arr); // Output: [1, ‘js’, ‘html’,’css’]

**Array Methods:**

JavaScript provides many built-in methods to work with arrays, such as **push**, **pop**, **shift**, **unshift**, **slice**, **splice**, **concat**, **indexOf**, **includes**, and many more.

**Array Length:**

**1. Array length:**

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

let arr = [1, 2, “html”, “Css”];

console.log(arr.length); // Output: 4

**2. Array at():**

   - The at() method of Array instances takes an integer value and returns the item at that index, allowing for positive and negative integers. Negative integers count back from the last item in the array.

**3. concat()** method is used to merge two or more arrays. It does not modify the existing arrays but instead returns a new array containing the elements of the original arrays concatenated together.

let array1 = [1, 2, 3];

let array2 = ['a', 'b', 'c'];

let newArray = array1.concat(array2);

console.log(newArray); // Output: [1, 2, 3, 'a', 'b', 'c']

**4. Array splice():**

**splice()** method in JavaScript is used to change the contents of an array by removing or replacing existing elements and/or adding new elements in place. It modifies the original array and returns an array containing the removed elements.

array.splice(**startIndex, deleteCount, addeditem1, addeditem2**);

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

numbers.splice(2, 2, 'a', 'b'); // Replaces 2 elements starting from index 2 with 'a' and 'b'

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

**5. Array slice():**

**slice()** method in JavaScript is used to extract a section of an array and returns a new array containing the extracted elements. It does not modify the original array; instead, it returns a shallow copy of a portion of the array.

array.slice**(startIndex, endIndex);**

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

let slicedArray = numbers.slice(1, 4); // Extracts elements from index 1 to index 3 (exclusive)

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

If startIndex is negative:

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

let slicedArray = numbers.slice(-3); // Extracts the last 3 elements of the array

console.log(slicedArray); // Output: [3, 4, 5]

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

let slicedArray = numbers.slice(1, -1); // Extracts elements from index 1 to second last element

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

**6. Array pop():**

   - `pop()` method removes the last element from an array and returns that element.

**7. Array push():**

   - `push()` method adds one or more elements to the end of an array and returns the new length of the array.

**8. Array shift():**

   - `shift()` method removes the first element from an array and returns that element.

**9. Array unshift():**

   - `unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array.

**10 Array sort and reverse method**

numbers.sort ();

numbers.reverse ();

11**. Array toString():**

**toString()** method is used to convert an array to a string. This method converts each element of the array to a string and then concatenates them together, separating each element with a comma

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

console.log(numbers.toString()); // Output: "1,2,3,4,5"

**12. Array join():**

**join()** method is used to join the elements of an array with a different separator, you can use the **join()** method, passing the desired separator as an argument:

let fruits = ['apple', 'banana', 'orange'];

console.log(fruits.join(' - ')); // Output: "apple - banana - orange"

**13. Array copyWithin():**

    - `copyWithin()` method copies a sequence of elements within the array to the position starting at the target index.

# let arr = [10, 20, 30, 40, 50];

# arr.copyWithin(0, 3); // Copies elements starting from index 3 to the beginning

# console.log(arr); // Output: [40, 50, 30, 40, 50]

**14. Array flat():**

    - `flat()` method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

     let arr=["a", "b", "c" ,"d","e",["f", ["g", ["h","i"]]]];

       newarr= arr.flat(4)

        console.log(newarr)

**Search methods**

**indexOf()** method in JavaScript is used to search for an element within an array. It returns the index of the first occurrence of the specified element, or -1 if the element is not found.

let fruits = ['apple', 'banana', 'orange', 'banana', 'grape'];

console.log(fruits.indexOf('banana')); // Output: 1

**lastIndexOf()** method in JavaScript is similar to the **indexOf()** method, but it searches for the last occurrence of a specified element within an array.

let fruits = ['apple', 'banana', 'orange', 'banana', 'grape'];

console.log(fruits.lastIndexOf('banana')); // Output: 3 (last occurrence of 'banana' is at index 3)

**includes()** method in JavaScript is used to determine whether an array contains a specific element. It returns **true** if the array contains the element, and **false** otherwise.

let fruits = ['apple', 'banana', 'orange', 'grape'];

console.log(fruits.includes('banana')); // Output: true (array contains 'banana')

console.log(fruits.includes('kiwi')); // Output: false (array does not contain 'kiwi')

**Objects**

An object in JavaScript is a collection of data in key-value pairs where each key is a string (or a Symbol) and each value can be of any data type, including other objects, functions, arrays, and primitive data types like strings, numbers, and booleans. Objects are created using curly braces **{}**.

**Creating Objects:**

1. **Literal notation:**

let person = { name: "John", age: 30 };

1. **Using the Object constructor:**

let person = new Object();

person.name = "John";

person.age = 30;

**Accessing Object Properties:**

You can access object properties using dot notation or square bracket notation:

console.log(person.name); // Dot notation

console.log(person['age']); // Square bracket notation

**Adding and Modifying Properties:**

person.gender = "Male"; // Adding a new property

person.age = 31; // Modifying an existing property

**Deleting Properties:**

delete person.age;

**Object Methods:**

Methods are functions stored as object properties.

let person = {

name: "John",

greet: function() { console.log("Hello, my name is " + this.name); } };

person.greet(); // Output: Hello, my name is John

**Object Iteration:**

You can iterate over an object's properties using loops or methods like **Object.keys()**, **Object.values()**, or **Object.entries()**.

for (let key in person) { console.log(key + ": " + person[key]); } Object.keys(person).forEach(function(key) { console.log(key + ": " + person[key]); });

**Methods in js**

1. **Object.keys()**: Returns an array of a given object's property names.

    const obj = { a: 1, b: 2, c: 3 };

    console.log(Object.keys(obj)); // Output: ["a", "b", "c"]

1. **Object.values():** Returns an array of a given object's own enumerable property values.Top of Form

    const obj = { a: 1, b: 2, c: 3 };

    console.log(Object.values(obj)); // Output: [1, 2, 3]

1. **Object.entries()**: Returns an array of a given object's own enumerable string-keyed property [key, value] pairs.

    const obj = { a: 1, b: 2, c: 3 };

    console.log(Object.entries(obj)); // Output: [["a", 1], ["b", 2], ["c", 3]]

1. **Object.assign():** Copies the values of all enumerable own properties from one or more source objects to a target object.

    const target = { a: 1, b: 2 };

    const source = { b: 3, c: 4 };

    Object.assign(target, source);

    console.log(target); // Output: { a: 1, b: 3, c: 4 }

1. **Object.create():** Creates a new object with the specified prototype object and properties.

    const obj = Object.create({ foo: 1 });

    console.log(obj.foo); // Output: 1

1. **Object.freeze():** Freezes an object, preventing new properties from being added to it, existing properties from being removed, and values from being changed.

  const obj = { a: 1, b: 2 };

    Object.freeze(obj);

    obj.c = 3; // This will not add 'c' to the object

    console.log(obj); // Output: { a: 1, b: 2 }

1. **Object.seal():** Seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable.

    const obj = { a: 1, b: 2 };

    Object.seal(obj);

    delete obj.a; // This will not delete 'a' from the object

    console.log(obj); // Output: { a: 1, b: 2 }

**8.Object.hasOwnProperty():** Returns a boolean indicating whether the object has the specified property as its own property

console.log(obj.hasOwnProperty("name"));

**How to itterate objects**

**for...in Loop**

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

for ( key in obj) {

  console.log( key + obj[key]);

}

**for...of Loop with Object.keys()**

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

for ( key of Object.keys(obj)) {

  console.log(`${key}: ${obj[key]}`);

}

**for...of Loop with Object.values()**

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

for ( value of Object.values(obj)) {

  console.log(value);

}

**for...of Loop with Object.entries()**

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

for ( [key, value] of Object.entries(obj)) {

  console.log(`${key}: ${value}`);

}

**Number methods**

1. **toFixed()** method formats a number using fixed-point notation, which means it returns a string representation of the number with a specified number of decimal places. This is useful for rounding numbers to a certain number of decimal places.

let num = 123.45678;

console.log(num.toFixed(2)); // "123.46" - rounded to two decimal places

console.log(num.toFixed(3)); // "123.457" - rounded to three decimal places

console.log(num.toFixed(0)); // "123" - no decimal places, rounds to nearest integer

let num1 = 150;

console.log(num1.toFixed(2)); // "150.00" - Adds two decimal places with zeros

console.log(num1.toFixed(0)); // "150" - No change

1. **parseInt()**: Parses a string argument and returns an integer.

let str = "123";

console.log(parseInt(str)); // Output: 123

1. **parseFloat()**: Parses a string argument and returns a floating point number.

// Example 1: Basic Parsing

console.log(parseInt("42")); // 42 (simple integer parsing)

console.log(parseFloat("42")); // 42 (simple float parsing)

console.log(parseFloat("42.5")); // 42.5 (floating-point parsing)

// Example 2: Handling Leading and Trailing Whitespaces

console.log(parseInt("   123   ")); // 123 (leading/trailing spaces ignored)

console.log(parseFloat("   123.45   ")); // 123.45 (leading/trailing spaces ignored)

// Example 3: Parsing with Non-Numeric Characters

console.log(parseInt("123abc")); // 123 (parsing stops at "abc")

console.log(parseFloat("123.45abc")); // 123.45 (parsing stops at "abc")

// Example 4: Strings Starting with Non-Numeric Characters

console.log(parseInt("abc123")); // NaN (no leading numeric characters)

console.log(parseFloat("abc123.45")); // NaN (no leading numeric characters)

// Example 5: Handling Float Strings in parseInt()

console.log(parseInt("3.14")); // 3 (truncates the decimal part)

console.log(parseFloat("3.14")); // 3.14 (returns the full float)

// Example 6: Parsing Strings with Exponential Notation

console.log(parseInt("1e4")); // 1 (stops at "e")

console.log(parseFloat("1e4")); // 10000 (interpreted as 1 \* 10^4)

// Example 8: Large and Small Numbers

console.log(parseInt("0.0001")); // 0 (fractional part ignored)

console.log(parseFloat("0.0001")); // 0.0001 (floating-point number)

1. **isNaN()**: Checks if a value is NaN (Not-a-Number). If it is number it returns false if it is not a number it returns true.

**NUMBER** – False

**Not a Number** - True

console.log(isNaN("hello")); // true

Output: true console.log(isNaN(123)); // Output: false

**5. Number method**

The Number constructor converts a value to a number.

console.log(Number("123")); // 123

console.log(Number("123abc")); // NaN

console.log(Number(true)); // 1

console.log(Number(false)); // 0

console.log(Number(null)); // 0

console.log(Number(undefined)); // NaN

**Type Coercion:**

**isNaN()** first tries to convert the parameter to a number, and then tests if the resulting value is NaN.

isNaN(NaN); // true

isNaN(undefined); // true

isNaN({}); // true

isNaN(true); // false

isNaN(false);//false

isNaN(null); // false

isNaN(37); // false

// Strings

isNaN("37"); // false: "37" is converted to the number 37 which is not NaN

isNaN("37.37"); // false: "37.37" is converted to the number 37.37 which is not NaN

isNaN("37,5"); // true

isNaN("123ABC","jhkhk"); // true: Number("123ABC") is NaN

isNaN(""); // false: the empty string is converted to 0 which is not NaN

isNaN(" "); // false: a string with spaces is converted to 0 which is not NaN

// Dates

isNaN(new Date()); // false; Date objects can be converted to a number (timestamp)

isNaN(new Date().toString()); // true; the string representation of a Date object cannot be parsed as a number

// Arrays

isNaN([]); // false; the primitive representation is "", which coverts to the number 0

isNaN([1]); // false; the primitive representation is "1"

isNaN([1, 2]); // true; the primitive representation is "1,2", which cannot be parsed as number

**Math methods**

**1. Math.abs()**

Returns the absolute value of a number.

  console.log(Math.abs(10)); // 10

        console.log(Math.abs(-10)); // 10

        console.log(Math.abs(0)); // 0

        console.log(Math.abs(-0)); // 0

        console.log(Math.abs("-42")); // 42 (string converted to number)

        console.log(Math.abs(null)); // 0 (null converted to 0)

        console.log(Math.abs("Hello")); // NaN (string that can't be converted to a number)

**2. Math.ceil()**

Rounds a number **up** to the next largest integer.

        console.log(Math.ceil(4.2)); // 5

        console.log(Math.ceil(-4.2)); // -4

        console.log(Math.ceil(0)); // 0

        console.log(Math.ceil(7.004)); // 8

        console.log(Math.ceil(-7.004)); // -7

**3. Math.floor()**

Rounds a number **down** to the previous largest integer.

        console.log(Math.floor(4.7)); // 4

        console.log(Math.floor(-4.7)); // -5

        console.log(Math.floor(0)); // 0

        console.log(Math.floor(7.999)); // 7

        console.log(Math.floor(-7.999)); // -8

**4. Math.round()**

Rounds a number to the nearest integer. If the fractional part is 0.5 or greater, the argument is rounded to the next higher integer.

        console.log(Math.round(4.5)); // 5

        console.log(Math.round(4.4)); // 4

        console.log(Math.round(-4.5)); // -4

        console.log(Math.round(-4.6)); // -5

        console.log(Math.round(7.999)); // 8

        console.log(Math.round(-7.999)); // -8

**5. Math.trunc()**

Returns the integer part of a number by removing any fractional digits.

    console.log(Math.trunc(4.9)); // 4

    console.log(Math.trunc(-4.9)); // -4

    console.log(Math.trunc(0)); // 0

    console.log(Math.trunc(7.004)); // 7

    console.log(Math.trunc(-7.004)); // -7

**6. Math.max()**

Returns the largest of zero or more numbers

    console.log(Math.max(1, 2, 3)); // 3

    console.log(Math.max(-1, -2, -3)); // -1

    console.log(Math.max(1, 2, 3, 10, 20)); // 20

**7. Math.min()**

Returns the smallest of zero or more numbers.

    console.log(Math.min(1, 2, 3)); // 1

    console.log(Math.min(-1, -2, -3)); // -3

    console.log(Math.min(1, 2, 3, 10, 20)); // 1

**8. Math.pow()**

Returns the base raised to the power of the exponent.

    console.log(Math.pow(2, 3)); // 8 (2^3)

    console.log(Math.pow(5, 2)); // 25 (5^2)

    console.log(Math.pow(4, 0.5)); // 2 (square root of 4)

    console.log(Math.pow(-7, 2)); // 49 (negative base, even exponent)

**9. Math.sqrt()**

Returns the square root of a number.

    console.log(Math.sqrt(16)); // 4

    console.log(Math.sqrt(9)); // 3

    console.log(Math.sqrt(0)); // 0

**10. Math.random()**

Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).

    console.log(Math.random()); // Random number between 0 and 1

    console.log(Math.random() \* 10); // Random number between 0 and 10

    console.log(Math.floor(Math.random() \* 10)); // Random integer between 0 and 9

    console.log(Math.floor(Math.random() \* 100) + 1); // Random integer between 1 and 100

**Date Methods**

**Creating Dates**

Var now=new Date();

**Specific Date and Time**

let specificDate = new Date('2024-06-12T10:20:30Z');

**Getting Date Components**

let year = now.getFullYear();// year

let month = now.getMonth(); //0 -11

let day = now.getDate(); // 1-31

**Day of the Week**

let dayOfWeek = now.getDay(); // 0-6 (0 = Sunday, 6 = Saturday)

**Hours, Minutes, Seconds, Milliseconds**

let hours = now.getHours(); // 0-23

let minutes = now.getMinutes(); // 0-59

let seconds = now.getSeconds(); // 0-59

let milliseconds = now.getMilliseconds(); // 0-999

**Setting Date Components**

now.setFullYear(2025);

now.setMonth(6); // July

now.setDate(15);

**Set Hours, Minutes, Seconds, Milliseconds**

now.setHours(15);

now.setMinutes(30);

now.setSeconds(45);

now.setMilliseconds(500);

**Formatting Date and Time**

JavaScript provides methods to format dates as strings in different formats:

1. **toDateString()**

let dateStr = now.toDateString(); // e.g., "Wed Sep 22 2024"

1. **toTimeString()**

let timeStr = now.toTimeString(); // e.g., "15:30:45 GMT+0530 (India Standard Time)"

1. **toLocaleDateString()**

let localDateStr = now.toLocaleDateString(); // e.g., "9/22/2024" in US format

1. **toLocaleTimeString()**

let localTimeStr = now.toLocaleTimeString(); // e.g., "3:30:45 PM" in US format

These methods allow you to present dates in a human-readable or standardized format.

**Age calculator**

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <input type="date" id="el1" />

    <button onclick="fun()">clicke me</button>

    <script>

      function fun() {

        var d = document.getElementById("el1").value;

        var olddate = new Date(d);

        var newdate = new Date();

        var year = newdate.getFullYear() - olddate.getFullYear();

        var milliseconds = newdate - olddate;

        var days = Math.floor(milliseconds / (60 \* 60 \* 24 \* 1000));

        console.log(days);

        console.log(year);

      }

    </script>

  </body>

</html>

**Callbacks**

callback is a function defintion is passed as an argument to another function and is executed after some operation has been completed. This is a powerful feature that allows for asynchronous programming, enabling tasks to run concurrently without blocking the main execution thread.

**Defining the Callback:**

      // Step 1: Define the callback function

      function myCallback() {

        console.log("Callback function executed!");

      }

The callback function can be defined separately or inline.

      // Defining separately

      function myCallback() {

        console.log("Callback executed!");

      }

      doSomething(myCallback);

      // Defining inline

      doSomething(function () {

        console.log("Callback executed!");

      });

**Define a Function that Takes Another Function as a Parameter(HOF)**

**Execute the Callback**

The outer function calls the callback function at the appropriate time.

      // Step 2: Define a function that takes another function as a parameter

      function doSomething(callback) {

        console.log("Doing something...");

        // Step 3: Execute the callback function

        callback();

      }

**Pass the Callback Function as a argument:**

      // Step 4: Pass the callback function as an argument

      doSomething(myCallback);

Ex-1

function myFirst() {

  console.log("Hello");

}

function mySecond(a) {

  console.log("Goodbye");

}

var a=myFirst();

mySecond(a);

//Hello

//Goodbye

Ex-2

function myFirst() {

  console.log("Hello");

}

function mySecond() {

    myFirst()

  console.log("Goodbye");

}

mySecond();

The problem with the first example above, is that you have to call two functions to display the result.

The problem with the second example, is that you cannot prevent the function from displaying the result when it invokes every time.

function myFirst() {

  console.log("Hello");

}

function mySecond(a) {

  console.log("Goodbye");

}

mySecond(myFirst());//can call two functions at a time

mySecond()//one function

**Usage of Callbacks**

Callbacks are commonly used in situations where you want to perform tasks asynchronously, such as:

* Event Handling
* Array Methods
* Higher-Order Functions
* Asynchronous Operations

Array for each

**Array.forEach()** is a method in JavaScript used to iterate over elements in an array. It executes a provided function once for each array element

**array.forEach(function(value, index, array) {**

// Your code here

**});**

**value:** The current item in a array.

**index (optional):** The index of the item in the array.

**array (optional):** The array that forEach()

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

numbers.forEach(function(val, index) {

    console.log(`Element at index ${index} is ${val }`);

});

**//output**

// Element at index 0 is 1

// Element at index 1 is 2

// Element at index 2 is 3

// Element at index 3 is 4

// Element at index 4 is 5

You can also use arrow functions for a more concise syntax:

numbers.forEach((number, index) => {

    console.log(`Element at index ${index} is ${number}`);

});

One important thing to note about forEach() is that it **doesn't return anything**. It simply **iterates over the array**. If you need to transform the elements of the array and create a new array based on those transformations, you might want to use methods like map() instead.

array.forEach(function(element) {

return element \* 2; // This return statement has no effect

});

Map method

The map() method in JavaScript is used to create a new array by calling a provided function on every element in the calling array. It doesn't change the original array; instead, it returns a new array with the results of applying the provided function to each element.

const newArray = array**.map**(function callback(**currentValue**, index, array) {

// Return element for newArray

});

**value:** The current item in a array.

**index (optional):** The index of the item in the array.

**array (optional):** The array that forEach()

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

const doubledNumbers = numbers.map(function(number) {

  return number \* 2; // Return value determines the value

});

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

You can also use arrow functions for more concise syntax:

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

const doubledNumbers = numbers.map(number => number \* 2);

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

**Filter method**

The filter() method in JavaScript is used to create a new array with all elements that pass a certain condition. It doesn't change the original array; instead, it returns a new array containing only the elements for which the provided filtering function returns true.

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

const filteredNumbers = numbers.filter(function(number) {

  return number >3;

});

console.log(filteredNumbers); // Output: [4, 5]

You can also use arrow functions for more concise syntax:

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

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

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

Use **map method** when you need to transform each element of an array and create a new array with the transformed values. when you want to perform operatons

Use **filter method** when you need to filter elements from an array based on some criteria and create a new array with only the elements that meet that criteria.when you want to perform filteration.

Use **for each method** When you need to perform an action for each element of an array without necessarily creating a new array or transforming the elements. When you want to perform itterations.

**Reduce method**

The **reduce()** method in JavaScript is used to reduce the elements of an array to a single value.

Accumulator (acc): The accumulated value computed from previous iterations.

Current Value (val): The current element being processed in the array.

Current Index (ind) (optional): The index of the current element being processed in the array.

Source Array (src) (optional): The array reduce() was called upon.

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

const sum = numbers.reduce((accumulator, currentValue) => {

  return accumulator + currentValue;

}, 0);

console.log(sum); // Output: 15 (1 + 2 + 3 + 4 + 5)

**reduce()** can be used for various tasks, such as calculating the maximum or minimum value in an array, concatenating elements into a string, or performing more complex operations. It's a powerful method for aggregating data in arrays.

**Reduceright method**

The reduceRight() method in JavaScript is used to reduce the elements of an array to a single value, but it processes the array from right to left.

* **Accumulator (acc)**: The accumulated value computed from previous iterations.
* **Current Value (cur)**: The current element being processed in the array.
* **Current Index (ind)** (optional): The index of the current element being processed in the array.
* **Source Array (src)** (optional): The array reduceRight() was called upon.

    const numbers = [1, 2, 3, 4, 5];

    const sum = numbers.reduceRight((accumulator, currentValue) => {

    return accumulator + currentValue;

    }, 0);

    console.log(sum); // Output: 15 (5 + 4 + 3 + 2 + 1)

 The reduceRight() method can be used for various tasks, such as calculating the maximum or minimum value in an array, concatenating elements into a string, or performing more complex operations.

 It's a powerful method for aggregating data in arrays, especially when the order of operations from right to left is significant.

**Sort method**

The **sort()** method in JavaScript is used to sort the elements of an array in place and returns the sorted array.

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

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

console.log(numbers); // Output: [1, 2, 3, 4, 5]

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

fruits.sort();

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

// method will sort the elements alphabetically as strings.

In this example, the **compareFunction** **(a, b) => a - b** sorts the numbers in ascending order. If **a - b** is negative, **a** comes before **b**, if it's positive, **b** comes before **a**, and if it's zero, the order remains unchanged.

Top of Form

The **some()** and **every()** methods in JavaScript are both used to check the elements of an array against certain conditions.

**some() Method:**

Purpose: It checks if at least one element in the array satisfies the provided testing function. It returns true if any element passes the test; otherwise, it returns false.

Syntax: array.some(callback(element, index, array))

Return Value: true if at least one element passes the test; otherwise, false.

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

const numbers2 = numbers.some(number => number >3);

console.log(number2); // Output: true

**every() Method**:

* Purpose: It checks if all elements in the array satisfy the provided testing function. It returns **true** if all elements pass the test; otherwise, it returns **false**.
* Syntax: **array.every(callback(element, index, array))**
* Return Value: **true** if all elements pass the test; otherwise, **false**.

    const numbers = [1, 2, 3, 4, 5];

    const numbers2 = numbers.every(number => number >3);

    console.log(numbers2); // Output: false

**DOM- Document Object Model**

DOM is a standard **object** model that allows programs and scripts to dynamically access and update the content, structure, and style of a document

**Document Object Model** (**DOM**) connects web pages to scripts languages by representing the structure of a document

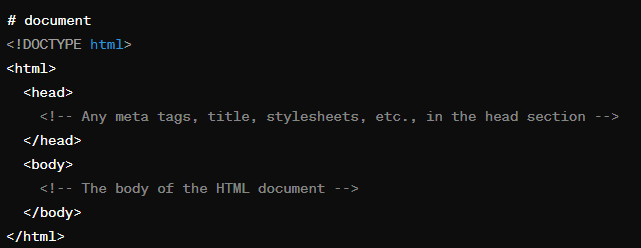
The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree. With them, you can change the document's structure, style, or content.

Here's a breakdown of some key concepts related to the JavaScript DOM:

1. **Document**: The top-level object in the DOM hierarchy, representing the entire HTML document. It serves as an entry point to access and manipulate the document's content.

console.log(document);

Logging **document** to the console in JavaScript will display the entire Document Object Model (DOM) of the current HTML page.



1. **Node**: Every part of an HTML document, such as elements, attributes, and text, is represented by a node in the DOM tree. Nodes can be of different types, including element nodes, text nodes etc.
2. **Element**: Elements are the building blocks of an HTML document, such as **<div>**, **<p>**, **<span>**, etc. They are represented as element nodes in the DOM tree.
3. **Attributes**: Elements can have attributes like **id**, **class**, **src**, etc. These attributes are accessible and modifiable through the DOM.
4. **Methods for Accessing Elements:**
   * document.getElementById(): Retrieves an element by its unique ID.
   * document.getElementsByClassName(): Retrieves elements by their class name.
   * document.getElementsByTagName(): Retrieves elements by their tag name.
   * document.querySelector(): Retrieves the first element that matches a CSS selector.
   * document.querySelectorAll(): Retrieves all elements that match a CSS selector
5. **Manipulating Elements**:
   * Changing element attributes (**element.attribute**).
   * Changing element content (**element.innerHTML**, **element.innerText**, **element.textContent**).
   * Adding or removing classes (**element.classList.add()**, **element.classList.remove()**).
   * Creating new elements (**document.createElement()**).
   * Appending or removing child nodes (**parentNode.appendChild()**, **parentNode.removeChild()**).
6. **Event Handling**: DOM allows attaching event handlers to elements to listen for specific events like click, hover, keypress, etc., and execute JavaScript code in response to those events.
7. **Traversing the DOM**: You can navigate through the DOM tree by accessing parent, child, or sibling nodes using properties like **parentNode**, **childNodes**, **firstChild**, **lastChild**, **nextSibling**, and **previousSibling**.

### Using the children or childNodes Properties

The **children** property selects all child elements that are directly under a given element. Here's an example of the **children** property in action:

 <div **class**="wrapper-1">  
            <ul **class**="apple-list">  
                <li **class**="apple">Apples</li>  
                <li **class**="orange">Oranges</li>  
                <li **class**="avocado">Avocados</li>  
                <li **class**="grape">  
                    Grapes  
                    <**ul**>  
                        <li **class**="type-1">Moon drops</li>  
                        <**li**>Sultana</**li**>  
                        <**li**>Concord</**li**>  
                        <**li**>Crimson Seedless</**li**>  
                    </**ul**>  
                </**li**>  
                <li **class**="banana">Bananas</li>  
            </**ul**>  
 </**div**>

**const** appleList = document.querySelector('.apple-list');  
**const** apples = appleList.children;   
console.log(apples);

### Using Special lastChild and firstChild Properties

These two methods are not as robust as the first two. As their names suggest, the **lastChild**and **firstChild** properties return an element's last and first child nodes

**const** appleList = document.querySelector('.apple-list');  
**const** firstChild = appleList.firstChild;  
**const** lastChild = appleList.lastChild;

### Using parentElement or parentNode

Both **parentElement** or **parentNode** properties let you select the selected element's parent node one level up. The critical difference is that **parentElement**only chooses the parent node that is an element. On the other hand, **parentNode** can select a parent regardless of whether it's an element or a different node type.

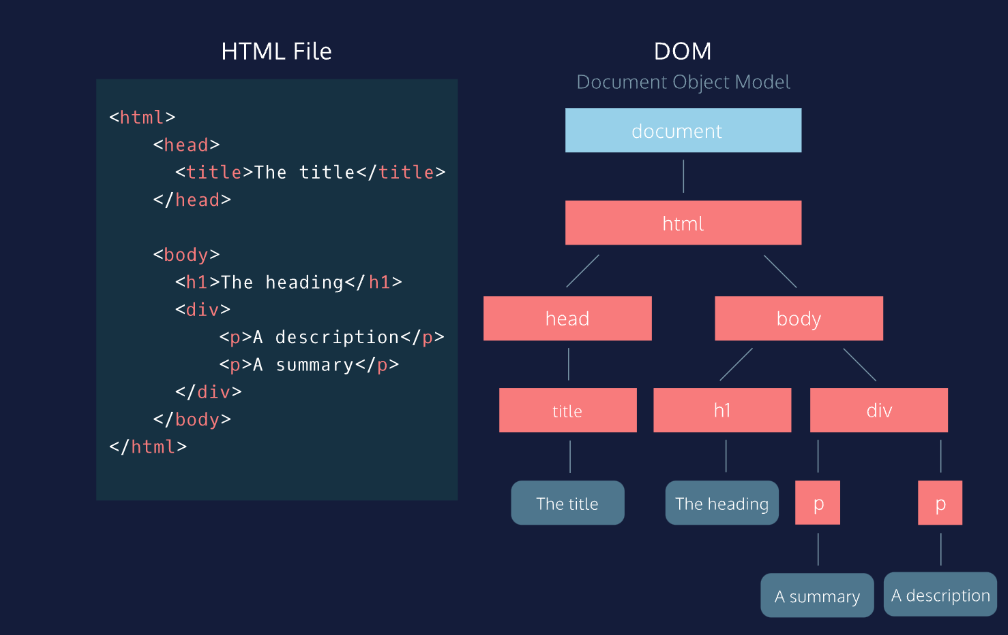
In the code sample below, we use **parentElement** to select the div with the "wrapper-1" class from "apple-list":

**const** appleList = document.querySelector('.apple-list');  
**const** parentDiv = appleList.parentElement;   
console.log(parentDiv); // displays div element with class wrapper-1

## Traversing the DOM Sideways

There are two methods available for walking the DOM sideways. You can use **nextElementSibling** or **previousElementSibling**. Use **nextElementSibling** to select the following sibling element and **previousElementSibling** to select the previous sibling.

**const** orange = document.querySelector('.orange');  
**const** apple = orange.previousElementSibling;  
**const** avocado = orange.nextElementSibling;



**DOM**

**It is a object model used to manipulate the document and there are two ways to create document object**

1) Field Names – document level object creation

2) Methods – element level object creation

**Create dom object by field names**

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **DOM** |
| document.body  document.head  document.scripts | Returns all <body> element  Returns the <head> element  Returns all <script> elements | 1 |
| document.anchors | Returns all <a> elements that have a name attribute | 1 |
| document.forms | Returns all <form> elements | 1 |
| document.images | Returns all <img> elements | 1 |
| document.links | Returns all <area> and <a> elements that have a href attribute | 1 |
|  |  | 3 |
| document.title | Returns the <title> element | 1 |
|  |  |  |

**Get methods using dom**

**1)document.getElementById()**: Retrieves an element by its unique ID

<div id="myDiv"></div>

var elementById = document.getElementById("myDiv"); //line gets the element by id

console.log(elementById);//below is the ouput



**2)document.getElementsByClassName()**: Retrieves elements by their class name.

  <p class="myClass">Paragraph 1</p>

  <p class="myClass">Paragraph 2</p>

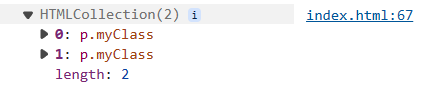
var elementsByClassName = document.getElementsByClassName("myClass");

console.log(elementByClassname);

//here the point to note is classnames are always in collections

You can get the element by their index numbers

Var elementsByClassName= document.getElementByClassName(“myClass”)[0]



**3) document.getElementsByTagName()**: Retrieves elements by their tag name.

  <h1>Heading</h1>

  <p>Paragraph 1</p>

  <p>Paragraph 2</p>

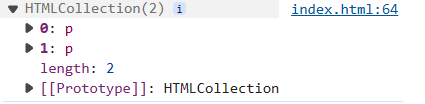
  var elementsByTagName = document.getElementsByTagName("p");

  console.log(elementsByTagName);

//here the point to note is tagnames are always in collections

You can get the element by their index numbers

Var elementsByTagName= document.getElementByTagName(“p”)[0]



**4)Accessing Elements by CSS Selector:**

  <div class="container">

    <p class="para">Paragraph 1</p>

    <p class="para">Paragraph 2</p>

</div>

**querySelector**() method allows you to select the first element in the document

var elementBySelector = document.querySelector(".para");//selects by classname

var myDiv = document.querySelector("#myDiv");//select by id

var elselector = document.querySelector("div");//select by element name

**querySelectorAll** -iIt operates similarly to **querySelector**(), but instead of returning only the first matching element, it returns a list of all matching elements.

var paragraphs = document.querySelectorAll(".para");//select all elements by class names

var divs = document.querySelectorAll("div");//select all div elements in a collections

**Get content of the html**

**innerText** and **innerHTML** are properties of DOM elements in JavaScript that deal with the content of HTML elements

**innerText:**

* **innerText** is a property that represents the visible text content of an element.
* It retrieves the text content of the element, excluding any HTML tags.

<div id="myDiv">This is <span>some</span> text content.</div>

var element = document.getElementById("myDiv");

var text = element.innerText;

console.log(text); // Output: "This is some text content."

**innerHTML:**

* **innerHTML** is a property that represents the HTML content of an element.
* It retrieves or sets the HTML markup within the element, including any nested elements and tags.
* It can be used to dynamically change the structure and content of an element.

var element = document.getElementById("myDiv");

var html = element.innerHTML;

console.log(html); // Output: "This is <span>some</span> text content."

**How to modify existing content**

    // Select the element by its ID

    var paragraph = document.getElementById("myParagraph");

    // Update the text content using innerText

    paragraph.**innerText** = "Updated text!";

**How to apply styles using dom**

 // Step 1: Access the element where you want to append the text node

 var myDiv = document.getElementById("myDiv");

// Step 2: Apply styles

myDiv.style.backgroundColor="red";

apply styles using document.getElementById(“myDIv”).style.backgroundColor=”red”;

**How to create element and how to append element in dom**

    // Create a new paragraph element

    var newParagraph= document.createElement("p");

    // Set innertext or other properties if needed

    newParagraph.innerText = "This is a dynamically created paragraph.";

    // Append the paragraph to the document body

    document.body.appendChild(newParagraph);

* A new paragraph element is created using **document.createElement("p")**.
* The **innerText** property of the newly created paragraph element is set to "This is a dynamically created paragraph."
* The paragraph element is appended to the document body using **document.body.appendChild(newParagraph)**.

**Appendchild and Append**

Append and appendChild methods are used in JavaScript to add nodes to the DOM, but they have some differences in terms of usage, accepted parameters, and behavior:

**appendChild**

**syntax :**

parentNode.appendChild(newChild);

**Parameters**:

newChild: A single node (an element, text node, or any other node) that will be appended as the last child of parentNode

**Behavior:**

If the newChild is already in the DOM, it will be removed from its current position and moved to the new position.

Only accepts a single node.

**Append**

**syntax :**

parentNode.append(node1, node2, node3);

**Parameters**:

nodes: One or more nodes or strings that will be appended as the last children of parentNode.

**Behavior:**

Can append multiple nodes and/or strings at once.

If a string is provided, it will be added as a text node.

Allows appending a combination of nodes and text.

**How to create textNode**

 // Step 1: Access the element where you want to append the text node

 var myDiv = document.getElementById("myDiv");

// Step 2: Create a text node

var textNode = document.createTextNode("This is a dynamically created text node.");

// Step 3: Append the text node to the element

myDiv.appendChild(textNode);

1. Access the element where you want to append the text node.
2. Create a text node using **document.createTextNode()**.
3. Append the text node to the desired element.

**How to change attribute values by using setAttribute**

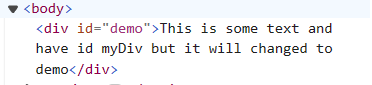
<div id="myDiv">This is some text and have id myDiv but it will changed to demo</div>

 var a=document.getElementById("myDiv").setAttribute("id","demo");

console.log(document)//can inspect and check weather it was changed or not

we can change the attribute by using .setAttribute.(“attribute name”,”attribute value”)

//output

****

**How to get attribute**

We can get the element attribute by using get attribute method in dom

<img  id="myElement" src="https://www.w3schools.com/myl-green-off.png" alt="lkdj">

 var element = document.getElementById("myElement");

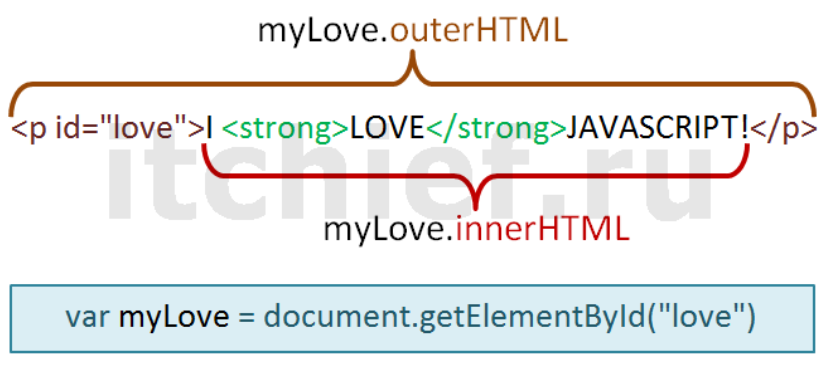
// To get the value of an attribute, such as "src" for an image element:

var srcValue = element.getAttribute("src");

console.log(srcValue);

//output





Classlist add and remove

The classList property is an incredibly useful method for manipulating the classes of HTML elements in JavaScript. It allows you to add, remove, toggle, and check classes without altering the entire className string. Here's an in-depth explanation of how classList.add() and classList.remove() work.

**1. What is classList?**

* classList is a property that returns a live DOMTokenList collection of the classes of an element.
* You can think of classList as a way to interact with the classes applied to an HTML element, allowing you to add, remove, or toggle CSS classes dynamically.

**2. Syntax**

* The syntax for accessing classList is as follows:

element.classList

**classList methods include:**

* .add()
* .remove()
* .toggle()

**3. Adding a Class: classList.add()**

The add() method adds one or more class names to the element. If the class already exists, it won't be added again (no duplicates).

# element.classList.add(className1, className2, ..., classNameN);

**4. Removing a Class: classList.remove()**

The remove() method removes one or more class names from the element. If the class does not exist, nothing happens.

# element.classList.remove(className1, className2, ..., classNameN);

**5. Common Use Cases**

**A. Toggling Classes (With classList.toggle())**

* Sometimes, you may want to add a class if it’s not present or remove it if it is. This can be done with the toggle() method.

# element.classList.toggle('className');

**B. Checking If an Element Has a Class (classList.contains())**

* To check if an element has a certain class, use the contains() method.

# if (element.classList.contains('className')) { // do something }

**Types of Events**

1. **Mouse Events:**
   * **click**: Occurs when a mouse button is clicked.
   * **dblclick**: Occurs when a mouse button is double-clicked.
   * **mouseover**: Occurs when the mouse pointer enters the area of an element.
   * **mouseout**: Occurs when the mouse pointer leaves the area of an element.
   * **mousemove**: Occurs when the mouse pointer is moved over an element.
2. **Keyboard Events:**
   * **keydown**: Occurs when a keyboard key is pressed down.
   * **keyup**: Occurs when a keyboard key is released.
   * **keypress**: Occurs when a keyboard key is pressed and released.
3. **Form Events:**
   * **submit**: Occurs when a form is submitted.
   * **change**: Occurs when the value of an input element changes.
   * **focus**: Occurs when an element receives focus.
   * **blur**: Occurs when an element loses focus.
4. **Window Events:**
   * **load**: Occurs when a resource and its dependent resources have finished loading.
   * **resize**: Occurs when the browser window is resized.
   * **scroll**: Occurs when the user scrolls through a webpage.

**Event handlers:**

Event handlers are functions in JavaScript that are responsible for handling specific types of events. They define what should happen when a particular event occurs. Event handlers are associated with HTML elements and are triggered when the corresponding event takes place.

**1.Inline Event Handlers:** Inline event handlers are defined directly within the HTML markup using the **on** attribute followed by the event name.

<button **onclick="myFunction()"**>Click me</button>

**2.DOM Event Handlers:** DOM event handlers are assigned to HTML elements using JavaScript code.

You can attach event handlers using methods like **addEventListener()**

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

**button.addEventListener('click', myFunction);**

**Event listeners:**  
Event listeners in JavaScript are functions that wait for a specific event to occur and then execute code in response to that event.

**Using addEventListener() Method:** The **addEventListener()** method attaches an event listener to an HTML element. It takes three parameters: the event name, the function to be executed when the event occurs, and an optional boolean value indicating whether to use capturing or bubbling (default is **false**, indicating bubbling).

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

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

console.log('Button clicked!');

});

**Removing Event Listeners:** You can remove event listeners using the **removeEventListener()** method. It requires the same parameters as **addEventListener()**.

function handleClick() {

console.log('Button clicked!');

}

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

button.addEventListener('click', handleClick);

// Later, if you want to remove the event listener

button.removeEventListener('click', handleClick);

**Event bubbling**  
Event bubbling is a concept in JavaScript (and many other programming languages) where an event triggered on a nested element will "bubble up" through its ancestors in the DOM hierarchy until it reaches the root of the document.

**e.stopPropagation()** is a method used within event handlers to prevent the event from bubbling up the DOM tree.

**Event capturing**  
Event capturing is another phase of event propagation in the DOM (Document Object Model) in addition to event bubbling. event capturing starts from the root element and moves down to the target element.

<div id="outer">

  <div id="inner">

    <button id="myButton">Click me</button>

  </div>

</div>

<script>

  // Event listener on the outer div with capturing

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

    console.log('Outer div clicked (capturing)');

  }, true);

  // Event listener on the inner div with capturing

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

    console.log('Inner div clicked (capturing)');

  }, true);

  // Event listener on the button without capturing (using default, which is bubbling)

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

    console.log('Button clicked (bubbling)');

  });

</script>

**Event Bubbling:**

   - Event bubbling is a mechanism where when an event is triggered on a nested element inside another element, the event 'bubbles up' through its ancestors.

   - By default, most events bubble.

   - You can stop the bubbling phase using `event.stopPropagation()`.

  <div id="parent">

        <button id="child">

            click me

        </button>

    </div>

    <script>

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

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

    child.addEventListener("click", function(event){

        event.stopPropagation()// this will prevent the bubbling to the parent

        console.log("child is clicked");

    })

    parent.addEventListener("click", function(){

        console.log("parent is clicked")

    })

    </script>

**Event Capturing:**

   - Event capturing is the opposite of event bubbling.

   - During the capturing phase, the event is first captured by the outermost element and then propagated to the innermost element.

   - You can listen to events during the capturing phase by passing `true` as the third parameter to `addEventListener()`.

  <div id="parent">

        <button id="child">

            click me

        </button>

    </div>

    <script>

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

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

    child.addEventListener("click", function(event){

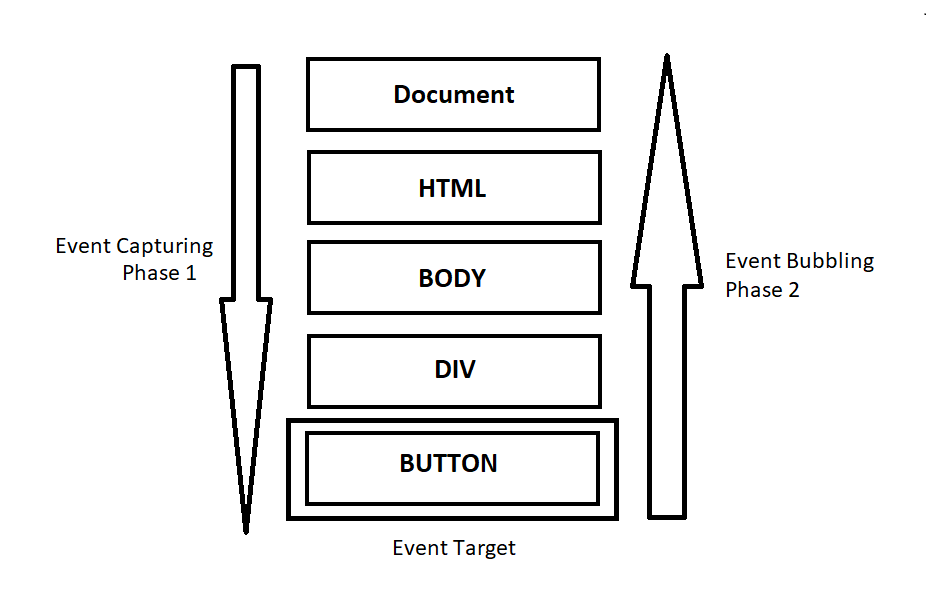
        console.log("child is clicked");

    })

    parent.addEventListener("click", function(){

        console.log("parent is clicked")

    },true)



**Event Binding:**

   - Event binding refers to the process of attaching event listeners to DOM elements.

   - This is typically done using `addEventListener()` or by assigning event handler properties like `onclick`.



**Form Handling**

**1. Accessing Form Elements:**

    <form id="myForm">

        <input type="text" id="username">

        <input type="email" id="email">

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

    </form>

    <script>

        const form = document.getElementById('myForm');

        const usernameInput = document.getElementById('username');

        const emailInput = document.getElementById('email');

    </script>

**2. Setting Input Values:**

You can set input values using the value property:

    usernameInput.value = "John Doe";

    emailInput.value = "john@example.com";

**3. Getting Form Data:**

You can get form data when the form is submitted:

    form.addEventListener('submit', function(event) {

    event.preventDefault(); // Prevent form submission

    const formData = new FormData(form);

    // Access form data

    const username = formData.get('username');

    const email = formData.get('email');

    // Do something with the data

   });

**4. Changing Input Field Values**

You can change the value of input fields dynamically based on user actions or other events:

<input type="text" id="myInput">

<button onclick="changeInputValue()">Change Value</button>

<script>

function changeInputValue() {

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

input.value = 'New Value';

}

</script>

**5.Accessing through values**

    document.forms[0].addEventListener('submit', function(event){

    event.preventDefault(); // Prevent form submission

    console.log(event);//stores the lot of form info including child tags of form

    console.log(event.target[0].value);//prints the value of first child in the first form

    })

In above program document.forms[0] get the first form,

e.preventdefault prevents the form default submission

console.log(event) prints the lot of form data and event.target gets the childrens of the form in collection format.

**JavaScript Form Validation: In-Depth Guide**

Form validation is a crucial aspect of web development that ensures data entered by users is accurate, complete, and safe. It prevents the submission of invalid data and enhances the user experience by providing instant feedback. Below is an in-depth guide covering form validation techniques in JavaScript.

**1. Types of Form Validation**

* **Client-Side Validation**: Validation performed in the browser before form data is sent to the server.
* **Server-Side Validation**: Validation performed on the server after the form is submitted.

**2. Importance of Form Validation**

* Prevents submission of incorrect or incomplete data.
* Protects against security risks (e.g., SQL injection).
* Improves user experience by giving real-time feedback.
* Reduces server load by catching errors before submission.

**3. Basic Form Validation Workflow**

1. **Capture Form Elements**:
   * Access form fields using document.getElementById() or document.querySelector() methods.
2. **Attach Event Listeners**:
   * Use addEventListener() to capture events such as form submission or input field changes.
3. **Check Form Fields**:
   * Validate input values using logic that checks for empty fields, specific patterns, data types, etc.
4. **Display Feedback**:
   * Display error messages to users when validation fails.
5. **Prevent Default Behavior**:
   * Prevent form submission using event.preventDefault() if validation fails.

4. **Types of Validations**

**A. Required Fields**

* Ensure mandatory fields like name, email, and phone number are filled.

 function validateRequired(field) {

  if (field.value.trim() === "") {

    return false;

  }

  return true;

}

**B. Email Validation**

* Check if the email address is formatted correctly using regular expressions.

function validateEmail(email) {

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

  return emailPattern.test(email);

}

**C. Password Validation**

* Ensure passwords meet specific criteria (e.g., length, special characters, numbers).

function validatePassword(password) {

  const passwordPattern = /^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{8,}$/;  // At least 8 characters, 1 letter, 1 number

  return passwordPattern.test(password);

}

**4. Event-Based Validation**

JavaScript can validate form fields dynamically using events such as:

* **onblur**: Validates when the user leaves the field.
* **oninput**: Validates as the user types.
* **onsubmit**: Validates when the form is submitted.

    document.getElementById('email').addEventListener('blur', function() {

  if (!validateEmail(this.value)) {

    console.log('Invalid Email');

  }

  });

**5. Best Practices for Form Validation**

* **Validate Both Client-Side and Server-Side**: Never rely on client-side validation alone for security.
* **Provide Clear Error Messages**: Use concise, user-friendly language for error messages.
* **Real-Time Feedback**: Validate fields in real-time for better user experience.
* **Avoid Blocking Valid Input**: Be cautious with overly strict validation that might reject valid input.

**6. Form Validation Libraries**

There are JavaScript libraries available that can simplify form validation:

* **jQuery Validation Plugin**: An easy-to-use form validation plugin for jQuery.
* **Parsley.js**: A powerful library that provides automatic form validation with customizable options.

**Bom – Browser Object Model**

The Browser Object Model (BOM) is a set of objects provided by web browsers to interact with the browser itself, beyond just manipulating the content of a web page. It provides JavaScript access to various components of the browser environment, such as the browser window, history, location, and more.

**1. Window Object**

The window object represents the browser's window. All global JavaScript objects, functions, and variables automatically become members of the window object.

Alert , Confirm and Prompt

    // Display an alert

    window.alert("Hello!");

    // Confirm action

    if (window.confirm("Do you want to proceed?")) {

        console.log("You pressed OK!");

    } else {

        console.log("You pressed Cancel!");

    }

    // Prompt user for input

    let name = window.prompt("Enter your name:");

    console.log("Your name is: " + name);

window.open and window.close

<button onclick="newwebopen()">click me to open new window</button>;

<button onclick="newwebclose()">click me to close the current window</button>

function newwebopen(){

    window.open("https://www.google.com", "\_blank", "width=500, height=500");

}

function newwebclose(){

    window.close();

}

Inner height and inner width

var height= window.innerHeight;

  var width=window.innerWidth;

  console.log(height,width);

  if(window.innerWidth<550){

    document.body.style.backgroundColor="red";

  }

**Window Object**:

* Get the width and height of the browser window

console.log(window.innerWidth);

console.log(window.innerHeight);

window.open("https://example.com", "\_blank", "width=600,height=400");

**Navigator objects**

The Navigator object in JavaScript provides information about the browser's name, version, platform, and capabilities.

online and geolocation

    // Online or offline status

    if (navigator.onLine) {

        console.log("Browser is online.");

    } else {

        console.log("Browser is offline.");

    }

        // Check if geolocation is supported

    if (navigator.geolocation) {

        navigator.geolocation.getCurrentPosition(function(position) {

            console.log("Latitude: " + position.coords.latitude);

            console.log("Longitude: " + position.coords.longitude);

        });

    } else {

        console.log("Geolocation is not supported by this browser.");

    }

Appname, apppversion, platform, useragent , cookie enabled

// Properties

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

// Example output: "Browser Name: Netscape"

console.log("Browser Version:", navigator.appVersion);

// Example output: "Browser Version: 5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36"

console.log("Platform:", navigator.platform);

// Example output: "Platform: Win32"

console.log("User Agent:", navigator.userAgent);

// Example output: "User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.99 Safari/537.36"

console.log("Cookies Enabled:", navigator.cookieEnabled);

// Example output: "Cookies Enabled: true"

  // Check if cookies are enabled

    if (navigator.cookieEnabled) {

        console.log("Cookies are enabled!");

    } else {

        console.log("Cookies are disabled!");

    }

**Location objects**

It provides properties that allow you to access and manipulate different parts of the URL

// Get the full URL

console.log(window.location.href);

// Get the protocol (http/https)

console.log(window.location.protocol);

// Get the hostname

console.log(window.location.hostname);

// Get the path name

console.log(window.location.pathname);

// Redirect to another page

window.location.href = "https://www.example.com";

**Screen objects**

Screen object in JavaScript provides information about the user's screen or display, such as its width, height, color depth, and pixel density.

// Get screen width and height

console.log("Screen width: " + screen.width);

console.log("Screen height: " + screen.height);

// Get available width and height (excluding taskbar)

console.log("Available screen width: " + screen.availWidth);

console.log("Available screen height: " + screen.availHeight);

console.log("Color Depth:", screen.colorDepth);

// Example output: "Color Depth: 24" (Color depth of the screen in bits per pixel)

console.log("Pixel Depth:", screen.pixelDepth);

// Example output: "Pixel Depth: 24" (Pixel depth of the screen in bits per pixel)

console.log("Screen Width:", screen.width);

// Example output: "Screen Width: 1920" (Width of the screen in pixels)

console.log("Screen Height:", screen.height);

// Example output: "Screen Height: 1080" (Height of the screen in pixels)

console.log("Available Screen Width:", screen.availWidth);

// Example output: "Available Screen Width: 1920" (Available width of the screen in pixels)

console.log("Available Screen Height:", screen.availHeight);

// Example output: "Available Screen Height: 1040" (Available height of the screen in pixels)

console.log("Color Depth:", screen.colorDepth);

// Example output: "Color Depth: 24" (Color depth of the screen in bits per pixel)

console.log("Pixel Depth:", screen.pixelDepth);

// Example output: "Pixel Depth: 24" (Pixel depth of the screen in bits per pixel)

**History objects**

History object in JavaScript represents the user's navigation history for the current browser window. It allows you to navigate back and forward through the history stack

// Methods

history.back(); // Moves the browser back one page

history.forward(); // Moves the browser forward one page

history.go(-2); // Moves the browser back two pages

**Cookie objects**

Cookies are small pieces of data stored in the user's web browser. They are typically used by websites to remember users' preferences, authentication status, and other information related to their browsing session. Cookies can be set, retrieved, and deleted using JavaScript and are commonly used for tasks like personalization, tracking, and session management on the web.

**Timing functions**

Timing functions are crucial for managing when and how often certain blocks of code execute. There are several methods and concepts related to timing functions in JavaScript:

**1. setTimeout()**

The setTimeout() function is used to execute a specified block of code once after a specified time interval.

## setTimeout(function, delay, param1, param2, ...)

* function: A function or code snippet to execute.
* delay: Time in milliseconds (1000 ms = 1 second) before executing the function.
* param1, param2, ...: Optional additional parameters to pass to the function.

# setTimeout(function() {

# console.log("This message will appear after 2000 milliseconds.");

# }, 2000);

 <div id="notification">This is a notification!</div>

    <button onclick="showNotification()">Show Notification</button>

      function showNotification() {

            const notification = document.getElementById("notification");

            notification.style.display = "block";

            setTimeout(() => {

                notification.style.display = "none";

            }, 5000); // Hide after 5 seconds

        }

 const messages = ["Loading resources...", "Connecting to the server...", "Almost there...", "Done!"];

        let index = 0;

        function showNextMessage() {

            if (index < messages.length) {

  document.getElementById("loadingMessage").innerText = messages[index];

                index++;

                setTimeout(showNextMessage, 1000); // Call after 1 second

            }

        }

        showNextMessage();

**2. setInterval()**

The setInterval() function is used to repeatedly execute a specified block of code at a fixed interval.

# setInterval(function, delay, param1, param2, ...)

* function: A function or code snippet to execute.
* delay: Time in milliseconds between each execution of the function.
* param1, param2, ...: Optional additional parameters to pass to the function.

# var count = 0;

# var intervalId = setInterval(function() {

# count++;

# console.log("Counter: " + count);

# if (count === 5) {

# clearInterval(intervalId); // Stop the interval after 5 executions

# }

# }, 1000); // Execute every 1000 milliseconds (1 second)

        let timeLeft = 10;

        const timer = setInterval(() => {

            timeLeft--;

            document.getElementById("countdown").innerText = timeLeft;

            if (timeLeft <= 0) {

                clearInterval(timer); // Stop the interval

                document.getElementById("countdown").innerText = "Time's up!";

            }

        }, 1000); // Decrease every second

**clearTimeout() and clearInterval()**

* clearTimeout(timeoutId): Clears the timeout specified by timeoutId, typically returned by setTimeout(), canceling the execution of the function.
* clearInterval(intervalId): Clears the interval specified by intervalId, typically returned by setInterval(), stopping further executions.

**Session storage and local storage**

**Session storage** is a part of the Web Storage API in web browsers that provides a way to store key-value pairs locally on the client-side.

* **sessionStorage** maintains a separate storage area for each given origin that's available for the duration of the page session (as long as the browser is open, including page reloads and restores).
* Data stored in **sessionStorage** is cleared when the page session ends.
* Data is only accessible within the window/tab that set it.

// Storing data in sessionStorage

sessionStorage.setItem('username', 'John');

// Retrieving data from sessionStorage

let username = sessionStorage.getItem('username');

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

// Removing data from sessionStorage

sessionStorage.removeItem('username');

**localStorage**:

**localStorage** is a feature of web browsers that allows web applications to store key-value pairs locally on the client-side. It provides a persistent storage mechanism, meaning that the data stored in **localStorage** remains available even after the browser is closed and reopened, and across browser sessions.

* **localStorage** does almost the same thing as **sessionStorage**, but it persists even when the browser is closed and reopened.
* Data stored in **localStorage** has no expiration time.
* Data is accessible across windows and tabs within the same origin.

// Storing data in localStorage

localStorage.setItem('email', 'example@example.com');

// Retrieving data from localStorage

let email = localStorage.getItem('email');

console.log(email); // Output: example@example.com

// Removing data from localStorage

localStorage.removeItem('email');

**how to display some data from one page to another page using local storage**

local storage limited to handle only string key/value pairs you can do like below using JSON.stringify and while getting value JSON.parse

var testObject ={name:"test", time:"Date 2017-02-03T08:38:04.449Z"};

Put the object into storage:

**localStorage.setItem('testObject', JSON.stringify(testObject));**

Retrieve the object from storage:

**var retrievedObject = localStorage.getItem('testObject');**

console.log('retrievedObject: ', JSON.parse(retrievedObject));

**Add to cart functionality**

//first file

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

    <style>

      .container {

        display: grid;

        grid-template-columns: auto auto auto;

        gap: 20px;

      }

      .container > div {

        padding: 20px;

        border: 1px solid red;

      }

      .container > div > div,

      h1 {

        padding: 10px;

        border: 2px solid blue;

      }

    </style>

  </head>

  <body>

    <h1>

      <button onclick="cart()">cart</button>

    </h1>

    <div id="row" class="container"></div>

    <script>

      async function apicall() {

        var newarr = [];

        var result = await fetch("https://fakestoreapi.com/products");

        var apidata = await result.json();

        console.log(apidata);

        var iterated = apidata.map((val) => {

          // console.log(val);

          var row = document.getElementById("row");

          var main = document.createElement("div");

          var child1 = document.createElement("h1");

          var child2 = document.createElement("div");

          var child3 = document.createElement("div");

          var child4 = document.createElement("div");

          child1.innerHTML = val.id + " <br>";

          child2.innerHTML = val.title + " <br>";

          child3.innerHTML = val.description + " <br>";

          child4.innerHTML = val.price + " <br>";

          var btn = document.createElement("button");

          btn.innerHTML = "click";

          btn.addEventListener("click", function () {

            newarr.push(val);

            sessionStorage.setItem("arr", JSON.stringify(newarr));

          });

          main.append(child1, child2, child3, child4, btn);

          row.appendChild(main);

        });

      }

      apicall();

      function cart() {

        window.open("sub.html", "\_self");

      }

    </script>

  </body>

</html>

//second file

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <div id="row"></div>

    <script>

      var newarrdata = JSON.parse(sessionStorage.getItem("arr"));

      console.log(newarrdata);

      var iterated = newarrdata.map((val) => {

        // console.log(val);

        var row = document.getElementById("row");

        var main = document.createElement("div");

        var child1 = document.createElement("h1");

        var child2 = document.createElement("div");

        var child3 = document.createElement("div");

        var child4 = document.createElement("div");

        child1.innerHTML = val.id + " <br>";

        child2.innerHTML = val.title + " <br>";

        child3.innerHTML = val.description + " <br>";

        child4.innerHTML = val.price + " <br>";

        var btn = document.createElement("button");

        btn.innerHTML = "click";

        btn.addEventListener("click", function () {

          main.style.display = "none";

        });

        main.append(child1, child2, child3, child4, btn);

        row.appendChild(main);

      });

    </script>

  </body>

</html>

**Regular Expressions (Regex)**

A regular expression is a sequence of characters that helps to create a search pattern, often used for string matching and manipulation.

**Ways to Create Regular Expressions**

1. **Literal Notation**: Uses slashes to define the pattern.
   * **Syntax**: /pattern/flags
   * **Example**:
     + /hello/ matches the string "hello".
     + /^[0-9]{10}$/ matches a 10-digit number (starts with a digit, ends with a digit).
     + /[0|1]/ matches either "0" or "1".
2. **Bracket Expressions**:
   * [ ] Matches any character inside the brackets.
     + Example: [a-z] matches any lowercase letter between 'a' and 'z'.
   * [^ ] Matches any character *not* in the brackets.
     + Example: [^a-z] matches any character except lowercase letters.
   * [0-9] Matches any digit between '0' and '9'.
   * [A-Z] Matches any uppercase letter between 'A' and 'Z'.

**Common Escape Sequences**

* \d Matches any digit (equivalent to [0-9]).
* \D Matches any non-digit character (equivalent to [^0-9]).
* \w Matches alphanumeric characters and underscores (equivalent to [A-Za-z0-9\_]).
* \W Matches any non-alphanumeric character.
* \. Matches a literal period (dot).

**Methods for Using Regular Expressions**

1. **test()**: Checks if the pattern exists in a string and returns true or false.

# const regex = /\d+/;

# console.log(regex.test("123")); // true

# console.log(regex.test("abc")); // false

**Quantifiers**

Quantifiers are used to specify how many times a character or group should be matched:

* \* Matches zero or more times.
  + **Example**: /a\*/ matches "a", "aa", or an empty string.
* + Matches one or more times.
  + **Example**: /a+/ matches "a", "aa", but not an empty string.
* ? Matches zero or one time.
  + **Example**: /a?/ matches "a" or an empty string.
* {n}: Matches exactly **n** occurrences of the preceding element.
  + **Example**: /a{3}/ matches exactly three "a" characters in a row ("aaa").
    - "aaa" matches, "aa" does not match.
* {n,}: Matches **n or more** occurrences of the preceding element.
  + **Example**: /a{3,}/ matches three or more "a" characters.
    - "aaa", "aaaa", and "aaaaa" match, but "aa" does not.
* {n,m}: Matches **between n and m** occurrences of the preceding element.
  + **Example**: /a{3,5}/ matches between 3 to 5 "a" characters.
    - "aaa", "aaaa", and "aaaaa" match, but "aa" and "aaaaaa" do not.

**Examples of Regex**

1. **Phone Number Validation**:
   * Regex: /^[0-9]{10}$/
   * Explanation: Matches exactly 10 digits from 0-9, used for validating phone numbers.
2. **Binary Values (0 or 1)**:
   * Regex: /[0|1]/
   * Explanation: Matches either 0 or 1.
3. **Simple Email Validation**:
   * Regex: /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
   * Explanation: Matches a basic email format.

**Examples**

**1. Validate an Email Address**

* **Pattern**: /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
* **Explanation**:
  + ^[a-zA-Z0-9.\_%+-]+ - Matches the username (alphanumeric characters and special symbols like .\_%+-).
  + @ - Requires the "@" symbol.
  + [a-zA-Z0-9.-]+ - Matches the reain name.
  + \.[a-zA-Z]{2,} - Ensures a valid top-level domain (TLD), like .com or .net.

**2. Validate a Phone Number (10 digits)**

* **Pattern**: /^[0-9]{10}$/
* **Explanation**:
  + ^[0-9]{10}$ - Ensures exactly 10 digits between 0 and 9.

**3. Validate a URL**

* **Pattern**: /^(https?:\/\/)?(www\.)?[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}(\/\S\*)?$/
* **Explanation**:
  + https?:\/\/ - Matches "http" or "https".
  + (www\.)? - Optional "www.".
  + [a-zA-Z0-9.-]+ - Matches the domain name.
  + \.[a-zA-Z]{2,} - Ensures a valid TLD.
  + (\/\S\*)? - Optionally allows a path after the domain.

**4. Validate a Credit Card Number**

* **Pattern**: /^(?:\d{4}-?){3}\d{4}$/
* **Explanation**:
  + (?:\d{4}-?){3} - Matches three groups of 4 digits, optionally separated by hyphens.
  + \d{4} - Matches the final group of 4 digits.

**5. Validate a ZIP Code (US, 5 digits)**

* **Pattern**: /^\d{5}(-\d{4})?$/
* **Explanation**:
  + ^\d{5} - Matches exactly 5 digits.
  + (-\d{4})? - Optionally matches a hyphen followed by 4 digits for ZIP+4 codes.

**6. Match a Date (MM/DD/YYYY)**

* **Pattern**: /^(0[1-9]|1[0-2])\/(0[1-9]|[12][0-9]|3[01])\/\d{4}$/
* **Explanation**:
  + (0[1-9]|1[0-2]) - Matches months from 01 to 12.
  + (0[1-9]|[12][0-9]|3[01]) - Matches days from 01 to 31.
  + \d{4} - Matches exactly 4 digits for the year.

**7. Match a Time (24-hour format)**

* **Pattern**: /^([01][0-9]|2[0-3]):[0-5][0-9]$/
* **Explanation**:
  + ([01][0-9]|2[0-3]) - Matches hours from 00 to 23.
  + :[0-5][0-9] - Matches minutes from 00 to 59.

**8. Match Only Numbers**

* **Pattern**: /^\d+$/
* **Explanation**:
  + ^\d+$ - Matches any sequence of one or more digits (whole numbers only).

**9. Match Only Alphanumeric Characters**

* **Pattern**: /^[a-zA-Z0-9]+$/
* **Explanation**:
  + ^[a-zA-Z0-9]+$ - Matches any sequence of alphanumeric characters.

**10. Match Hexadecimal Colors**

* **Pattern**: /^#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3})$/
* **Explanation**:
  + #? - The # symbol is optional.
  + ([a-fA-F0-9]{6}|[a-fA-F0-9]{3}) - Matches either a 6-character or 3-character hex color.

**11. Strip Whitespace from the Beginning and End of a String**

* **Pattern**: /^\s+|\s+$/g
* **Explanation**:
  + ^\s+ - Matches one or more whitespace characters at the start.
  + \s+$ - Matches one or more whitespace characters at the end.
  + g flag - Global search for all matches.

**12. Match a Word Boundary**

* **Pattern**: /\bword\b/
* **Explanation**:
  + \b - Ensures the pattern matches at word boundaries.
  + word - The specific word to match.

**13. Match a Floating-Point Number**

* **Pattern**: /^[+-]?([0-9]\*[.])?[0-9]+$/
* **Explanation**:
  + [+-]? - Allows an optional "+" or "-" sign.
  + ([0-9]\*[.])? - Optionally matches digits before and after a decimal point.
  + [0-9]+ - Requires at least one digit.

**14. Validate a Strong Password**

* **Pattern**: /^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/
* **Explanation**:
  + (?=.\*[a-z]) - Requires at least one lowercase letter.
  + (?=.\*[A-Z]) - Requires at least one uppercase letter.
  + (?=.\*\d) - Requires at least one digit.
  + (?=.\*[@$!%\*?&]) - Requires at least one special character.
  + {8,} - Must be at least 8 characters long.

**15. Match a Repeated Character**

* **Pattern**: /(\w)\1+/
* **Explanation**:
  + (\w) - Captures any alphanumeric character.
  + \1+ - Matches one or more occurrences of the same captured character.

**16. Match an IP Address (IPv4)**

* **Pattern**: /^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/
* **Explanation**:
  + Each part of the IP is matched using (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?) which ensures a valid number between 0 and 255.

**17. Match HTML Tags**

* **Pattern**: /^<\/?[\w\s"'-=]+>$/
* **Explanation**:
  + <\/? - Matches an opening or closing tag.
  + [\w\s"'-=]+ - Matches tag content (attributes, values).
  + > - Matches the closing bracket.

**18. Match Leading Zeroes**

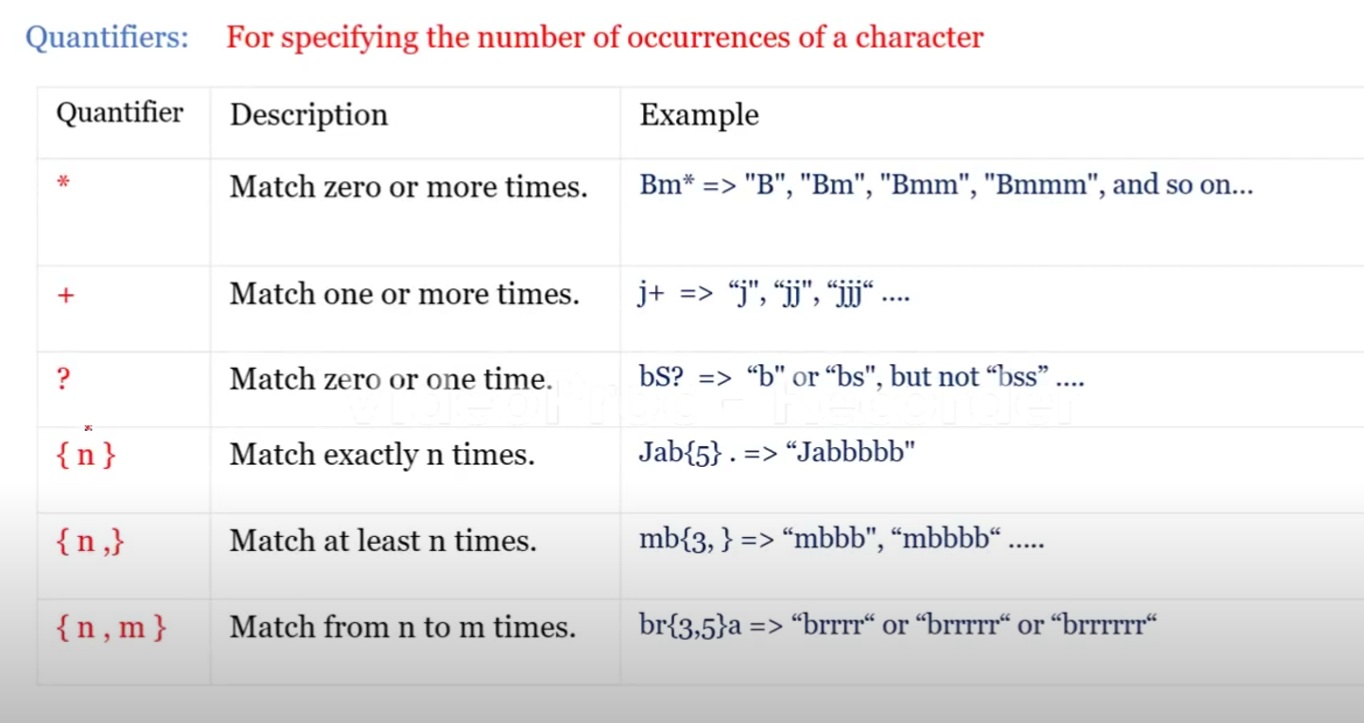
* **Pattern**: /^0+(?=\d)/
* **Explanation**:
  + ^0+ - Matches one or more leading zeroes.
  + (?=\d) - Ensures a digit follows.

**19. Validate a Username**

* **Pattern**: /^[a-zA-Z0-9\_]{3,16}$/
* **Explanation**:
  + ^[a-zA-Z0-9\_]{3,16}$ - Matches a username that is 3-16 characters long, only allowing alphanumeric characters and underscores.

**20. Match a Word with Only Letters**

* **Pattern**: /^[A-Za-z]+$/
* **Explanation**:
  + ^[A-Za-z]+$ - Matches a word containing only letters (upper or lowercase).



**Regexr.com**

**Synchronous and asynchronous**

**Synchronous:** In synchronous operations, code is executed sequentially, one line at a time. Each line must wait for the previous one to finish executing before it can start. This can sometimes lead to blocking behavior, where one task prevents another from executing until it's complete.

    console.log("Start");

    console.log("Middle");

    console.log("End");

    // start

    // middle

    // end

In this synchronous code, "Start" will be logged first, followed by "Middle", and then "End".

**Asynchronous:** Asynchronous operations allow code to execute independently from the main program flow. This means that while one operation is being processed, the program can continue to execute other tasks. Asynchronous operations are typically used for tasks that may take some time to complete, such as fetching data from a server or reading a file. In JavaScript, common asynchronous operations are handled using callbacks, promises, or async/await syntax.

    console.log("Start");

    setTimeout(()=>{

       console.log("middle")

    }),2000;

    console.log("End");

    // start

    // end

    // middle

In this example, "Start" is logged first, then after a delay of 2000 milliseconds, "End" is logged, followed by " Middle ".

**How js works**

1. **Call Stack:** The call stack is a data structure that keeps track of function calls in the code  runs in a last-in, first-out way. Whenever a function is called, it's added to the top of the call stack. When a function completes, it's removed from the stack
2. **Web APIs**

The Web APIs are not a part of the JavaScript engine, but they are part of the runtime environment provided by the browser. There are a large number of APIs available in modern browsers that allow us to a wide variety of things.

1. **Callback que:**

"callback queue," also known as the "task queue," is a part of JavaScript's event loop mechanism that manages asynchronous tasks or callbacks for execution. When an asynchronous operation such as a timer (setTimeout or setInterval) or an event (such as user interaction or network response) completes, its associated callback function is placed in the callback queue.

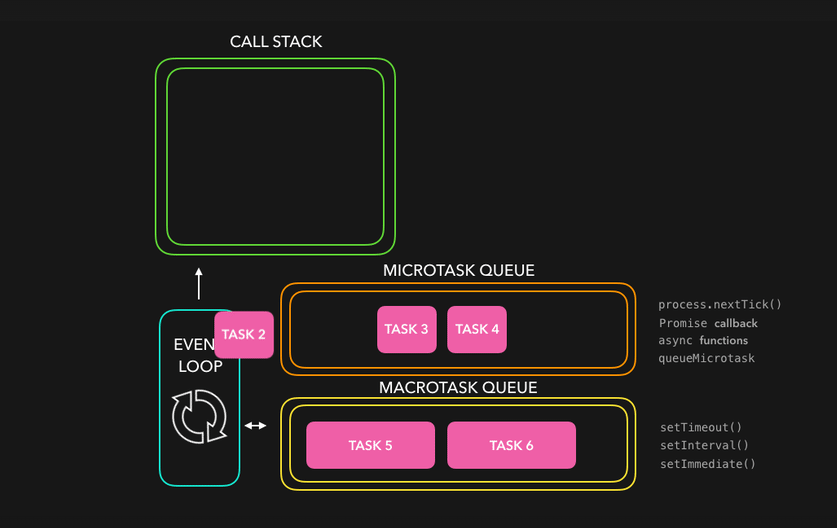
Such as fetch,settimeout,console , dom,localstorage and location

The callback queue stores the callback functions sent from the Web APIs in the order in which they were added. This queue is a data structure that runs [first in, first out](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)).

Callback functions will sit in the queue until the call stack is empty, they are then moved into the stack by the event loop.

1. **Micro task que**

Microtask queue in the event loop handles small units of asynchronous work with higher priority, ensuring they are executed before regular tasks. It's commonly used for tasks associated with promises



1. **Event Loop**: The event loop is a mechanism in JavaScript that continuously checks the call stack and the callback queue.

if the call stack is empty it will move the callback function at the front of the queue to the call stack, scheduling it for execution .

It ensures that the execution of code is done in the right order, especially when dealing with asynchronous operations.

1. **CallStack Overflow:** If the call stack grows too large, typically due to infinite recursion or excessive nested function calls, it can exceed the available memory allocated for the stack. This results in a "stack overflow" error and crashes the program.

1. **Basic Synchronous Code**

console.log("Task 1: Start");

console.log("Task 2: Middle");

console.log("Task 3: End");

// start

// middle

// end

**2 Asynchronous with setTimeout (Event Loop in Action)**

console.log("Task 1: Start");

setTimeout(() => {

    console.log("Task 2: Asynchronous Task (After 2 seconds)");

}, 2000);

console.log("Task 3: End");

start

// end

// asynchronous task

**3. Asynchronous Code with a Promise**

console.log("Task 1: Start");

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

    setTimeout(() => {

        resolve("Task 2: Promise Resolved (After 1 second)");

    }, 1000);

});

promise.then((message) => {

    console.log(message);

});

console.log("Task 3: End");

// Task 1: Start

// Task 3: End

// Task 2: Promise Resolved (After 1 second)

**4. Multiple Asynchronous Tasks**

  console.log("Task 1: Start");

    setTimeout(() => {

        console.log("Task 2: Asynchronous Task 1 (After 2 seconds)");

    }, 2000);

    setTimeout(() => {

        console.log("Task 3: Asynchronous Task 2 (After 0 seconds)");

    }, 0);

    console.log("Task 4: End");

    // Task 1: Start

    // Task 4: End

    // Task 3: Asynchronous Task 2 (After 0 seconds)

    // Task 2: Asynchronous Task 1 (After 2 seconds)

**5. setTimeout and Promise Together**

console.log("Task 1: Start");

setTimeout(() => {

    console.log("Task 2: Asynchronous Task (setTimeout)");

}, 0);

Promise.resolve().then(() => {

    console.log("Task 3: Promise Resolved");

});

console.log("Task 4: End");

// Task 1: Start

// Task 4: End

// Task 3: Promise Resolved

// Task 2: Asynchronous Task (setTimeout)

**6. Using async/await**

async/await is a syntactic sugar for working with promises. It behaves synchronously within an async function until an await keyword is encountered, at which point it returns to the event loop to handle other tasks.

console.log(“Task 1: Start");

    async function asyncTask() {

        console.log("Task 2: Inside asyncTask");

        await new Promise(resolve => setTimeout(resolve, 1000)); // Wait for 1 second

        console.log("Task 3: After 1 second wait in asyncTask");

    }

    asyncTask();

    console.log("Task 4: End");

//    Task 1: Start

//     Task 2: Inside asyncTask

//     Task 4: End

//     Task 3: After 1 second wait in asyncTask

**7) setTimeout with Different Delays**

   console.log("Task 1: Start");

    setTimeout(() => {

        console.log("Task 2: 2 seconds delay");

    }, 2000);

    setTimeout(() => {

        console.log("Task 3: 1 second delay");

    }, 1000);

    setTimeout(() => {

        console.log("Task 4: 0 seconds delay");

    }, 0);

    console.log("Task 5: End");

    // Task 1: Start

    // Task 5: End

    // Task 4: 0 seconds delay

    // Task 3: 1 second delay

    // Task 2: 2 seconds delay

**Higher order function**

A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result.

      function higherOrderFunction(callback) {

        // Perform some operation

        callback();

      }

      function callbackFunction() {

        console.log("Callback function called");

      }

      higherOrderFunction(callbackFunction);

**Callback hell**

Callback hell, also known as "Pyramid of Doom," is a term used in JavaScript programming to describe a situation where multiple nested callbacks make the code difficult to read, understand, and maintain. This usually happens when dealing with asynchronous operations, such as making API requests or reading files.

step1(function() {

    step2(function() {

        step3(function() {

            console.log("All steps completed");

        });

    });

});

      function first(callback) {

        console.log("first");

        callback();

      }

      function second(callback) {

        console.log("second");

        callback();

      }

      function third(callback) {

        console.log("third");

        callback();

      }

      function fourth(callback) {

        console.log("fourth");

      }

      first(() => {

        second(() => {

          third(() => {

            fourth();

          });

        });

      });

      function add(val, callback) {

        callback(val + 10);

      }

      function sub(val, callback) {

        callback(val - 5);

      }

      function mul(val, callback) {

        callback(val \* 2);

      }

      function div(val, callback) {

        callback(val / 5);

      }

      add(10, (addres) => {

        sub(addres, (subres) => {

          mul(subres, (mulres) => {

            div(mulres, (finalres) => {

               console.log(finalres);

            });

          });

        });

      });

function wakeUp(callback) {

    setTimeout(() => {

        console.log("1. Woke up");

        callback();

    }, 1000);

}

function eatBreakfast(callback) {

    setTimeout(() => {

        console.log("2. Ate breakfast");

        callback();

    }, 1000);

}

function study(callback) {

    setTimeout(() => {

        console.log("3. Studied");

        callback();

    }, 1000);

}

function goToSleep(callback) {

    setTimeout(() => {

        console.log("4. Went to sleep");

        callback();

    }, 1000);

}

// The callback hell part starts here

wakeUp(() => {

    eatBreakfast(() => {

        study(() => {

            goToSleep(() => {

                console.log("Finished all tasks!");

            });

        });

    });

});

To mitigate callback hell, several approaches have been developed:

* 1. Named functions
  2. Promises
  3. Async/await

**Promises**

A Promise in JavaScript represents the eventual completion (or failure) of an asynchronous operation and its resulting value. When we make API requests, they don’t return results immediately. Promises are useful to handle such operations, ensuring that we can handle the data once it’s available (or handle any errors that occur).

**Key States of a Promise:**

* Pending: The promise is still in process (operation is ongoing).
* Fulfilled: The promise is successfully completed (operation succeeded).
* Rejected: The promise failed (operation failed).

Promises in JavaScript provide a cleaner and more structured way to handle asynchronous operations compared to traditional callbacks.

1. **Creating a Promise(producing)** : You create a new Promise object using the **Promise** constructor. This constructor takes a function as an argument, which in turn takes two parameters: **resolve** and **reject**. Inside this function, you perform your asynchronous operation, and when it's completed, you call **resolve** with the result or **reject** with an error if it fails.

      //promises creation

      var promises=new Promise(function (resolve,reject){

      var a=100;

      if(a==10){

      resolve("a is 10")

      }

      else{

      reject("a is not 10")

      }

      });

1. **Consuming a Promise**: You consume a promise using the **then** method, which takes two optional parameters: a callback function to handle the resolved value, and a callback function to handle any errors.

      //print the response

      promises.then((val)=>{

      console.log(val)

      }).catch((err)=>{

      console.log(err)

      })

Promises also forming chain method which inturns make code readability difficult in order to avoid this

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

  const success = true; // Change to false to see reject in action

  if (success) {

    resolve("Promise resolved successfully!");

  } else {

    reject("Promise rejected!");

  }

});

promise

  .then((message) => {

    console.log(message);  // "Promise resolved successfully!"

  })

  .catch((error) => {

    console.error(error);  // "Promise rejected!" if success is false

  });

Promise with setTimeout (Asynchronous Operation)

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

        setTimeout(() => {

          resolve("Data fetched after 2 seconds!");

        }, 2000);

      });

      asyncOperation

        .then((data) => {

          console.log(data); // "Data fetched after 2 seconds!"

        })

        .catch((error) => {

          console.error(error);

        });

Promise with addres

 function sum(val){

            return new Promise((resolve)=>{

                resolve(val+10)

            })

           }

           function sub(val){

            return new Promise((resolve)=>{

                resolve(val-2)

            })

           }

           function mul(val){

            return new Promise((resolve)=>{

                resolve(val\*2)

            })

           }

           function div(val){

            return new Promise((resolve)=>{

                resolve(val/5)

            })

           }

           sum(10)

           .then((addres)=>sub(addres))

           .then((subres)=>mul(subres))

           .then((mulres)=>div(mulres))

           .then((divres)=>console.log(divres))

Chaining Promises

      var one = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("one");

        }, 100);

      });

      var two = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("two");

        }, 50000);

      });

      var three = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("three");

        }, 100);

      });

       one.then((resolve)=>{console.log(resolve)})

    two.then((resolve)=>{console.log(resolve)})

       three.then((resolve)=>{console.log(resolve)})

//asynchronous in synchronous way

      one

        .then((val) => {

          console.log(val);

          return two;

        })

        .then((val) => {

          console.log(val);

          return three;

        })

        .then((val) => {

          console.log(val);

        });

Promise all

      var one = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("one fun");

        }, 10000);

      });

      var two = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("two fun");

        }, 500);

      });

      var three = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("three fun");

        }, 100);

      });

      Promise.all([three, two, one])

        .then((results) => console.log(results))

        .catch((error) => {

          console.log("error");

        });

Promise race

      var one = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("one fun");

        }, 10000);

      });

      var two = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("two fun");

        }, 500);

      });

      var three = new Promise((resolve, reject) => {

        setTimeout(() => {

          resolve("three fun");

        }, 100);

      });

      Promise.race([three, two, one])

        .then((results) => console.log(results))

        .catch((error) => {

          console.log("error");

        });

**Async/Await**  
Async/await is a modern feature in JavaScript that simplifies working with asynchronous code, especially when dealing with Promises. It allows you to write asynchronous code in a synchronous-like manner, making it easier to read, write, and maintain.

1. **Async Functions**: An async function is a function that operates asynchronously via the event loop. You declare an async function by prefixing the function declaration with the **async** keyword.

      async function myAsyncFunction() {

        // Asynchronous code here

      }

1. **Await Keyword:** The await keyword is used inside an async function to pause the execution of the function until a Promise is settled (resolved or rejected). It allows you to write code that looks synchronous but behaves asynchronously.

      async function myAsyncFunction() {

        const result = await somePromise;

        // Code here executes after somePromise is resolved

      }

Example

//promise is created

      function apromise() {

        return new Promise(function (res, rej) {

          var a = 20;

          if (a % 2 == 0) {

            res("num is even");

          } else {

            rej("num is odd");

          }

        });

      }

//resolving the promise value using async/await

      async function asyncfun() {

        var v = await apromise();

        console.log(v);

      }

      asyncfun();

//callback hell

        async function executor(){

          var addres=await add(10);

          var subres=await sub(addres);

          var mulres=await mul(subres);

          var divres=await div(mulres);

          console.log(divres);

        }

        executor()

//asynchronous

      async function executor() {

        let result1 = await promise1;

        console.log(result1);

        let result2 = await promise2;

        console.log(result2);

        let result3 = await promise3;

        console.log(result3);

        let result4 = await promise4;

        console.log(result4);

      }

      executor();

Fetch

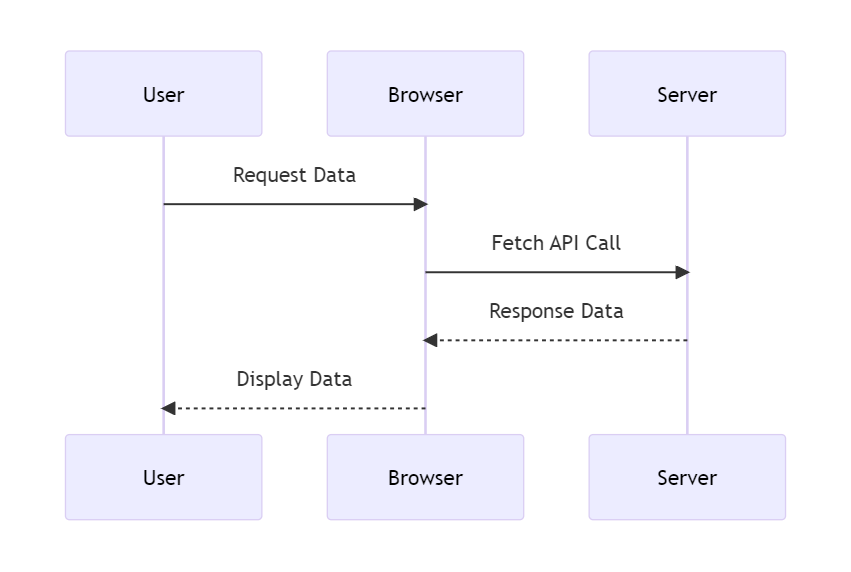
The fetch() API is a modern way to make network requests and handle responses. It is widely used to interact with web APIs, allowing to request and send data to servers. It returns a Promise that resolves to the Response object representing the response to the request.

# fetch(url, options)

\* url: The endpoint from which the resource is to be fetched.

\* options (optional): An object containing additional settings such as HTTP method, headers, body, etc.

Example using **fetch()** API with Promises:



When Data fetched successfuly

  fetch("https://fakestoreapi.com/products/1")

  .then(res=>res.json())

  .then(data=>console.log(data))

  .catch(err=>console.log("there was a problem"))

When there is a error

  fetch("https://fakestoreai.com/products/1")

  .then(res=>res.json())

  .then(data=>console.log(data))

  .catch(err=>console.log("there was a problem"))

**Explanation:**

1. fetch() returns a **promise** that resolves with the response object once the data is available.
2. The .then() method is used to handle the fulfilled promise.
3. If something goes wrong, we can use .catch() to handle the rejection.

**. Async/Await**

The async and await keywords simplify working with promises, allowing us to write asynchronous code that looks synchronous.

* **async**: Marks a function as asynchronous. This means it always returns a promise.
* **await**: Pauses the execution of the function until the promise is resolved or rejected.

Example using fetch() API with async/await:

        async function executor(){

            var res= await fetch("https://fakestoreapi.com/products/1");

            var data = await res.json();

            console.log(data);

        }

        executor()

**async with try and catch**

      async function executor() {

        try {

          var res = await fetch("https://fakestoreapi.com/products/1");

          var data = await res.json();

          console.log(data);

        } catch (err) {

          console.log("data not loading");

        }

      }

      executor();

**Explanation:**

1. The executor function is marked as **async**, which means it can use **await**.
2. Inside the function, await fetch() pauses the execution until the fetch() promise is resolved.
3. Once resolved, the response is checked and the JSON data is awaited.
4. If there's an error (e.g., network failure), it is caught by the try...catch block.

**Differences Between Promises and Async/Await:**

1. **Readability**: Async/await makes asynchronous code look synchronous, which improves readability, especially in complex scenarios with many chained .then() blocks.
2. **Error Handling**: Async/await uses try...catch, which is often more intuitive for handling errors compared to .catch() in promises.

JSON- javascript object notation



**Ajax**

Ajax, which stands for "Asynchronous JavaScript and XML," is a web development technique used to create interactive web applications. It allows for the asynchronous exchange of data between the browser and the server

**JSON**

JSON, or JavaScript Object Notation, is a **lightweight data interchange format** widely used in web development and other software applications. Its **syntax is derived from JavaScript object notation, but it is language independent** making it easy to read and write for humans. JSON is commonly used for transmitting data between a server and a web application due to its simplicity, universality, and support for complex data structures like nested objects and arrays. It is supported by virtually all modern programming languages and is **commonly used in web APIs** for its lightweight nature and ease of parsing. JSON's security is generally robust, but precautions should be taken to prevent vulnerabilities like JSON injection attacks. Additionally, JSON Schema provides a way to define and validate the structure and constraints of JSON documents, enhancing data integrity and interoperability. Overall, JSON's simplicity, readability, and wide support make it a popular choice for **data interchange in various software applications**.

**Why json**

B2b

data

Make my trip(nodejs)

Irctc(java)

json

**JSON.stringify()**:

**JSON.stringify()** is a built-in JavaScript method used to convert a JavaScript object into a JSON string.

**JSON.parse()**:

**JSON.parse()** is a built-in JavaScript method used to parse a JSON-formatted string and convert it into a JavaScript object.

res.ok

res.status

**Http status codes**

HTTP status codes are standard response codes returned by web servers to indicate the outcome of a client's request.

Important HTTP status codes along with their meanings:

1. **200 OK**: This status code indicates that the request was successful, and the server has returned the requested resource.
2. **201 Created**: Indicates that the request was successful, and a new resource has been created as a result.
3. **204 No Content**: The server successfully processed the request, but there is no content to return.
4. **400 Bad Request**: This status code is returned when the server cannot process the request due to a client error, such as malformed syntax or invalid parameters.
5. **401 Unauthorized**: Indicates that the client needs to authenticate itself to access the requested resource.
6. **403 Forbidden**: The server understood the request, but the client is not allowed to access the requested resource.

**Http methods**

HTTP methods, also known as HTTP request methods, are actions that indicate the desired operation to be performed on a resource identified by a URI (Uniform Resource Identifier).

1. **GET**: The GET method requests a representation of the specified resource. It is primarily used for retrieving data from the server. GET requests should only retrieve data and should not have any other effect on the server.
2. **POST**: The POST method submits data to be processed to a specified resource. It is commonly used for creating new resources on the server or submitting form data.
3. **PUT**: The PUT method replaces all current representations of the target resource with the request payload. It is typically used to update or create a resource with a specific identifier.
4. **PATCH**: The PATCH method is used to apply partial modifications to a resource. It is similar to the PUT method but only updates the parts of the resource specified in the request.
5. **DELETE**: The DELETE method requests the removal of the specified resource. It is used to delete resources identified by the URI from the server.

**How to call api using fetch by then method**

      fetch("https://fakestoreapi.com/products")

        .then((val) => {

          return val.json();

        })

        .then((val) => {

          console.log(val);

        });

1. **fetch("https://fakestoreapi.com/products")**: This line initiates a request to the specified URL, which returns a Promise representing the response to that request.
2. **.then((response) => { return response.json(); })**: Once the request is complete, this line chains a **.then()** method to the Promise returned by **fetch()**. Inside this **.then()** method, it takes the response object, and the **json()** method is called on it. This method returns a Promise that resolves to the JSON representation of the response body.
3. **.then((data) => { console.log(data); })**: After parsing the JSON response, this line chains another **.then()** method to the Promise returned by **response.json()**. Inside this **.then()** method, it receives the parsed JSON data as a JavaScript object. In this example, it logs the retrieved data to the console using **console.log()**

Some apis

**Food api**

[**https://api.edamam.com/search?q=biriyani&app\_id=a52b4d43&app\_key=e0e5c667605f5e91d8275c973531b80a**](https://api.edamam.com/search?q=biriyani&app_id=a52b4d43&app_key=e0e5c667605f5e91d8275c973531b80a)

**Weather api -** [https://api.openweathermap.org/data/2.5/weather?q=**hyderabad**&units=metric&appid=466ddaa21a8de191e9f608bd11a56acb](https://api.openweathermap.org/data/2.5/weather?q=hyderabad&units=metric&appid=466ddaa21a8de191e9f608bd11a56acb)

**Quotes api**

<https://api.quotable.io/random>

**Random joke**

[**https://v2.jokeapi.dev/joke/Programming?blacklistFlags=nsfw,religious,political,racist,explicit&type=single**](https://v2.jokeapi.dev/joke/Programming?blacklistFlags=nsfw,religious,political,racist,explicit&type=single)

**Movies info**

[https://www.omdbapi.com/?t=**titanic**&apikey=76d079f0](https://www.omdbapi.com/?t=titanic&apikey=76d079f0)

**Random joke**

async function getRandomJoke() {

  const rawRes = await fetch('https://icanhazdadjoke.com/', {

    headers: {

      Accept: 'application/json'

    }

  });

  const res = await rawRes.json();

  console.log(res.joke);

}

getRandomJoke()

<https://github.com/saiteja-yernagula/javascript-mini-projects>

**JSON Server**

**How to create a local json server**

1. Download and Install Node js
2. Check whether it is installed or not by using below commands

**node –version**

**npm –version**

**Open powershell run as administrator and run the commands**

**Get-ExecutionPolicy**

**Set-ExecutionPolicy RemoteSigned**

**Y – for yes**

**Get-ExecutionPolicy**

1. Select the folder in your local , open in vs code terminal and use below command

**npm init –y**

You will find json packages then you can install any libraries

1. Install json server for api creation in local server

**npm install -g json-server@0**

1. After installation watch the server using below command if it throws error follow the next step for script enabalation.

**json-server --watch db.json –port num**

1. Adjust the version by reinstalling using below command

**npm install -g json-server@0**

**json-server --watch db.json –port 6000**

**How to use post or patch in fetch**

      fetch("url", {

        method: "POST",

        body: JSON.stringify({

          name: "John",

          age: 20,

        }), //need to write headers

        headers: {

          "Content-type": "application/json; charset=UTF-8",

        },

      });

      //put changes the entire value

      //patch changes the one value which is updated

**Stepwise all methods**

**Step 1 – create db.json file**

**Step 2 – add the data**

{

    "data":[

{

    "name":"teja",

    "id":"4"

}

,

{

    "name":"sai",

    "id":"2"

},

{

    "name":"hemanth",

    "id":"3"

},

{

    "name":"chaitanya",

    "id":"1"

}

]}

**Step 3 – json-server --watch db.json –port 6000**

**Step 4 - open js file paste the below program in it**

script>

        //  get method - used to get data from the server

        fetch("http://localhost:3000/data")

        .then(response=>{

            if(response.ok){

                return response.json()

            }else{

                return "error code" + response.statusText

            }

        }).then(data=>{

            console.log(data);

        })

**Json server**

HTTP methods (GET, POST, PUT, PATCH, DELETE) using fetch() and interacting with a **JSON Server**. JSON Server is a simple API tool that allows you to simulate a REST API with basic HTTP methods

Assume your JSON server is running at http://localhost:3000 with the following db.json:

{

    "data":[{

    "name":"teja",

    "id":"4"

}

,

{

    "name":"sai",

    "id":"2"

},

{

    "name":"hemanth",

    "id":"3"

},

{

    "name":"chaitanya",

    "id":"1"

}

]}

1. **GET Request**: Fetching Data from the Server

fetch('http://localhost:3000/posts')

  .then(response => response.json())  // Parse the JSON response

  .then(data => console.log(data))    // Handle the received data

  .catch(error => console.error('Error:', error));  // Handle errors

**If you want to add queries params**

**By name**

 fetch("http://localhost:3000/data**?name=teja**")

**By id**

 fetch("http://localhost:3000/data**?id =2**")

**By both**

 fetch("http://localhost:3000/data**?id =2&name=teja**")

**By limit- used for pagination**

 fetch("http://localhost:3000/data**?\_limit=2** ")

**By \_sort**

 fetch("http://localhost:3000/data**?\_sort=-id**")

2. **POST Request**: Sending Data to the Server

fetch('http://localhost:3000/posts', {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json'  // Indicate we are sending JSON

  },

  body: JSON.stringify({                // Data to be added

    title: 'New post',

    author: 'Sam'

  })

})

  .then(response => response.json())    // Parse the JSON response

  .then(data => console.log('Post added:', data))  // Handle the response

  .catch(error => console.error('Error:', error));  // Handle errors

. **Explanation**: This adds a new post with the title "New post" and author "Sam" to the posts endpoint.

**3. PUT Request**: Replacing Existing Data on the Server

  fetch('http://localhost:3000/posts/1', {

  method: 'PUT',

  headers: {

    'Content-Type': 'application/json'

  },

  body: JSON.stringify({

    id: 1,            // `PUT` requires the `id`

    title: 'Updated first post',

    author: 'John Doe'

  })

})

  .then(response => response.json())

  .then(data => console.log('Post updated:', data))

  .catch(error => console.error('Error:', error));

**Explanation**: This updates the post with id: 1, changing its title and author.

4. **PATCH Request**: Partially Updating Data on the Server

  fetch('http://localhost:3000/posts/2', {

  method: 'PATCH',

  headers: {

    'Content-Type': 'application/json'

  },

  body: JSON.stringify({

    author: 'Jane Smith'  // Only update the author field

  })

})

  .then(response => response.json())

  .then(data => console.log('Post partially updated:', data))

  .catch(error => console.error('Error:', error));

**Explanation**: This updates only the author field of the post with id: 2.

5. **DELETE Request**: Removing Data from the Server

  fetch('http://localhost:3000/posts/1', {

  method: 'DELETE'

})

  .then(() => console.log('Post deleted'))  // Handle deletion success

  .catch(error => console.error('Error:', error));

**How to fetch the api data using aync and await**

      async function apidata(){

        const response = await fetch("https://fakestoreapi.com/products");

        const data = await response.json();

        console.log(data);,

      }

      apidata()

fetch the data in local server

        fetch("http://localhost:3000/data")

        .then(response=>{

            if(response.ok){

                return response.json()

            }else{

                return "error code" + response.statusText

            }

        }).then(data=>{

            console.log(data);

        })

**API- application programming interface**

* API stands for Application Programming Interface
* It is a collection of communication protocols used by various programs to form the communication between them.
* In simple words It allows communication between two different software systems.

**How api works**

* Client initiates the request via the api uri (uniform resource identifier)
* The api makes a call to the server after receiving the request
* Then the server sends the response back to API with the information
* Finally, the API transfers data to the client

Types of apis

* Browser APIs: Interact with browser and web page elements (dom api), storage api
* Server Apis: Provide data and services from a server (restfull api)
* Third-party Apis: Offered by external services (social media apis)
* Rest apis: [REST APIs](https://blog.postman.com/rest-api-examples/) are designed to make server-side data readily available by representing it in simple formats such as [JSON](https://blog.postman.com/what-is-json) and [XML](https://blog.postman.com/what-is-xml/).
* Soap apis(**Simple Object Access Protocol**): SOAP uses **XML** exclusively to format data, making it more verbose and complex compared to REST.
* GraphQL: Allows clients to request exactly the data they need, great for complex data relationships. The client controls what data to retrieve, reducing over-fetching or under-fetching of data.
* gRPC (Google Remote Procedure Call): gRPC is a modern, high-performance, open-source RPC (Remote Procedure Call) framework developed by Google suitable for microservices and performance-critical systems.
* WebSockets: WebSocket is a protocol that allows for real-time, two-way communication between the client and server over a single TCP connection. WebSocket can send data in **JSON**, **XML**, or any other format, but the communication is typically binary, Real-time, bi-directional communication for applications like chat and gaming.

**Error handling methods is js**

In JavaScript, try...catch statements are used to handle exceptions (errors) that occur in your code. By using these statements, you can ensure that your code continues to run even if an error occurs.

Error – It is an object that is created to represent a problem that occurs offten with userinput or establishing a connection

Why we need try catch

// case -1 ------> both statements will execute

console.log("hi hello");

console.log("you have reached the end");

// case -2 ------>It interupts the program from line to end because console.lag is not a method it

console.lag("hi hello");

console.log("you have reached the end");

To overcome this we need try catch methods

Syntax

try {

// Code that may throw an error

} catch (err) {

// Code to handle the error

} finally {

// Code that will always run, regardless of error

}

**Explanation**

**try block**: Contains the code that may throw an error. If no errors occur, the code inside the catch block is skipped.

**catch block**: Contains code that will execute if an error occurs in the try block. The err parameter contains the error object.

**finally block** (optional): Contains code that will run after the try and catch blocks, regardless of whether an error was thrown or not.

//try block contains error catch block takes one parameter err means error it will store the error caused by try

try{

    console.lag("hi hello");

}

catch(err){

    console.log(err);

}

We can also use

**console.error(err)**

try{

    console.lag("hi hello");

}

catch(err){

   console.error(err);

}

finally{

    console.log("this will always run");

}

console.log("you have reached the end");

**Throw statement in js**

The throw statement is used to raise an exception in JavaScript. When an exception is thrown, the normal flow of code execution is stopped, and control is passed to the nearest enclosing catch block. If no catch block is found, the script will terminate.

      num = prompt("entry");

      try {

        if (num <= 0) {

          throw new Error("The number must be positive.");

        }

      } catch (err) {

        console.error(err);

      }

      console.log("you have reached the end");

This program prompts the user to enter a number, checks if the number is positive, and handles errors if the number is not positive. It uses a try...catch block to manage potential exceptions and ensures that a final message is logged to indicate the end of the program.

**Try Block**:

* if (num <= 0) { throw new Error("The number must be positive."); }
  + **Condition**: Checks if the input num is less than or equal to 0.
  + **Error Handling**: If the condition is true, an error is thrown with the message "The number must be positive."

**Catch Block**:

* catch (err) { console.error(err); }
  + **Function**: Catches the error thrown in the try block.
  + **Error Logging**: Uses console.error(err) to log the error object to the console, which includes the error message and stack trace.

**Final Log Statement**:

console.log("you have reached the end");

**Purpose**: To indicate that the program has reached its end, regardless of whether an error occurred.

**Types of errors**

**1. Syntax Errors**

* **Description**: Occur when the code contains invalid syntax, which prevents the script from being parsed correctly.

if (true {

console.log("This is a syntax error");

}

**2. Reference Errors**

* **Description**: Occur when trying to reference a variable that is not declared.

console.log(nonExistentVariable);

**3. Type Errors**

* **Description**: Occur when an operation is performed on a value of an inappropriate type

let num = 5;

num.toUpperCase(); // TypeError: num.toUpperCase is not a function

**4. Range Errors**

* **Description**: Occur when a numeric variable or parameter is outside its valid range

 let arr = new Array(-1);

function createArray(size) {

if (size > 100) {

throw new RangeError("Array size is too large");

}

return new Array(size);

}

createArray(101);

**5. Eval Errors**

* **Description**: Occur due to improper use of the eval() function. This error type is rarely encountered and is primarily included for backward compatibility.

eval("alert('Hello’)");// This example won't necessarily throw an EvalError in modern browsers but shows misuse of eval.

**6. URI Errors**

* **Description**: Occur when global URI handling functions are used incorrectly, such as encodeURI(), decodeURI(), encodeURIComponent(), and decodeURIComponent().

decodeURIComponent("%");

**This keyword**

`this` is a keyword that refers to the current calling object or context in which it is used. It is used to access the properties and methods of the current object .

**1.Global Context**: When **this** is used in the global scope (outside of any function), it refers to the global object. In a web browser environment, the global object is **window**.

  console.log(this === window); // true

1. **Function Context**: When **this** is used within a function that is not a method of an object, its value depends on how the function is called. If the function is called as a standalone function, **this** will typically refer to the global object (or **undefined** in strict mode).

  function sayHello() {

    console.log(this);

}

sayHello(); // In a browser, this would log the window object

**3.Method Context**: When **this** is used within a method of an object, it refers to the object that the method is called on.

**//here this refers to the person means object name**

  const person = {

    name: 'John',

    greet: function() {

        console.log('Hello, my name is ' + this.name);

    }

};

person.greet(); // Logs "Hello, my name is John"

**This in arrow function**

        let obj={

            name:"John",

            age:30,

            sayHello:()=>{

                console.log(this.age) //undefined

            }

        }

        obj.sayHello()

        let obj={

            name:"John",

            age:30,

            sayHello:()=>{

                console.log(this)// window

            }

        }

        obj.sayHello()

**Understanding this in Arrow Functions**

Arrow functions (() => { }) in JavaScript behave differently with respect to this compared to regular functions. Here are the key points to note:

1. **Lexical Scope**: Arrow functions do not have their own this context. Instead, they inherit this from the surrounding (lexical) scope where they are defined.
2. **Global Object**: In most cases where obj is defined at the top level (not inside another function or block), this refers to the global object (window in browsers).

**4.This refers to the particular event in evenhandlers**

  document.getElementById('myButton').onclick = function() {

  console.log(this); // Refers to the button element with the ID 'myButton'

    };

**Changing this with call, apply, and bind**

**call** and **apply** immediately invoke the function with a specified this value and arguments.

      function showThis() {

        console.log(this);

      }

      const obj = { name: "John" };

      showThis()// window

      showThis.call(obj); // obj

      showThis.apply(obj); // obj

**bind**

bind returns a new function with a specified this value, without immediately invoking the function.

      const boundFunction = showThis.bind(obj);

      boundFunction(); // obj

**call with parameters**

      function hello(city, profession) {

        console.log(

          "hello my name is " +

            this.name +

            " iam from " +

            city +

            " my profesion is " +

            profession

        );

      }

      obj = {

        name: "john",

      };

      obj2 = {

        name: "peter",

      };

      hello.call(obj, " vizag", " senior trainer");

      hello.call(obj2, "hyd", " developer ");

**apply**

      function hello(city, profession) {

        console.log(

          "hello my name is " +

            this.name +

            " iam from " +

            city +

            " my profesion is " +

            profession

        );

      }

      obj = {

        name: "john",

      };

      obj2 = {

        name: "peter",

      };

      hello.apply(obj, [" vizag", " senior trainer"]);

      hello.apply(obj2, ["hyd", " developer "]);

**bind**

      function hello(city, profession) {

        console.log(

          "hello my name is " +

            this.name +

            " iam from " +

            city +

            " my profesion is " +

            profession

        );

      }

      obj = {

        name: "john",

      };

      var bind1 = hello.bind(obj);

      bind1(" vizag", " senior trainer");

**Key Differences and Use Cases**

**call vs apply**: The primary difference between call and apply is how arguments are passed:

* call passes arguments individually.
* apply expects an array of arguments.

Use call when you know the exact arguments to pass, and use apply when you have arguments in an array or array-like object.

**bind**: Unlike call and apply, bind does not immediately invoke the function. Instead, it creates a new function with the bound this value and optionally prepended arguments. This is useful when you want to create a function with a fixed this value that you can later execute.

**Use cases in real time**

        // 1. Fetching Data in Different Contexts (API Callbacks)

        const apiConfig = { baseURL: 'https://default.api.com' };

        function fetchData(endpoint) {

            console.log(`Fetching from ${this.baseURL}${endpoint}`);

        }

        const weatherAPI = { baseURL: 'https://weather.api.com' };

        const newsAPI = { baseURL: 'https://news.api.com' };

        fetchData.call(weatherAPI, '/current-weather');  // Fetching from https://weather.api.com/current-weather

        fetchData.call(newsAPI, '/latest-news');         // Fetching

        //     Custom Sorting with Dynamic Context

        //    You might want to sort an array of objects based on different properties dynamically. call can set the this context for a comparison function.

        const items = [

            { name: 'Apple', price: 30 },

            { name: 'Banana', price: 20 },

            { name: 'Cherry', price: 25 },

        ];

        function compare(property) {

            return function (a, b) {

                return a[property] - b[property];

            };

        }

       item= items.sort(compare.call(null, 'price')); // Sorts by price

        console.log(item);

**Constructor function**

It is a old way to create objects

Single same objects

 function Person(){

        this.name = "john";

        this.age = 30;

    }

    let person1=new Person();

    console.log(person1);

    let person2=new Person();

    console.log(person2);

Multi objects

  // constructor function - it is a old way to create objects in js

    function Person(name, age){

        this.name = name;

        this.age = age;

    }

    let person1=new Person("Hemanth", "26");

    console.log(person1);

    let person2=new Person("teja", "25");

    console.log(person2);

**Prototype**

    function Person(){

        this.name = "john";

        this.age = 30;

    }

    Person.prototype.course="trainers"

    let person1=new Person();

    console.log(person1.course);

    let person2=new Person();

    console.log(person2.course)

**function Person(name, age) {**

**this.name = name;**

**this.age = age;**

**}**

**function Student(name, age, grade) {**

**// Using call to inherit the properties from Person**

**Person.call(this, name, age);**

**this.grade = grade;**

**}**

**const student1 = new Student('John', 20, 'A');**

**console.log(student1); // Output: { name: 'John', age: 20, grade: 'A' }**

**Javascript classes**

 The old way to create objects was using [constructor functions](https://www.programiz.com/javascript/constructor-function).

     // constructor function

     function Person () {

       this.name = "John",

       this.age = 23

      }

     // create an object

     const person = new Person();

**Oops in js**

OOP in JavaScript provides a **way to structure and organize code** in a more modular and **reusable** manner. By utilizing objects, classes, inheritance, encapsulation, and polymorphism.

1.class

2.object

3.encapsulatiion

4.inheritance

5.polymorphism

6.abstraction

**Constructor class**

Construtor class is a way to define a template for creating object

\* class was introduced recently in [ES6](https://www.programiz.com/javascript/ES6)

         \* class is a collection of properties and methods(static/intance)

         \* syntax :

         \*          class Classname{

         \* }

         \*

         \* obj syntax :

         \*        new Classname(arguments)

\* constructor should be one

**Class Definition**

* **Class**: A class is a template for creating objects. It encapsulates data and functions that operate on that data.
* **Properties (state)**: Variables that belong to the class.
* **Methods (behaviour)**: Functions that belong to the class.

**Ex:-**

**Marker Class**

The **Marker** class has three properties: **color, lid**, and **type**. It also has a constructor to initialize these properties.

      class Marker {

        color; // property to hold the color of the marker

        lid; // property to hold the type of lid of the marker

        type; // property to hold the type of the marker

        // Constructor to initialize the properties

        constructor(color,lid, type) {

          this.color = color; // initializes the color property

          this.lid = lid; // initializes the lid property

          this.type = type; // initializes the type property

        }

        // method initialisation - it is a behaviour of object

        write() {

            console.log("Writing with " + this.color + " marker");

        }

      }

**Creating Objects**

Objects are instances of a class. The new keyword is used to create an object from a class.

      var marker1 = new Marker("red", "square", "permanent");

      var marker2 = new Marker("black", "circle", "temporary");

**Logging the object**

     //logging the objects

     console.log(marker1);

     console.log(marker2);

**Calling the method**

    //calling the methods

    marker1.write();

**Key Points**

* **Class Definition**: Defines the state and behavior of objects.
* **Properties**: Variables within a class that hold data.
* **Methods** : function that holds the behaviour
* **Constructor**: Special method to initialize properties when an object is created.
* **this Keyword**: Refers to the current instance of the class.
* **Creating Objects**: Use new keyword followed by class name and arguments to create objects.

use the class keyword to create a JavaScript class.

// create a class

class Person{

// body of class

};

**To use the class constructor we must need to assign object to it**

    class Person{

        constructor (name,age){

            this.name=name;

            this.age =age;

        }

        hi(a,b){

            return "my name is "+ this.name+ " age is " + this.age;

        }

     }

     var b=new Person("teja",25);

    console.log(b);//{name:teja; age:25}

    console.log(b.age);//25

    console.log(b.hi());//my name is teja age is 25

**by using prototypes we can add properties directly**

**Person.prototype.fullName = “sai teja”**

**Javascript classes with private fields**

**Class Definition**

* **Class**: Employee
  + **Private Fields**: #name, #age, #salary (indicated by # prefix)
  + Introduced to provide encapsulation and data protection within a class.
  + Enhances security and prevents accidental modification of sensitive data.

    class Employee {

    #name;

    #age;

    #salary;

    constructor(name, age, salary) {

        this.#name = "John";    // Private field initialization

        this.#age = 30;         // Private field initialization

        this.#salary = 50000;   // Private field initialization

    }

//private fields can only accessed inside the class

  hello(){

        console.log("Hello, my name is " + this.#name )

}

   }

**Private Fields**

* **Definition**: Private fields are declared using the # symbol before the field name (#name, #age, #salary).
* **Access Control**: Private fields can only be accessed and modified from inside the class where they are defined.

Accessing object will thrown an error because we cannot able to access them in outside

    var person1=new Employee("john", 25, 50000);

    console.log(person1.name)// error - private property cannot be accessed directly

method will print the output because properties can be accessed inside a class

    person1.hello();

**Encapsulation**

Encapsulation in JavaScript refers to the **bundling of data (attributes) and methods (functions) that operate on the data into a single unit**, typically known as an object. This concept helps in hiding the internal state of an object and only exposing the necessary functionalities to interact with that state.

**Access the values using getter**

Syntax:

    // we can giving access

    get methodname(){

      return this.#name;

    }

    // we can giving access

    get accessname(){

      return this.#name;

    }

    class Employee {

    #name;

    #age;

    #salary;

    constructor(name, age, salary) {

        this.#name = name;    // Private field initialization

        this.#age = age;         // Private field initialization

        this.#salary = salary;   // Private field initialization

    }

    hello(){

        console.log("Hello, my name is " + this.#name + " and I am a " )

    }

    // Getter for accessing private field #name

    get accessname(){

      return this.#name;

    }

   }

    var person1=new Employee("john", 25, 50000);

//we can able to access now because of encapsulation

    console.log(person1.accessname)// john - will not thrown an error

**Modify the values using setter**

    // Setter for modifying private field #salary

    set setsalary(sal){

      this.#salary=sal;

    }

// Getter for accessing private field #salary

    get getsalary(){

      return this.#salary;

    }

   //creating a object

    var person1=new Employee("john", 25, 50000);

    //setting the values

    person1.setsalary=99500;

    //accessing the updated values

    console.log(person1.getsalary);

**//complete program**

    class Employee {

    #name;

    #age;

    #salary;

    constructor(name, age, salary) {

        this.#name = name;    // Private field initialization

        this.#age = age;         // Private field initialization

        this.#salary = salary;   // Private field initialization

    }

    hello(){

        console.log("Hello, my name is " + this.#name + " and I am a " )

    }

    // we can giving access

    get accessname(){

      return this.#name;

    }

    //giving the access to change the values

    set setsalary(sal){

      this.#salary=sal;

    }

    get getsalary(){

      return this.#salary;

    }

   }

   //creating a object

    var person1=new Employee("john", 25, 50000);

    //setting the values

    person1.setsalary=99500;

    //accessing the updated values

    console.log(person1.getsalary);

**usecase**

       /\*

       Banking System (Account & Transaction Management)

For a banking system, you can create Account and Transaction classes, where each account can handle deposits, withdrawals, and transaction history.

       \*/

        class Account {

        #balance;

        constructor(owner) {

            this.owner = owner;

            this.#balance = 0;

        }

        deposit(amount) {

            this.#balance += Number(amount);

            console.log(`${this.owner} deposited ${amount}. Balance: ${this.#balance}`);

        }

        withdraw(amount) {

            if (this.#balance >= amount) {

                this.#balance -= amount;

                console.log(`${this.owner} withdrew ${amount}. Balance: ${this.#balance}`);

            } else {

                console.log(`Insufficient balance.`);

            }

        }

    }

    let person1=new Account("john");

    console.log(person1)///logs owner name and blance =0

    person1.deposit("10000")//deposited 10k in first day

    person1.deposit("20000")//deposited 20k in first day

    person1.withdraw("2000")//it withdraws 2k from the account

Let's break down the concept of inheritance in JavaScript using the provided code as an example and create detailed notes.

**JavaScript Inheritance**

**What is Inheritance?**

* **Inheritance**: A feature in object-oriented programming that allows one class (subclass or derived class) to inherit properties and methods from another class (superclass or base class).
* **Purpose**: Promotes code reusability and establishes a natural hierarchy between classes.

**Base Class: Animal**

The Animal class serves as the base class with common properties and methods that can be inherited by other classes.

    class Animal{

      weight;

      height;

      voice;

      constructor(weight, height, voice){

        this.weight = weight;

        this.height = height;

        this.voice = voice;

      }

      eating(){

        console.log("Eating...");

      }

    }

**Subclass: Dog**

The Dog class extends the Animal class, inheriting its properties and methods while adding its own specific properties.

    class Dog extends Animal {

    breed;

    food;

    constructor(weight, height, voice, breed, food) {

        super(weight, height, voice); // Calls the constructor of the Animal class

        this.breed = breed;

        this.food = food;

    }

}

**Properties**:

* Inherited: weight, height, voice
* Specific: breed, food

**Constructor**:

* Calls super(weight, height, voice) to initialize inherited properties.
* Initializes breed and food properties.

**Creating Objects and Using Methods**

      var dog1 = new Dog("15kgs", "2feet", "bow bow", "indie", "nonveg");

      console.log(dog1); // Outputs: Dog { weight: '15kgs', height: '2feet', voice: 'bow bow', breed: 'indie', food: 'nonveg' }

      dog1.eating(); // Outputs: Eating...

**Notes**

1. **Inheritance in JavaScript**:
   * **Base Class**: Defines common properties and methods.
   * **Subclass**: Inherits from the base class and can add or override properties and methods.
2. **super Keyword**:
   * Used in the constructor of the subclass to call the constructor of the base class.
   * Ensures inherited properties are properly initialized before accessing subclass-specific properties.
3. **Constructor Rules**:
   * In a subclass constructor, super must be called before accessing this.
   * Ensures proper initialization of the inherited fields.
4. **Method Inheritance**:
   * Subclasses inherit all methods from the base class.
   * Methods can be overridden in the subclass if needed.
5. **Code Reusability**:
   * Inheritance promotes code reusability by allowing subclasses to use and extend the functionality of base classes.
   * Reduces redundancy and simplifies maintenance.
6. **Example Scenario**:
   * **Base Class**: Animal with common properties (weight, height, voice) and methods (eating).
   * **Subclasses**: Dog and Cat with additional specific properties (breed, food for Dog, and color for Cat).

**What is Polymorphism?**

* **Polymorphism**: A core concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables methods to perform different tasks based on the object they are called on.
* **Purpose**: Enhances flexibility, reusability, and maintainability of code by allowing a single interface to represent different underlying forms (data types).

**Example: Polymorphism in Action**

Using the provided code as an example, we can see how polymorphism works in JavaScript

**Base Class: Animal**

The Animal class serves as the base class with common properties and methods that can be inherited by other classes.

      class Animal {

        weight;

        height;

        voice;

        constructor(weight, height, voice) {

          this.weight = weight;

          this.height = height;

          this.voice = voice;

        }

        eating() {

          console.log("Eating...");

        }

      }

 **Properties**: weight, height, voice

 **Methods**:

* eating(): Prints "Eating..." to the console.

**Subclass: Dog**

The Dog class extends the Animal class, inheriting its properties and methods but also overriding the eating method.

      class Dog extends Animal {

        breed;

        food;

        constructor(weight, height, voice, breed, food) {

          // we cannot write anything above the super

          super(weight, height, voice);

          this.breed = breed;

          this.food = food;

        }

        eating() {

          console.log("Dog is eating " + this.food);

        }

      }

 **Properties**:

* Inherited: weight, height, voice
* Specific: breed, food

 **Methods**:

* Overridden: eating(): Prints "Dog is eating " followed by the type of food the dog eats.

**Object Creation and Method Overriding**

      var dog1 = new Dog("15kgs", "2feet", "bow bow", "indie", "nonveg");

      console.log(dog1);

      // Outputs: Dog { weight: '15kgs', height: '2feet', voice: 'bow bow', breed: 'indie', food: 'nonveg' }

      dog1.eating();

      // Outputs: Dog is eating nonveg

 **Object Creation**: dog1 is created as an instance of the Dog class.

 **Method Overriding**: The eating method in the Dog class overrides the eating method in the Animal class.

**Polymorphism**

Polymorphism, in the context of object-oriented programming, refers to the ability of **different objects to respond to the same message or method invocation** in different ways. It allows objects of different classes to be treated as objects of a common superclass, providing a unified interface for different classes.

**Abstraction**

Abstraction in JavaScript refers to the concept of hiding complex implementation details and showing only the necessary features of an object or function. It allows developers to work with high-level representations without needing to understand all the underlying complexities.

class CoffeeMachine {

constructor() {

this.\_waterAmount = 0; // private-like property (not truly private)

}

// Public method to make coffee (abstracts the complex process)

makeCoffee() {

this.\_heatWater();

this.\_brew();

console.log("Coffee is ready!");

}

// Internal methods that the user doesn’t need to know about

\_heatWater() {

console.log("Heating water...");

}

\_brew() {

console.log("Brewing coffee...");

}

}

const myCoffeeMachine = new CoffeeMachine();

myCoffeeMachine.makeCoffee();

In this example, the makeCoffee method abstracts away the details of the coffee-making process, hiding \_heatWater and \_brew. Users only need to call makeCoffee without worrying about how the coffee is prepared internally.

**ES-6 concepts**

ES6, or ECMAScript 2015, brought significant enhancements to JavaScript, introducing many new features and syntax improvements.

**1) let and const**:

* **let** allows declaring block-scoped variables that can be reassigned.
* **const** declares constants whose values cannot be reassigned.

**2) Arrow Functions:**

Provides a concise syntax for writing functions, with an implicit return if the function body is a single expression.

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

**3) Template Literals:**

Allow embedding expressions inside strings using ${} syntax.

const name = "John";

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

**4) Destructuring Assignment:**

Allows extracting values from arrays or objects into distinct variables.

   \*  Destructuring is a process assigining a value to a single varaible

         \*  Types

         \*         Array

         \*         Object

**5) Spread Syntax**:

* Allows expanding iterable elements in places where zero or more arguments or elements are expected.it creates the deep copy

const arr = [1, 2, 3];

const newArr = [...arr, 4, 5];

**6) Rest operator**

**7) Classes:**

**8) Modules**:

**9) Promises**:

**Rest Parameters:**

 \* Rest operator is introduced in ES6

 \* Rest operator are used in function parameters

 \* Rest operator is denoted by ...

 \* Rest operator is implicitly an array

 \* Rest operator will accepts from 0 to n number of arguments

 \* Rest operator should be the last parameter of function

 \* In Functions parameters rest operators can be only one

 \* Rest Operator is used for better readibility

 \* Syntax :

 \*          ...varName (passed as function parameter)

Allows representing an indefinite number of arguments as an array.

function sum(a,b,...args) {

return …args

}

Sum(1,2,3,4,5,6,7)

**Why spread operator introduced**

var a=[12,2,2,22,] //create an array

var b=a; //assigned a array to another variable

b[0]=500; //change the second variable value

console.log(a);//500,2,2,22 -here value of array a is also changed because here value is not assigned . memory location is assigned in order to overcome this spread operator was introduced.

In JavaScript, when we assign an array or object to another variable, we do not create a new independent copy of the original array or object. Instead, we assign a reference to the original memory location. This means that any changes made to the new variable will also reflect in the original array or object.

**Understanding Shallow Copy**

The behavior observed in the above example is due to shallow copying, where only the reference to the original memory location is copied, not the actual data. To overcome this and create an independent copy of the array, we can use the spread operator or other methods.

By using spread operator

var a=[12,2,2,22,] //create an array

var b=**[...a]**; //create a deep copy of an array using spread operator

b[0]=500; //change the second variable value

console.log(a);//12,2,2,22 --here value of a is not changed

console.log(b);//500,2,2,22 --value of b is only changed because spread operator creates a copy of an array

**Spread operator**

spread operator (...) in JavaScript is a powerful tool used to expand elements of an iterable (like arrays, strings, or objects) into individual elements. It's often used for array manipulation, function arguments, and object spreading.

\* Spread operator is used to spread the data which is present in

array,obj and string

\* Spread operator can be used as function argument

\* Spread operator will create deep copy for array and object

\* Spread is denoted by ...

\* for array [...arrRef] , for object {...objectRef}

**Expanding an Array:**

const arr1 = [1, 2, 3];

const arr2 = [...arr1, 4, 5, 6];

console.log(arr2); // Output: [1, 2, 3, 4, 5, 6]

**Passing Arrays as Function Arguments:**

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

const sum = (a, b, c, d, e) => a + b + c + d + e;

console.log(sum(...numbers)); // Output: 15

**Merging Objects:**

const obj1 = { name: 'john' };

const obj2 = { age: 23 };

const mergedObj = { ...obj1, ...obj2 };

console.log(mergedObj); // Output: { name: 'john', age: 23 }

**shallow copy with the example**

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

const shallowCopyArray = [...originalArray];

shallowCopyArray[0] = 10; // Modify the shallow copy

2

console.log(originalArray); // Output: [1, 2, 3, 4, 5]

console.log(shallowCopyArray); // Output: [10, 2, 3, 4, 5]

**Call by value and call by reference**

understanding the difference between call by value and call by reference is crucial for managing how data is passed and manipulated within your code

**Call by Value**

**Call by Value** means that when a variable is passed to a function, the value of the variable is passed. If the variable is a primitive type (like number, string, boolean, null, undefined, symbol, and bigint), its value is copied to the function parameter. Changes to the parameter do not affect the original variable.

    function changePrimitive(val) {

        val = 100;

    }

    let num = 50;

    changePrimitive(num);

    console.log(num); // Output: 50

num remains 50 even after calling changePrimitive because the function works with a copy of num.

**Call by Reference**

**Call by Reference** means that when a variable is passed to a function, a reference to the variable is passed. If the variable is an object (like arrays, functions, objects, etc.), its reference is passed to the function. Changes to the parameter will affect the original object.

    function changeObject(obj) {

        obj.name = "John";

    }

    let person = { name: "Doe" };

    changeObject(person);

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

the name property of the person object changes to "John" because the function works with a reference to the person object.

**Key Points**

1. **Primitive Types (Call by Value)**:
   * number, string, boolean, null, undefined, symbol, bigint
   * Changes inside the function do not affect the original variable.
2. **Reference Types (Call by Reference)**:
   * object, array, function
   * Changes inside the function affect the original object or array.

**Destructuring**

Destructuring in JavaScript is a powerful feature that allows you to extract values from arrays or properties from objects and bind them to variables in a concise and expressive way. It provides a convenient syntax for extracting data from arrays and objects.

**Array Destructuring:**

1. **We can destructure values individually**
2. **We can can skip elements in the array by using commas.**
3. **If the value at the specified position is undefined, you can assign a default value.**
4. **We can use rest operator (...) to collect the remaining elements into a new array.**
5. **We can destructure nested arrays**

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

const [first, second, ...rest] = numbers; // Extracting values from the array into variables

console.log(first); // Output: 1

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

console.log(rest); // Output: [3, 4, 5]

**Object Destructuring:**

1. **We can destructure values individually**
2. **We can rename variables while destructuring by using a colon (:).**

const { name: fullName, age: yearsOld } = person;

1. **We can set default values for properties that might be undefined.**
2. **Destructuring can be used to extract values from nested objects.**
3. **We can pass remaining values using rest operator**

const person = { name: 'John', age: 30, city: 'New York' };

const { name, age } = person; // Extracting properties from the object into variables

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

console.log(age); // Output: 30

**Nested objects destructuring**

const person = { name: 'John', age: 30, address: { city: 'New York', country: 'USA' } };

const { name, address: { city, country } } = person; // Nested destructuring

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

console.log(city); // Output: 'New York'

console.log(country); // Output: 'USA'

**Modules :-**

modules are reusable pieces of code that encapsulate related functionality and can be exported from one file and imported into another. This allows for better organization, maintainability, and reusability of code in large applications. ES6 (ECMAScript 2015) introduced native support for modules.

Note:- mention type=”module” in script tag to work with the modules

<script **type="module"**>  
import message from "./message.js";  
</script>

There are two types of exports: **Named Exports** and **Default Exports**.

**1)**Named export

     const name = "Jesse";

     const age = 40;

     export {name, age};

Named import

 import { name, age } from "./person.js";

**2)**Default Export

export default message = () => {

const name = "Jesse";

const age = 40;

return name + ' is ' + age + 'years old.';

};

Default import

 import msg from "./message.js";

**sets and maps**

**Sets in JavaScript**

A **Set** is a collection of unique values. This means that no value can occur more than once in a set. Sets allow you to store distinct values of any type, whether primitive or object references.

**Key Characteristics of Sets:**

1. **Unique Values**: No duplicates are allowed. If you try to add the same value multiple times, it will only be added once.
2. **Order of Values**: Insertion order is maintained. The elements are iterated in the order of their insertion.
3. **No Key-Value Pairs**: Unlike objects or maps, a set is just a collection of values (no keys).

**Common Methods for Sets:**

1. **add(value)**: Adds a new value to the set.
2. **delete(value)**: Removes the specified value from the set.
3. **has(value)**: Checks if the value exists in the set.
4. **clear()**: Removes all values from the set.
5. **size**: Returns the number of values in the set.
6. **forEach()**: Iterates over the values in the set.

 **values()**: Returns an iterator over the set’s values.

 **keys()**: Identical to values(), as sets don't have keys.

 **entries()**: Returns an iterator of [value, value] pairs, mimicking the [key, value] behavior of maps.

    let mySet = new Set();

    // Add values

    mySet.add(1);

    mySet.add(2);

    mySet.add(2);  // Won't be added, as 2 is already present

    mySet.add('hello');

    // Check if a value exists

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

    console.log(mySet.size);    // 3

    // Remove a value

    mySet.delete(2);

    // Iterate over the set

    mySet.forEach(value => console.log(value));

**Maps in JavaScript**

A **Map** is a collection of key-value pairs, similar to an object. However, maps have some key differences that make them more useful.

**Key Characteristics of Maps:**

1. **Key-Value Pairs**: Each element in a map is stored as a pair of keys and values.
2. **Any Data Type as Keys**: Unlike objects, where keys are strings or symbols, in maps, any type (primitive or object) can be used as a key.
3. **Map Size**: You can easily determine the number of entries in a map using the size property.

**Common Methods for Maps:**

1. **set(key, value)**: Adds a key-value pair to the map or updates an existing key.
2. **get(key)**: Returns the value associated with the key.
3. **has(key)**: Checks if the map contains the specified key.
4. **delete(key)**: Removes the key and its associated value from the map.
5. **clear()**: Removes all key-value pairs from the map.
6. **size**: Returns the number of key-value pairs in the map.
7. **forEach()**: Iterates over the key-value pairs in the map.

 **keys()**: Returns an iterator over the keys of the Map.

 **values()**: Returns an iterator over the values of the Map.

 **entries()**: Returns an iterator over [key, value] pairs.

    let myMap = new Map();

    // Add key-value pairs

    myMap.set('name', 'John');

    myMap.set(1, 'one');

    myMap.set(true, 'boolean value');

    // Get values

    console.log(myMap.get('name'));  // 'John'

    console.log(myMap.get(1));       // 'one'

    // Check for existence of a key

    console.log(myMap.has(true));    // true

    // Remove a key-value pair

    myMap.delete('name');

    // Iterate over the map

    myMap.forEach((value, key) => {

    console.log(key, value);

    });