**JavaScript**

**What is JavaScript?**

JavaScript is a high-level, interpreted programming language. It's often referred to as a single threaded synchronous , scripting language because it's typically used to write scripts that are embedded in or included from HTML pages and interact with the Document Object Model (DOM) of the web browser. JavaScript is a dynamic, prototype-based language with first-class functions, meaning functions are treated as first-class citizens, allowing them to be assigned to variables, passed as arguments, and returned from other functions.

JavaScript is indeed a high-level, interpreted programming language. Let's break down the key points:

1. High-level: JavaScript is designed to be easy to read and write, abstracting away low-level details and providing developers with a more intuitive syntax.

2. Interpreted: JavaScript code is executed line by line by an interpreter, rather than being compiled into machine code before execution. This allows for rapid development and easy debugging but may result in slightly slower performance compared to compiled languages.

3. Scripting Language: JavaScript is often used for scripting tasks, especially within web development. It's commonly embedded within HTML pages and executed by web browsers to provide dynamic and interactive functionality.

4. Document Object Model (DOM)/ Browser Object Model (BOM): JavaScript interacts with the DOM, which represents the structure of HTML documents as a tree of objects. This allows JavaScript to manipulate the content, structure, and style of web pages dynamically, enabling interactivity and responsiveness. Browser Object Model (BOM) in JavaScript provides access to browser-related features like controlling windows, manipulating the document (web page), navigating browser history, and retrieving information about the user's screen and browser environment. It enables JavaScript to interact with the browser itself, beyond just the content of web pages.

5. Dynamic and Prototype-based: JavaScript is dynamically typed, meaning variable types are determined at runtime rather than compile time. It also uses a prototype-based inheritance model, where objects inherit properties and behaviors from prototype objects rather than through class-based inheritance as in traditional object-oriented languages like Java or C++.

6. First-class functions: Functions in JavaScript are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, returned as values from other functions, and stored in data structures. This makes JavaScript highly flexible and powerful for functional programming paradigms.

7. Single-threaded: JavaScript operates on a single thread within the browser's runtime environment. This means that it can only perform one task at a time from start to finish. For example, when executing JavaScript code within a web browser, it's running on the same thread that handles user interface updates and other browser tasks. This can lead to issues like blocking the UI if long-running tasks are performed.

8. Synchronous: JavaScript is primarily synchronous, meaning that code is executed line by line in the order it appears in the script. Synchronous operations block further execution until they are completed. For example, if a function call is made that performs a time-consuming operation, the entire execution of the script will pause until that operation completes.

However, JavaScript also has asynchronous capabilities:

Asynchronous Operations: JavaScript supports asynchronous programming through mechanisms like callbacks, promises, and async/await. Asynchronous operations allow certain tasks, such as I/O operations (e.g., fetching data from a server), to be performed without blocking the main execution thread. Instead, these operations are delegated to the browser's background tasks or external processes, and the script continues to execute other tasks in the meantime.

Event-driven Architecture: In addition to asynchronous operations, JavaScript's event-driven architecture allows it to respond to user actions and system events without blocking execution. Event handlers are registered to respond to specific events (such as clicks or keyboard input), and the browser's event loop ensures that these handlers are executed when the corresponding events occur.

So while JavaScript is primarily single-threaded and synchronous, it also supports asynchronous programming patterns, which are crucial for building responsive and efficient web applications, especially when dealing with tasks like network requests or file I/O.

Overall, JavaScript's combination of ease of use, flexibility, and broad adoption has made it a fundamental technology for web development and beyond. modern web development and has applications in a wide range of software development scenarios.

**Why JavaScript?**

JavaScript is widely used for web development because it enables interactive and dynamic content on websites. It's supported by all modern web browsers, making it a universal language for client-side scripting. It's also used for server-side development (Node.js), mobile app development (React Native), desktop app development (Electron), game development (Unity), and more.

JavaScript (JS) is a versatile language that can be used for both front-end (FE) and back-end (BE) development.

1. Front-End (FE) Development:

- In front-end development, JavaScript is primarily used to create interactive and dynamic user interfaces on web pages. It allows developers to manipulate the Document Object Model (DOM), handle user interactions, perform client-side form validation, create animations, and fetch data from servers asynchronously.

- JavaScript is often combined with HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets) to create modern web applications. Libraries and frameworks such as React, Angular, and Vue.js further enhance JavaScript's capabilities for building complex and responsive front-end applications.

2. Back-End (BE) Development:

- With the introduction of Node.js, JavaScript can also be used for server-side development. Node.js is a runtime environment that allows developers to run JavaScript code outside the browser, making it possible to build scalable and high-performance back-end applications.

- JavaScript frameworks like Express.js provide a robust and minimalist web application framework for Node.js, enabling developers to create RESTful APIs, handle HTTP requests and responses, interact with databases, and implement server-side business logic using JavaScript.

- Additionally, JavaScript can be used for serverless computing, where developers write functions that run in response to events triggered by external sources (e.g., HTTP requests, database changes) without managing the server infrastructure. Platforms like AWS Lambda and Azure Functions support JavaScript as a language for serverless development.

By leveraging JavaScript for both front-end and back-end development, developers can build full-stack applications entirely using a single programming language, which can streamline development workflows, reduce context switching, and promote code reuse between different parts of the application.

https://www.developer.com/news/stack-overflow-survey-shows-developer-shift/

link for survey

**Where is JavaScript used?**

JavaScript is primarily used in web browsers to enhance the functionality of websites and web applications.

It's also used on the server-side with Node.js to build scalable network applications.

JavaScript can be found in various environments beyond the web, including mobile app development, desktop app development, game development, IoT (Internet of Things), and more.

JavaScript's versatility extends beyond its traditional role in web development. Here's a more detailed elaboration on each point:

1. Web Browsers: JavaScript is the language of the web. It runs in web browsers, enabling developers to create dynamic and interactive web pages. With JavaScript, developers can manipulate the DOM, handle user interactions, perform client-side form validation, make AJAX requests to fetch data from servers asynchronously, create animations, and much more. JavaScript frameworks and libraries like React, Angular, and Vue.js further enhance its capabilities for building modern web applications.

2. Server-Side with Node.js: Node.js is a runtime environment that allows developers to run JavaScript code on the server-side. This opens up new possibilities for building scalable and high-performance network applications. With Node.js, developers can create web servers, RESTful APIs, real-time chat applications, microservices architectures, and more. Node.js leverages JavaScript's non-blocking I/O model, making it well-suited for handling concurrent connections and I/O-bound tasks.

3. Mobile App Development: JavaScript is increasingly used for mobile app development, thanks to frameworks like React Native and Ionic. React Native allows developers to build cross-platform mobile apps using JavaScript and React. It allows for code sharing between iOS and Android platforms while providing a native-like user experience. Ionic, on the other hand, is a framework that allows developers to build hybrid mobile apps using HTML, CSS, and JavaScript, targeting multiple platforms with a single codebase.

4. Desktop App Development: JavaScript is used for desktop app development through frameworks like Electron. Electron combines JavaScript, HTML, and CSS to create cross-platform desktop applications. Apps like Slack, Visual Studio Code, and Discord are built using Electron. With Electron, developers can leverage their existing web development skills to build powerful desktop applications that run on Windows, macOS, and Linux.

5. Game Development: JavaScript is used for game development with libraries like Phaser.js and game engines like Unity. Phaser.js is a fast and lightweight JavaScript game framework that allows developers to create 2D games for the web. Unity, on the other hand, is a popular game engine that supports JavaScript (via UnityScript) alongside other programming languages like C# and Boo. With Unity, developers can create games for various platforms, including desktop, mobile, and console.

6. Internet of Things (IoT): JavaScript is also making inroads into IoT development. Platforms like Johnny-Five and Espruino enable developers to program microcontrollers and embedded devices using JavaScript. With Johnny-Five, developers can interact with hardware components like sensors, motors, and LEDs using a JavaScript API. Espruino provides a JavaScript interpreter that runs directly on microcontrollers, allowing for rapid prototyping and development of IoT applications.

In summary, JavaScript's versatility allows it to be used in a wide range of environments beyond web browsers, including server-side development, mobile app development, desktop app development, game development, and IoT. This ubiquity makes JavaScript a valuable and in-demand skill for developers across various domains.

**How is JavaScript used?**

Introducing JavaScript in HTML is typically done by including `<script>` tags within the HTML document. Here's how you can do it:

**1. Inline JavaScript:**

You can include JavaScript directly within the HTML document using the `<script>` tag. Place the `<script>` tag inside the `<head>` or `<body>` section of your HTML document.

**2. External JavaScript File:**

Alternatively, you can create a separate JavaScript file with a `.js` extension and include it in your HTML document using the `<script>` tag's `src` attribute.

html

**3. Best Practices:**

- Place `<script>` tags at the end of the `<body>` section to ensure that HTML content is loaded before JavaScript execution.

- Use external JavaScript files for better code organization and maintainability.

- Ensure that your JavaScript code is placed within appropriate HTML elements or event listeners to trigger actions at the desired time.

With these methods, you can easily introduce JavaScript into your HTML documents to add interactivity and dynamic behavior to your web pages.

**Variables**

variables are the containers for storing data values. Here are some key notes about a variables:

**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;

**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

**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 };

**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.

**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.

**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

**Global Variables**: Variables declared outside of any function have global scope and can be accessed and modified from any part of the script.

**Local Variables**: Variables declared within a function have local scope and are only accessible within that function.

**Variable Reassignment**: Variables declared with **var** and **let** can be reassigned, while variables declared with **const** cannot be reassigned, though their properties can be modified if they are objects.

      let x = 5;

      x = 10; // This is valid

**Initialization**: It’s recommended to initialize variables when declaring them to avoid unexpected behavior due to **undefined** values.

      x = 5;

**Use of Strict Mode**: Using **"use strict";** at the beginning of a script or function enables strict mode, which helps catch common coding errors and prevents certain actions

**Data types**

**Primitive Data Types**:primitive data types are the fundamental building blocks used to represent single values. Primitive data types are directly stored in memory and are immutable, meaning their values cannot be changed after they are created.

**Number**: Represents numeric values, including integers and floating-point numbers.

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

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

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

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

**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.

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

**Array**: Represents an ordered collection of elements, accessed by index, starting from zero.

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

**Null & Undefined**

**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**.

**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.

**Type Conversion**

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 Conversion**

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

**Explicit Conversion**: 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.

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

let num = 123; let str = String(num); console.log(str); // Output: "123"

**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

**Operators:**

Operators are symbols used to perform operations on variables and values

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

      let increment = x++; // Increment

      let decrement = y--; // Decrement

**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

**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)

**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 end**

**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**

**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'

**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 ?? "man";

        console.log("hello " + b);

      }

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

      hello("teja")

**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

**Bitwise operator**

Bitwise operators in JavaScript perform operations on binary representations of numbers. These operators treat their operands as a sequence of 32 bits (zeros and ones) and perform operations at the binary level.

**AND (&)**

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 (|)**

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)**

**XOR (^)**

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)**

**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!"

**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**

**Expression Evaluation**: The **expression** inside the switch statement is evaluated once.

**Case Matching**: The result of the expression is compared with the values specified in each **case** clause using strict equality (**===**).

**Code Execution**: If a match is found, the code block associated with that **case** is executed.

**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).

**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; increment/decrement) {

// 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);

**for...in loop**: Used to iterate over the properties of an object. 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.

**Functions**

In JavaScript, functions are blocks of reusable code that can be defined and invoked to perform a specific task or calculate a value. Functions are one of the fundamental building blocks of JavaScript programming, and they play a crucial role in organizing and structuring code, promoting code reuse, and encapsulating logic.

**Let's learn functions in a simple way:**

Imagine you're baking cookies. You have a recipe that tells you exactly what ingredients you need and what steps to follow. Now, think of a function as a mini-recipe within the bigger recipe.

**Here's how it works:**

**Ingredients (Parameters):**

Just like in a recipe, you have specific ingredients like flour, sugar, and eggs. In a function, you have parameters, which are like placeholders for values that you'll use inside the function.

**Instructions (Code):**

In your mini-recipe (function), you have a set of instructions to follow. These instructions tell you what to do with the ingredients. Similarly, in a function, you have a block of code that performs a specific task.

**Usage (Invocation):**

Once you have your mini-recipe (function) prepared, you can use it whenever you need it. You can call the function, just like you'd follow the steps in your mini-recipe to make cookies.

**Output (Return Value):**

After following the instructions in your mini-recipe, you get something in return, like delicious cookies. Similarly, a function can also give you something back after performing its task. This is called the return value.

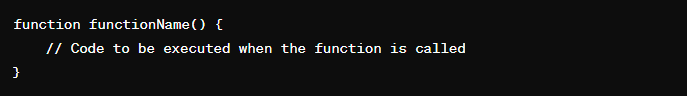
**Here's a simple analogy:**

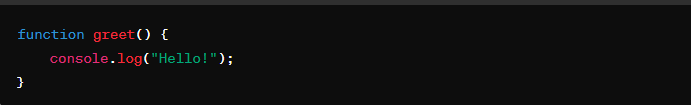
Imagine you have a function called "MakeCookies". You give it ingredients like flour, sugar, and eggs (parameters). Inside the "MakeCookies" function, there are instructions on how to mix the ingredients and bake them to make delicious cookies (code). When you want to make cookies, you simply call the "MakeCookies" function and pass it the ingredients you have. Then, the function executes its instructions and gives you back the freshly baked cookies (return value).

In summary, a function is like a mini-recipe that takes some inputs, performs a specific task, and gives you an output. It helps keep your code organized, reusable, and easy to understand, just like a recipe helps you bake delicious treats

**Function Declaration:**

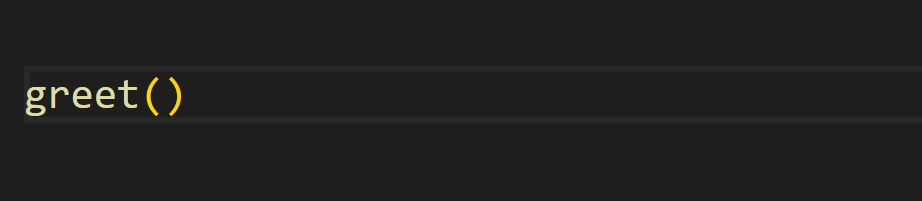
You can define a function using the function keyword followed by the function name, parameters (optional), and the function body enclosed in curly braces {}.

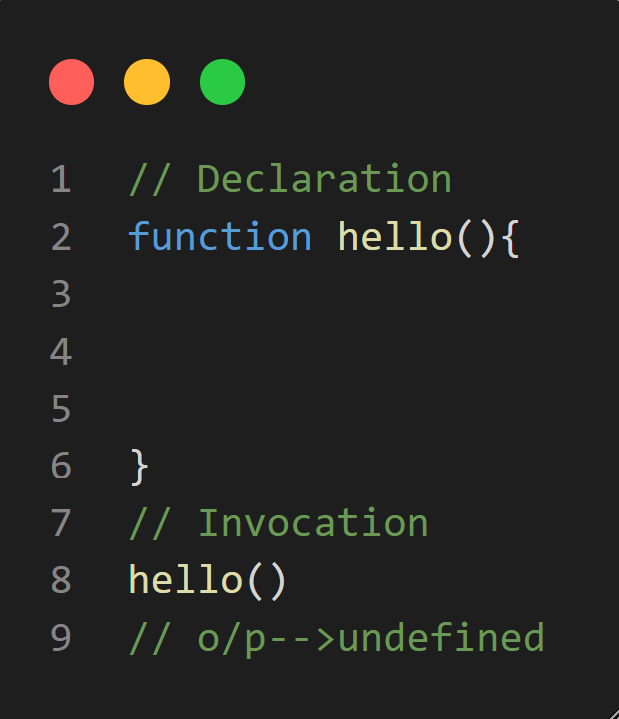




**Function Invocation:**

To execute or invoke a function, you simply use its name followed by parentheses (), optionally passing any required arguments.





In JavaScript, if a function doesn't explicitly return a value, it implicitly returns undefined. This means that when the function is called and executed, but no explicit return statement is encountered, the function returns undefined by default.

**"parameter" and "argument"**

**1. Parameter:**

- In the context of functions, a parameter is a variable name listed in the function declaration. It represents a value that the function expects to receive when it is called.

- Parameters are essentially placeholders for values that will be provided to the function when it is invoked.

- Parameters are defined in the function declaration and act as local variables within the function body.

- Parameters define the function's interface, specifying what kind of input the function can accept.

**2. Argument:**

- An argument is a value that is passed to a function when the function is called or invoked.

- Arguments are the actual values supplied to the function during its execution.

- When a function is called, the arguments are passed to the corresponding parameters in the function definition.

- Arguments can be literals, variables, expressions, or even other functions.

In summary, parameters are variables declared in the function declaration, representing values that the function expects to receive, while arguments are the actual values passed to the function when it is called, matching the parameters specified in the function declaration.



Here we define a function named greet that takes one parameter, name.

Inside the function, we use the parameter name to construct a greeting message.

When we call the greet function, we pass actual values, known as arguments, for the name parameter.

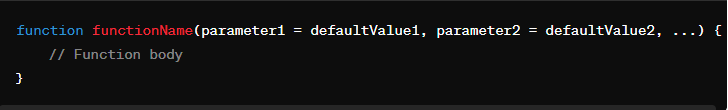
The function is called twice with different arguments: "Alice" and "Bob".

Each time the function is called, it receives the corresponding argument and logs a greeting message to the console.

Here, name is a parameter in the function definition, and "Alice" and "Bob" are the arguments passed to the function when it is called. The function uses these arguments to perform its task, which, in this case, is to greet the provided name.

**default parameters:**

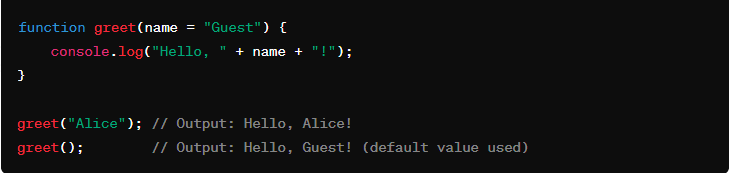
Default parameters allow you to define parameters with default values that will be used if no argument is provided for that parameter when the function is called. This feature was introduced in ECMAScript 6 (ES6).



In this syntax:

- `parameter1`, `parameter2`, etc.: Parameters of the function.

- `defaultValue1`, `defaultValue2`, etc.: Default values assigned to the parameters if no argument is provided when the function is called.



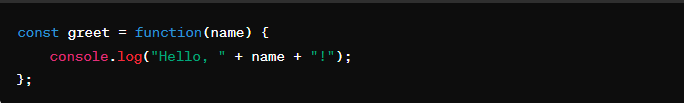
In this example, the `greet` function has a single parameter `name` with a default value of `"Guest"`. When the function is called with an argument (`greet("Alice")`), it uses the provided argument (`"Alice"`) as the `name`. However, when the function is called without any argument (`greet()`), it uses the default value `"Guest"` for the `name` parameter.

**Functions as first-class citizens**

In JavaScript, functions are considered first-class citizens, which means they can be treated as values and used in the same way as other types of values, such as strings, numbers, or objects. This characteristic of functions as first-class citizens enables several powerful programming paradigms, including functional programming and higher-order functions. Here's what it means for functions to be first-class citizens in JavaScript

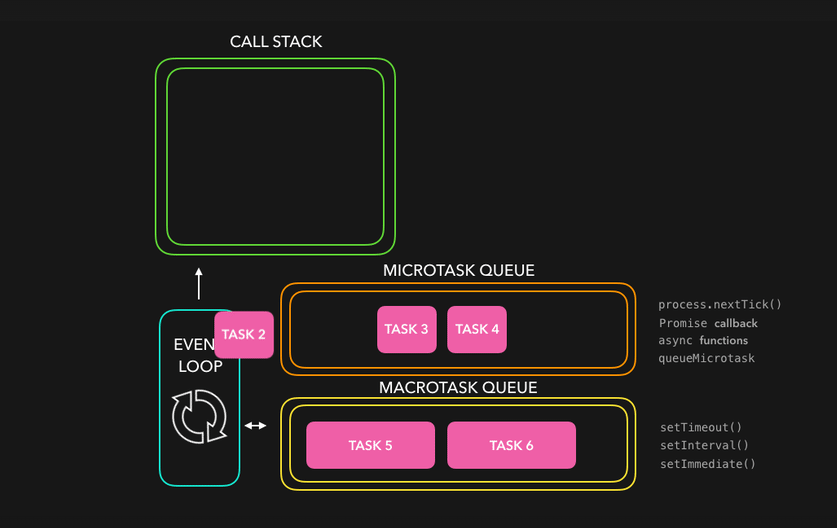
**Functions Can Be Assigned to Variables:**

You can assign a function to a variable, just like you would assign a string or a number.



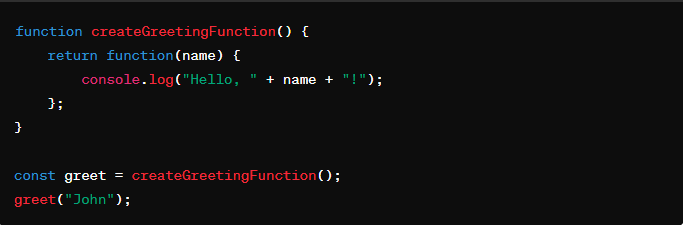
**Functions Can Be Passed as Arguments to Other Functions:**

You can pass a function as an argument to another function.



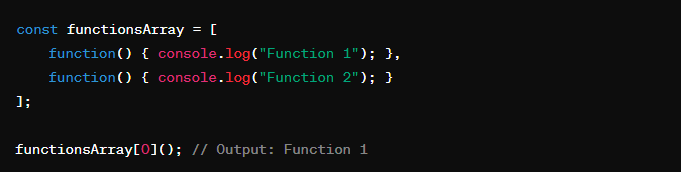
**Functions Can Be Returned as Values from Other Functions**:

You can return a function from another function.



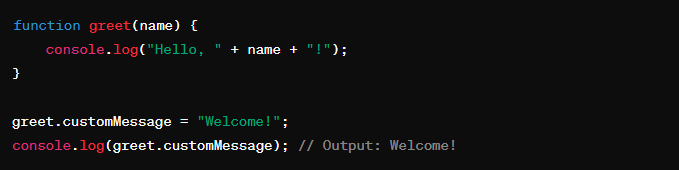
**Functions Can Be Stored in Data Structures:**

You can store functions in arrays, objects, or other data structures.



**Functions Can Have Properties and Methods:**

Functions are objects in JavaScript and can have properties and methods like any other object.

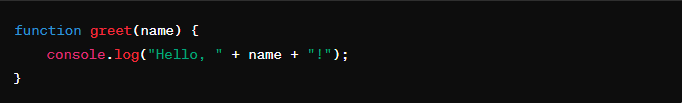


**Types of functions**

In JavaScript, there are several types of functions, each serving different purposes and offering unique features. Here are some common types of functions:

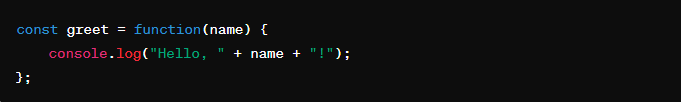
**1. Named Function Declaration:**

- This is the most common type of function declaration, where you specify a name for the function.



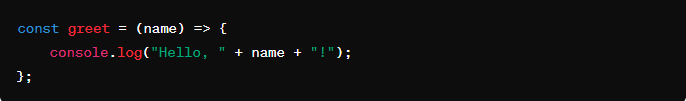
**2. Function Expression:**

- A function expression is when you define a function as part of an expression. It can be anonymous or named.



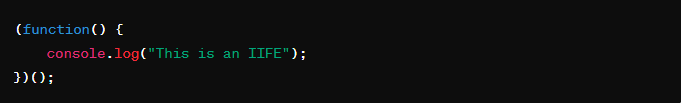
**3. Arrow Function:**

- Arrow functions provide a concise syntax for defining functions, especially for short, single-expression functions.



**4. Immediately Invoked Function Expression (IIFE):**

- An IIFE is a function that is executed immediately after it's defined. It's often used to create a separate scope and prevent polluting the global namespace.



**5. Generator Function:**

- Generator functions allow you to define functions that can pause execution and yield intermediate results using the `yield` keyword.



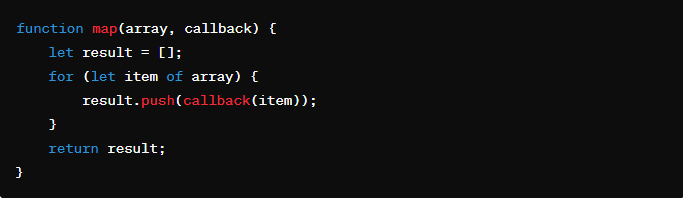
**6. Constructor Function:**

- Constructor functions are used to create objects with a specific structure and behavior. They are often used with the `new` keyword.



**7. Higher-Order Function:**

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



**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.

**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”.

**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);

**STRING**

A string is a data type used to represent text rather than numeric data. It's called a "string" because it's essentially a string of characters, where a character is a single unit of text, such as a letter, digit, punctuation mark, or whitespace.

Strings are for **storing text**

Strings are written **with quotes (single / double )**

**Here are some key points about strings:**

1**. Immutable**: JavaScript, strings are immutable, meaning once they are created, their value cannot be changed. Any operation that appears to modify a string actually creates a new string with the modified value.

2**. Encapsulation**: Strings are typically enclosed within quotation marks, either single (' ') or double (" "). This encapsulation helps the programming language distinguish between the actual text and other parts of the code.

3. **Character Encoding**: Internally, strings are represented using character encoding, which assigns a unique numerical value to each character. The most common encoding schemes are ASCII (American Standard Code for Information Interchange) and Unicode, which supports a much wider range of characters, including various alphabets, symbols, and emojis.

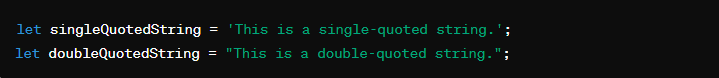
4. **Length**: The length of a string is the number of characters it contains. This can vary depending on the language and the encoding used. In JavaScript, you can get the length of a string using the `length` property.

5**. Operations**: Strings support various operations, such as concatenation (joining two or more strings together), substring extraction (getting a portion of a string), searching for substrings, converting case (to uppercase or lowercase), and more.

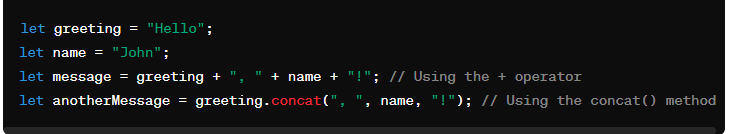
6. **Escaping Characters**: certain characters have special meanings within strings. To include these characters as part of the string itself, you often need to escape them using special sequences, such as `\n` for a newline or `\t` for a tab.

Strings are fundamental in programming and are used extensively in tasks ranging from simple text manipulation to more complex tasks like parsing data, pattern matching, and working with file input/output. Understanding how to work with strings effectively is a crucial skill for any programmer.

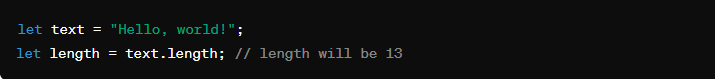
**Declaration**: You can declare a string by enclosing text within either single (' ') or double (" ") quotation marks. JavaScript allows flexibility in choosing between single or double quotes, but it's essential to be consistent.



**Concatenation**: You can concatenate (combine) strings using the `+` operator or the `concat()` method.



**String Length**: You can find the length of a string using the `length` property.

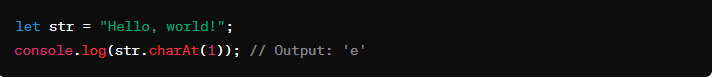


**Accessing Characters:** You can access individual characters in a string using bracket notation with the character's index.



**String charAt():**

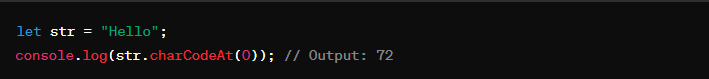
Returns the character at a specified index in a string.



Explanation: The `charAt()` method returns the character at the specified index in the string. Indexing starts from 0.

**String charCodeAt():**

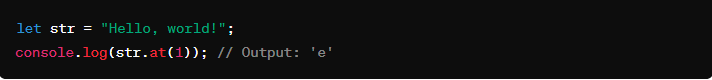
Returns the Unicode value of the character at a specified index in a string.



Explanation: This method returns the Unicode value (a numeric value) of the character at the specified index in the string.

**String at():**

Returns the character at a specified index in a string.



Explanation: Similar to `charAt()`, `at()` returns the character at the specified index in the string.

**String [ ]:**

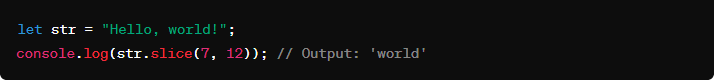
Provides a way to access the character at a specified index in a string.



Explanation: This is another way to access characters in a string using bracket notation.

**String slice():**

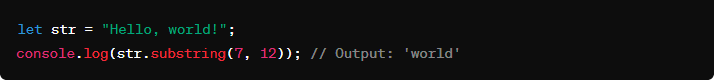
Extracts a section of a string and returns it as a new string.



Explanation: The `slice()` method extracts a portion of the string between the specified start and end indices.

**String substring():**

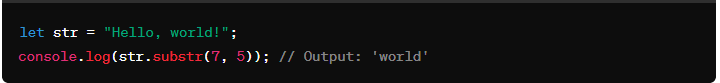
Extracts the characters from a string between two specified indices and returns the new substring.



Explanation: Similar to `slice()`, `substring()` extracts a portion of the string between the specified start and end indices.

**String substr():**

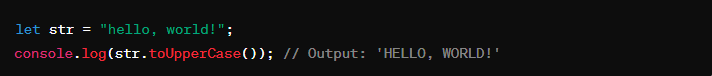
Extracts a specified number of characters from a string, starting at a specified index.



Explanation: The `substr()` method extracts the specified number of characters starting from the specified index.

**String toUpperCase():**

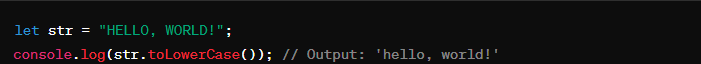
Converts a string to uppercase letters.



Explanation: This method converts all the characters in a string to uppercase.

**String toLowerCase():**

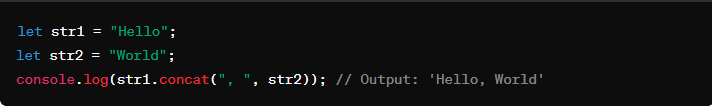
Converts a string to lowercase letters.



Explanation: This method converts all the characters in a string to lowercase.

**String concat():**

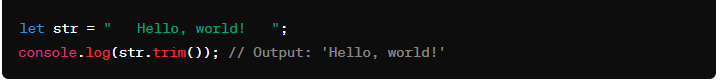
Concatenates two or more strings and returns a new string.



Explanation: The `concat()` method joins two or more strings together.

**String trim():**

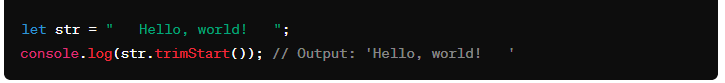
Removes whitespace from both ends of a string.



Explanation: This method removes whitespace (spaces, tabs, and newlines) from the beginning and end of a string.

**String trimStart():**

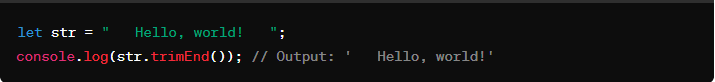
Removes whitespace from the beginning of a string.



Explanation: Similar to `trim()`, `trimStart()` removes whitespace only from the beginning of a string.

**String trimEnd():**

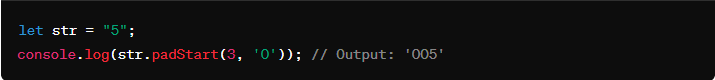
Removes whitespace from the end of a string.



Explanation: Similar to `trim()`, `trimEnd()` removes whitespace only from the end of a string.

**String padStart():**

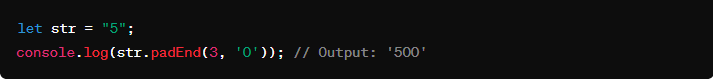
Pads a string with another string until the resulting string reaches the specified length.



Explanation: This method pads the current string with another string (in this case, '0') until the resulting string reaches the specified length.

**String padEnd():**

Pads a string with another string until the resulting string reaches the specified length.



Explanation: Similar to `padStart()`, `padEnd()` pads the current string with another string until the resulting string reaches the specified length, but it adds padding to the end.

**String repeat():**

Returns a new string containing the specified number of copies of the string on which it was called.



Explanation: This method creates and returns a new string by concatenating the specified number of copies of the string on which it was called.

**String replace():**

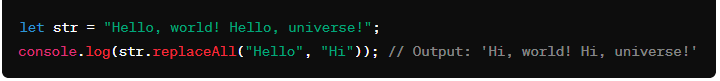
Searches a string for a specified value or regular expression and returns a new string where the specified values are replaced.



Explanation: This method searches a string for a specified value or regular expression and returns a new string where the specified values are replaced with another string.

**String replaceAll():**

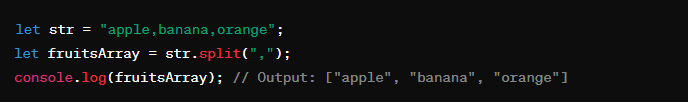
Searches a string for all occurrences of a specified value or regular expression and returns a new string where all occurrences are replaced.



Explanation: Similar to `replace()`, `replaceAll()` replaces all occurrences of a specified value or regular expression with another string.

**String split():**

Splits a string into an array of substrings based on a specified separator.



Explanation: In this example, the split() method splits the original string str into an array of substrings using the comma , as the separator. Each substring between the commas becomes an element in the resulting array.

**String indexOf():**

Returns the index within the calling String object of the first occurrence of the specified value, starting the search at fromIndex. Returns -1 if the value is not found.

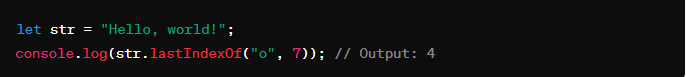




**String lastIndexOf():**

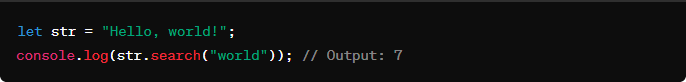
Returns the index within the calling String object of the last occurrence of the specified value. Returns -1 if the value is not found.





**String search():**

Executes a search for a match between a regular expression and this String object.



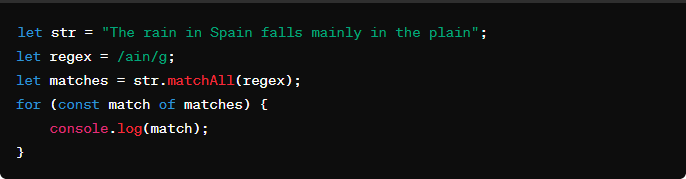
**String match():**

Retrieves the result of matching a string against a regular expression.

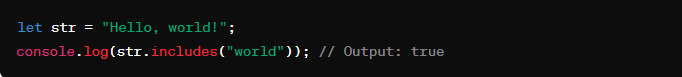


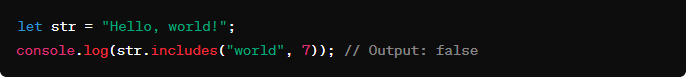
**String matchAll():**

Returns an iterator of all results matching a string against a regular expression, including capturing groups.



**String includes():**

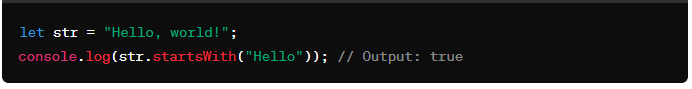




Determines whether one string may be found within another string, returning true or false as appropriate.

**String startsWith():**

Determines whether a string begins with the characters of a specified string, returning true or false as appropriate.



**String endsWith():**

Determines whether a string ends with the characters of a specified string, returning true or false as appropriate.



**Template Literals**: ES6 introduced template literals, which are enclosed by backticks (` `) and allow embedding expressions and multiline strings.

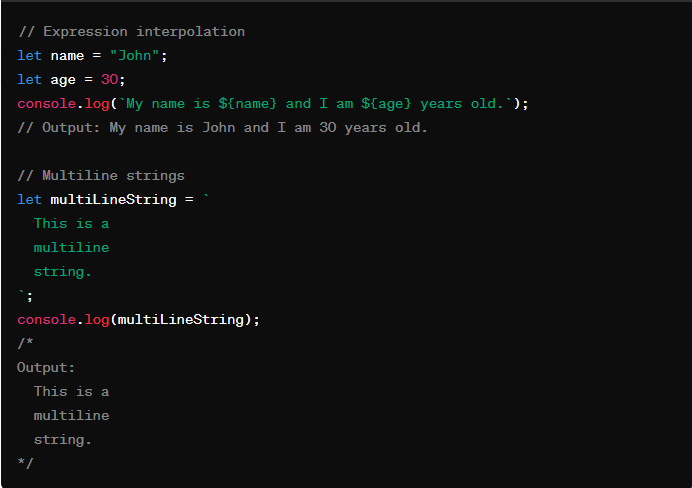
Template literals, introduced in ECMAScript 6 (ES6), provide a more convenient and flexible way to work with strings in JavaScript. They are enclosed by backticks (\`\`) rather than single or double quotes and allow for easy embedding of expressions and multiline strings.

**Expression Interpolation:**

Template literals allow you to embed expressions directly into the string by wrapping them in `${}`. The expression within `${}` is evaluated and its result is inserted into the string.

**Multiline Strings:**

With template literals, multiline strings can be created without the need for escape characters like `\n`. You can simply write multiline text as it appears, making code more readable.



In the first example, `${name}` and `${age}` are expressions that are evaluated and inserted into the string. This feature makes it easy to include dynamic values within strings.

In the second example, the string spans multiple lines without the need for escape characters. This improves code readability, especially for longer strings or when formatting text blocks.

template literals provide a cleaner and more expressive way to work with strings in JavaScript, particularly in scenarios where dynamic content or multiline text is involved.

**Objects**

An object in JavaScript is a collection of 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:**

**Literal notation:**

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

**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]); });

**Object Methods in JS**

**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"]

**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]

**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]]

**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 }

**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

**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 }

**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 }

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

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

**How to Iterate 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}`);

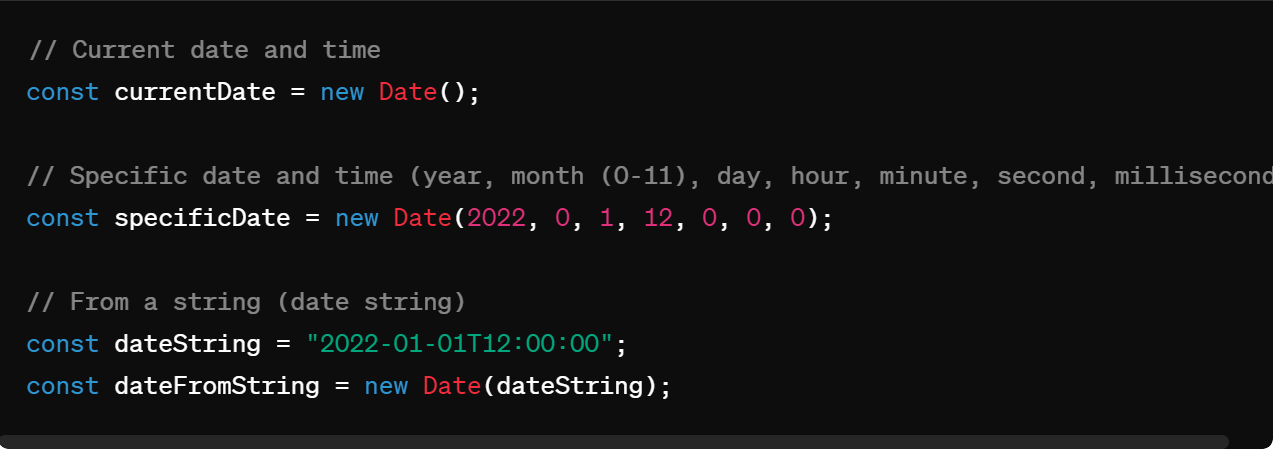
}

**Dates**

Dates in JavaScript are represented by the `Date` object, which is built into the language. Here's a more in-depth explanation covering various aspects of working with dates in JavaScript:

**Creating Date Objects:**

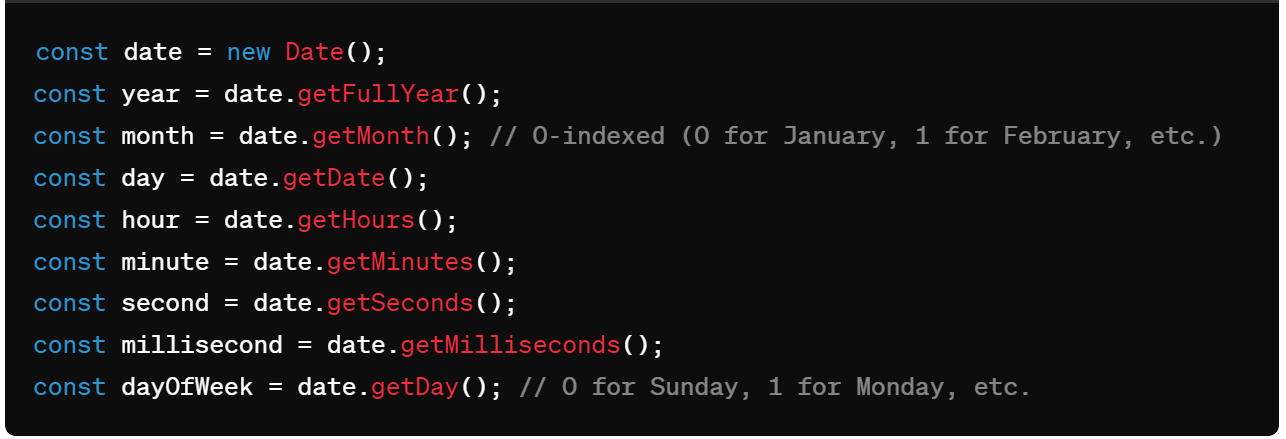
You can create a new `Date` object using several methods:



These methods allow you to work with dates in various formats and contexts.

**Getting Date Components:**

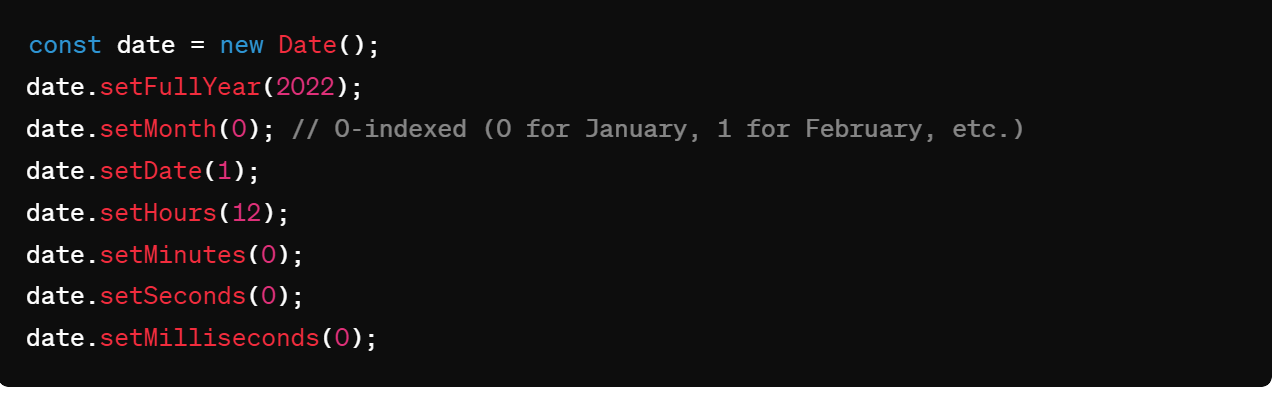
The `Date` object provides methods to extract various components of a date, such as year, month, day, hour, minute, second, millisecond, and day of the week:



These methods enable you to access individual parts of a date for further processing or display.

**Setting Date Components:**

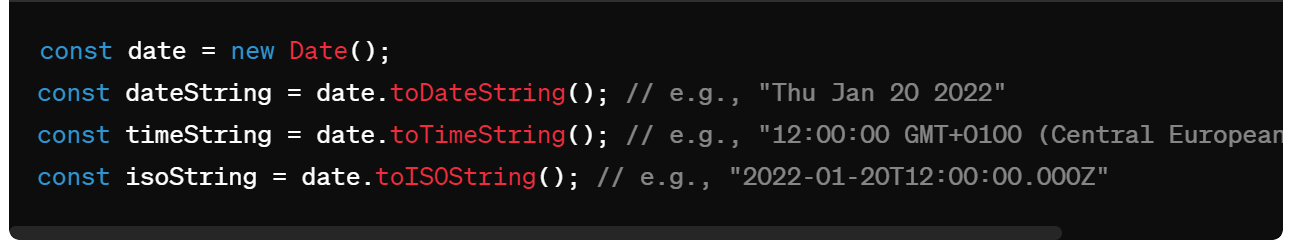
You can also set various components of a date using `set` methods provided by the `Date` object:



These methods allow you to modify specific parts of a date as needed.

**Formatting Dates:**

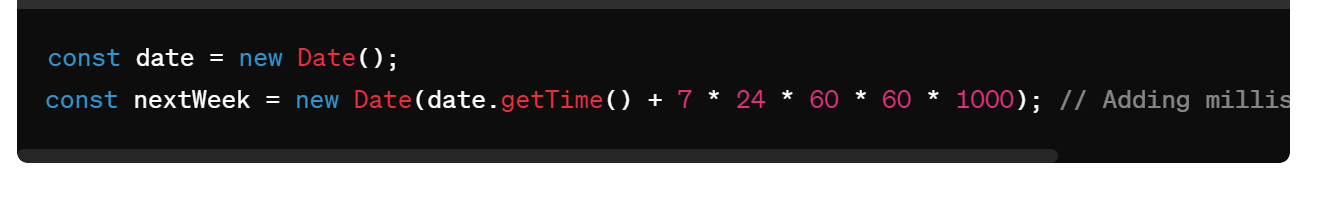
JavaScript provides methods to format dates as strings in different formats:

****

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

**Date Arithmetic:**

You can perform arithmetic operations on dates by adding or subtracting milliseconds:



This allows you to calculate dates in the future or past based on a reference date.

**Date Comparison:**

Dates can be compared using comparison operators (`<`, `>`, `<=`, `>=`) or by comparing their millisecond values:



**Numbers**

In JavaScript, numbers are fundamental data types used to represent numeric values. Here's a brief explanation of numbers in JavaScript:

**Data Type:** Numbers in JavaScript are represented using the `Number` data type. They can hold both integer and floating-point values.

**Numeric Literals**: Numeric literals are written numbers in JavaScript code. They can be integers (whole numbers) or floating-point numbers (numbers with decimal points).

**Arithmetic Operations:** JavaScript supports various arithmetic operations on numbers, including addition (`+`), subtraction (`-`), multiplication (`\*`), division (`/`), and modulus (`%`).

**Mathematical Functions:** The `Math` object in JavaScript provides a set of built-in mathematical functions and constants. These include functions like `Math.sqrt()` (square root), `Math.sin()` (sine), `Math.cos()` (cosine), and constants like `Math.PI` (the mathematical constant π).

**Special Values**: JavaScript has special values such as `NaN` (Not-a-Number), which represents the result of an undefined or unrepresentable mathematical operation, and `Infinity` and `-Infinity`, which represent positive and negative infinity respectively.

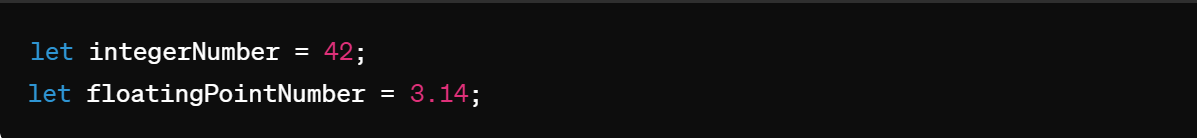
**Number Methods**: The `Number` object in JavaScript provides methods for working with numbers, including formatting methods like `toFixed()` and `toPrecision()`, conversion methods like `toString()`, and parsing methods like `parseInt()` and `parseFloat()`.

**Number Constants**: JavaScript defines several constants related to numbers, such as `Number.MAX\_VALUE` (the largest representable number), `Number.MIN\_VALUE` (the smallest positive number greater than zero), and `Number.EPSILON` (the smallest difference between two representable numbers).

Numbers in JavaScript are represented as 64-bit floating-point values according to the IEEE 754 standard, commonly known as "double-precision" floating-point numbers

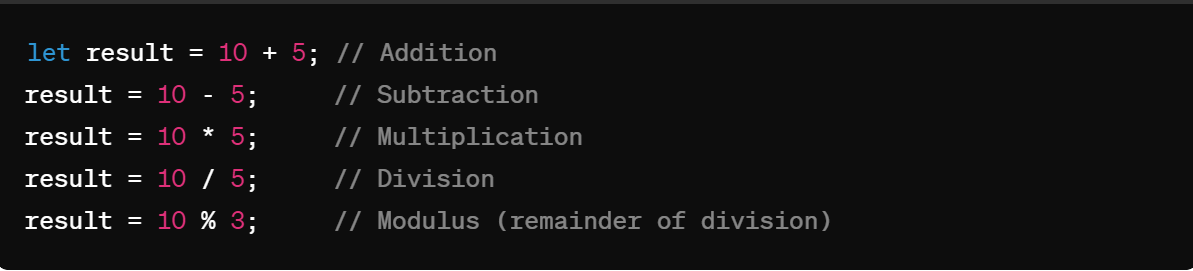
**Integer and Floating-Point Numbers:**

JavaScript supports both integer and floating-point numbers. All numbers in JavaScript, whether integer or floating-point, are represented using the `Number` data type.



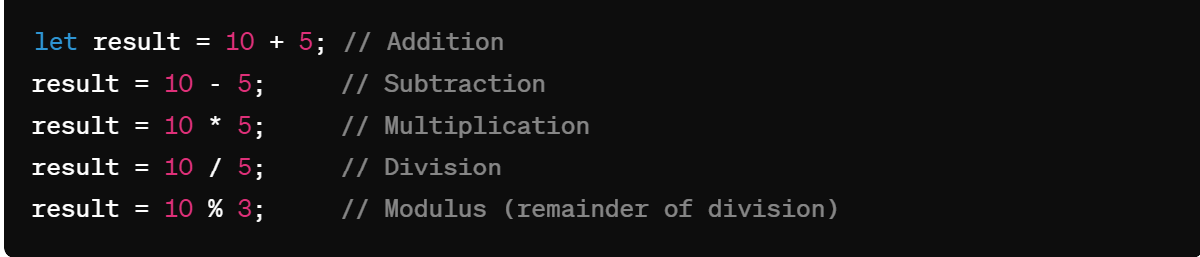
**Numeric Literals:**

Numeric literals are simply written numbers in JavaScript code. They can be integers or floating-point numbers. For example:



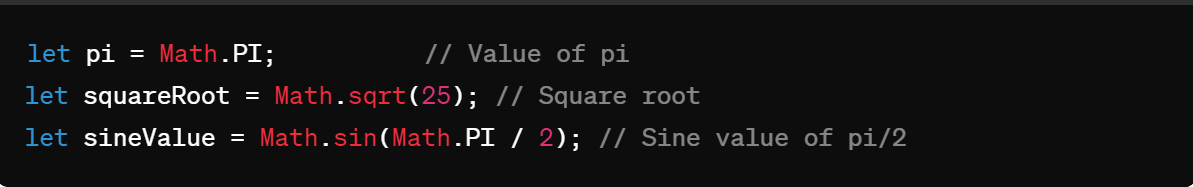
**Mathematical Operations:**

JavaScript supports various mathematical operations, including addition (`+`), subtraction (`-`), multiplication (`\*`), division (`/`), and modulus (`%`). For example:



**Math Object:**

JavaScript provides a built-in `Math` object that contains various mathematical functions and constants. These include trigonometric functions, exponential functions, logarithmic functions, and more. For example:



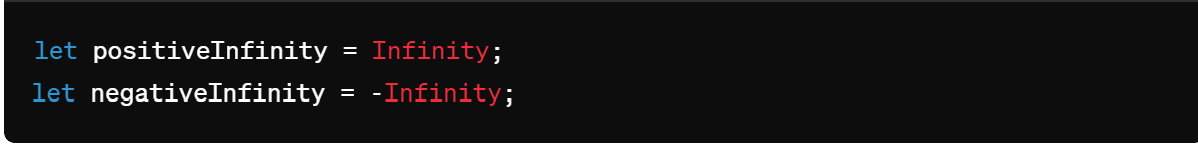
**NaN (Not a Number):**

`NaN` is a special value representing "Not-a-Number." It is returned when a mathematical operation results in an undefined or unrepresentable value. For example:



**Infinity and -Infinity:**

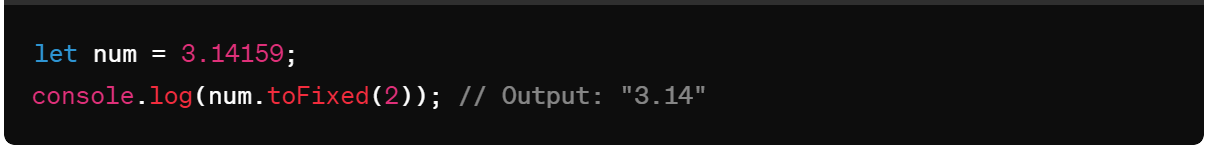
`Infinity` represents positive infinity, while `-Infinity` represents negative infinity. These values are returned when a number exceeds the upper or lower limit of representable values. For example:



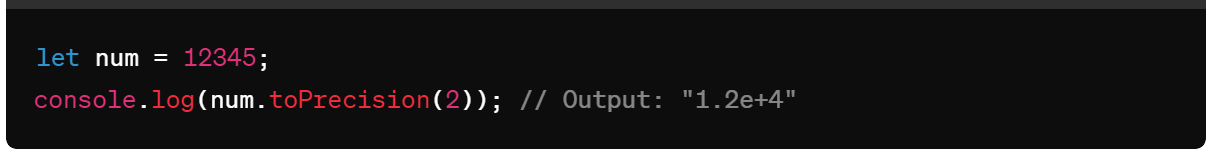
**Number Methods:**

The `Number` object in JavaScript also provides several methods for working with numbers. These include `toFixed()`, `toPrecision()`, `toString()`, `parseInt()`, `parseFloat()`, and more.

**toFixed**(): Converts a number into a string, rounding the number to a specified number of decimal places.



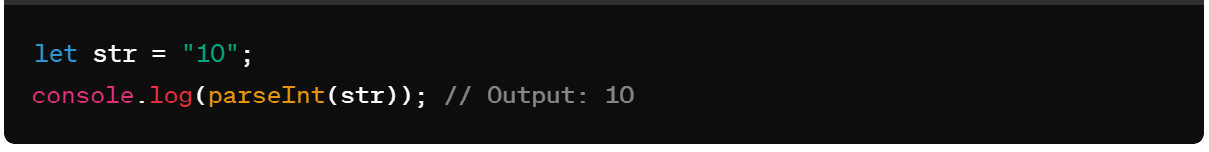
**toPrecision**(): Returns a string representing the number to a specified precision.



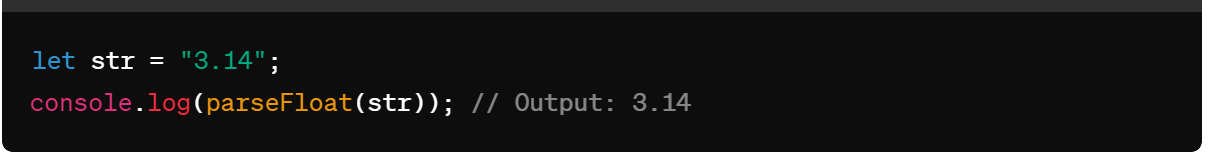
**toString**(): Returns a string representing the specified number.



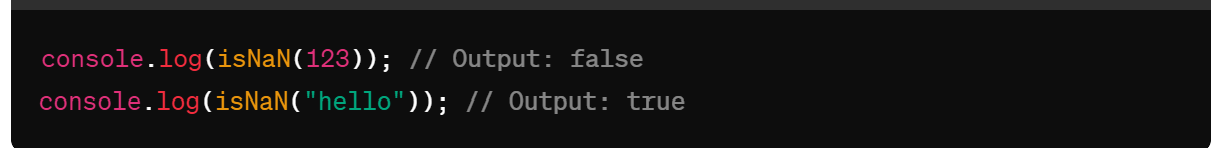
**parseInt**(): Parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).



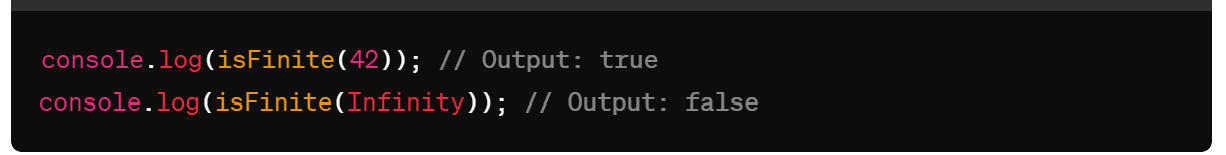
**parseFloat**(): Parses a string argument and returns a floating point number.



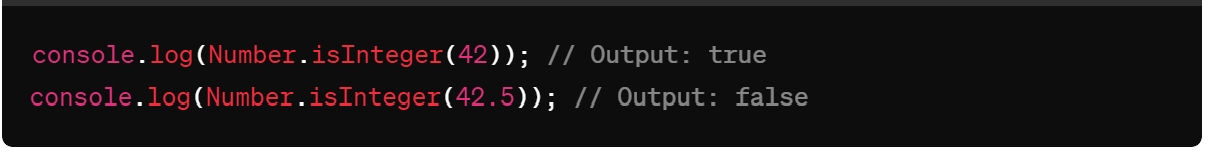
**isNaN**(): Determines whether a value is NaN (Not-a-Number).



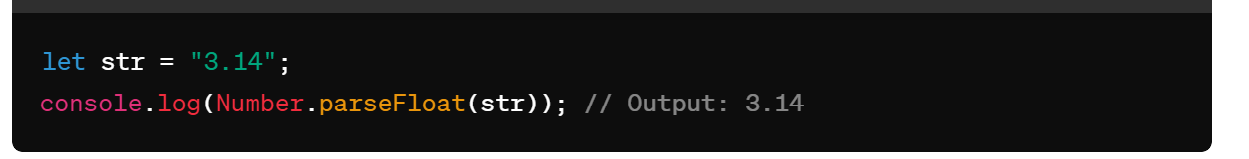
**isFinite**(): Determines whether a value is a finite, legal number.



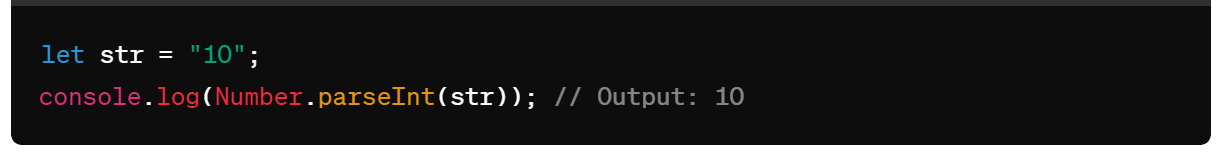
**Number.isInteger**(): Determines whether the passed value is an integer.



**Number.parseFloat**(): Parses an argument and returns a floating point number.



**Number.parseInt**(): Parses a string argument and returns an integer of the specified radix.



These methods are useful for various tasks such as formatting numbers, converting numbers to strings, parsing strings into numbers, and checking number-related properties.

**Math**

JavaScript provides the `Math` object, which contains a set of properties and methods for mathematical operations.

Math.PI: Returns the mathematical constant π (pi).

Math.abs(): Returns the absolute value of a number.

Math.ceil(): Rounds a number up to the nearest integer.

Math.floor(): Rounds a number down to the nearest integer.

Math.round(): Rounds a number to the nearest integer.

Math.max(): Returns the largest of zero or more numbers.

Math.min(): Returns the smallest of zero or more numbers.

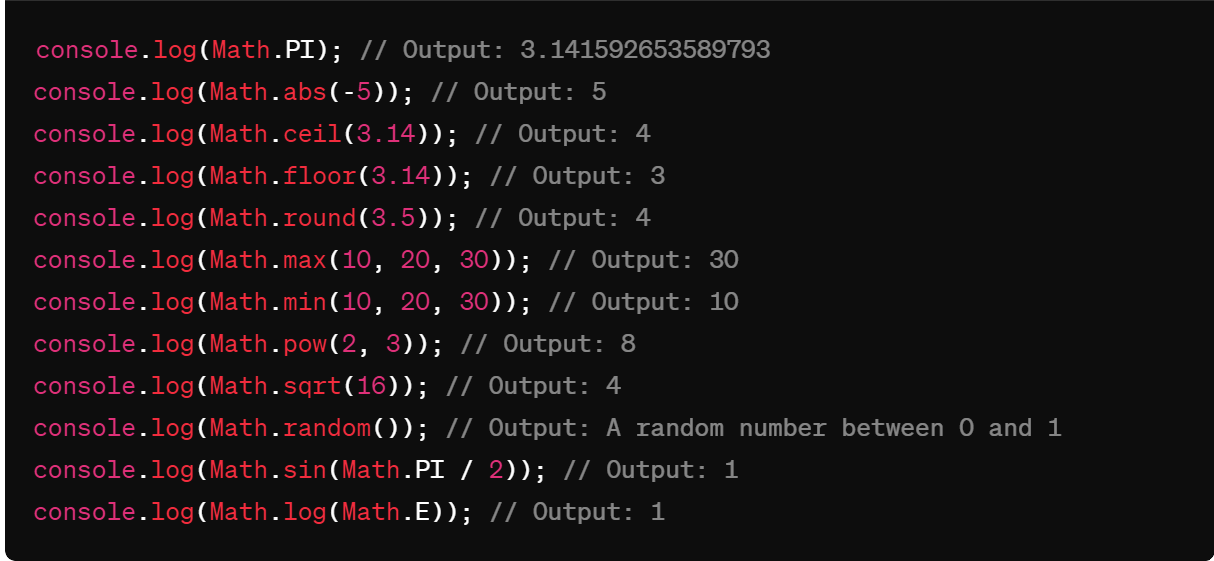
Math.pow(): Returns the base to the exponent power.

Math.sqrt(): Returns the square root of a number.

Math.random(): Returns a random floating-point number between 0 (inclusive) and 1 (exclusive).

Math.sin(), Math.cos(), Math.tan(): Trigonometric functions to calculate sine, cosine, and tangent of an angle (in radians), respectively.

Math.log(), Math.exp(): Exponential functions to calculate natural logarithm and exponential of a number, respectively.



**This keyword**

`this` is a keyword that refers to the current 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

**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

2.**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:

**Lexical Scope**: Arrow functions do not have their own this context. Instead, they inherit this from the surrounding (lexical) scope where they are defined.

**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).

**This refers to the particular event in event handlers**

  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.

**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' }

**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.

      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:

**Global Scope:** Variables declared outside of any function or block are in the global scope. They are accessible from anywhere in the code.

**Local Scope:** Variables declared within a function or block are in the local scope. They are only accessible within that function or block.

**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).

**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") ;

        })();

**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 message from "./message.js";

**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

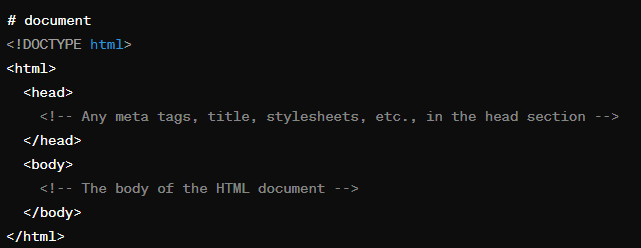
The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree. With them, you can change the document's structure, style, or content.

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

**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.



**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.

**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.

**Attributes**: Elements can have attributes like **id**, **class**, **src**, etc. These attributes are accessible and modifiable through the DOM.

**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

**Manipulating Elements**:

Changing element attributes (**element.attribute**).

Changing element content (**element.innerHTML**, **element.innerText**, **element.textContent**).

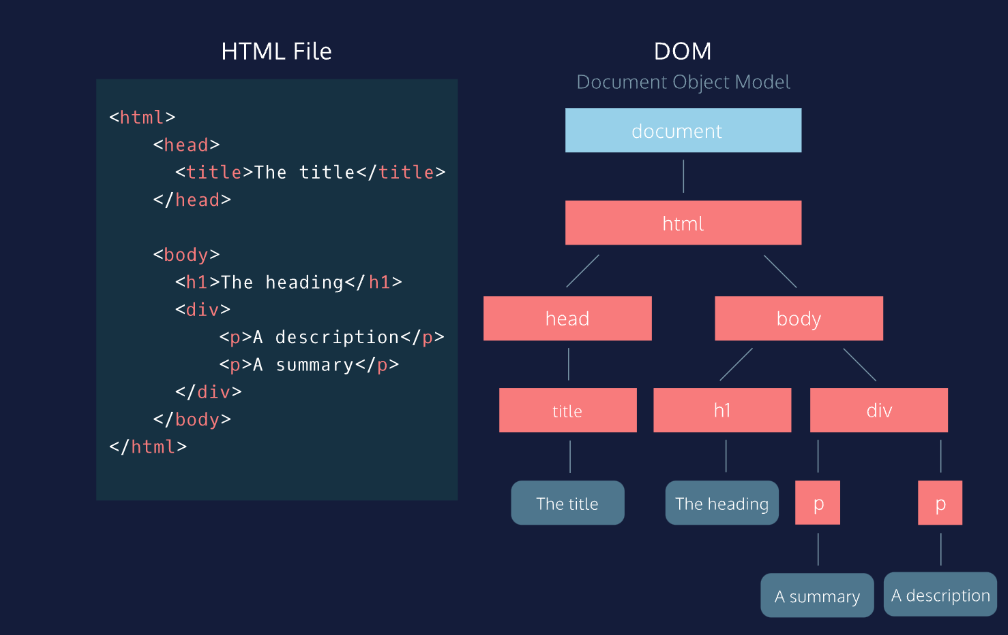
Adding or removing classes (**element.classList.add()**, **element.classList.remove()**).

Creating new elements (**document.createElement()**).

Appending or removing child nodes (**parentNode.appendChild()**, **parentNode.removeChild()**).

**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.

**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**.



**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  document.tables | Returns the <title> element  Returns the tables elements | 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>

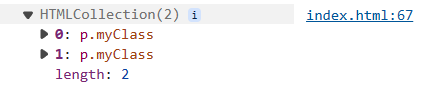
var elementsByClassName = document.getElementsByClassName("myClass");

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>

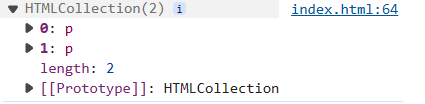
  var elementsByTagName = document.getElementsByTagName("p");

  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 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);

Access the element where you want to append the text node.

Create a text node using **document.createTextNode()**.

Append the text node to the desired element.

**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 change attribute values by using setAttribute**

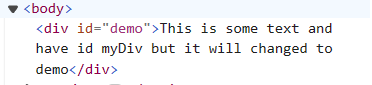
<div id="myDiv">This is some text and have id myDiv but it will changed to demo</div>

 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");

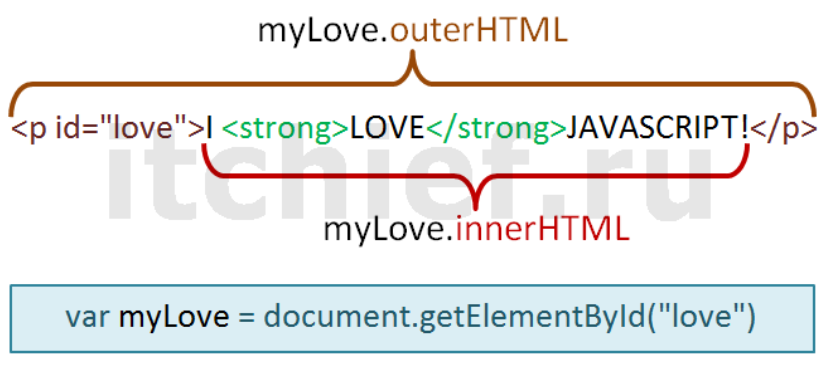
// To get the value of an attribute, such as "src" for an image element:

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

console.log(srcValue);

//output



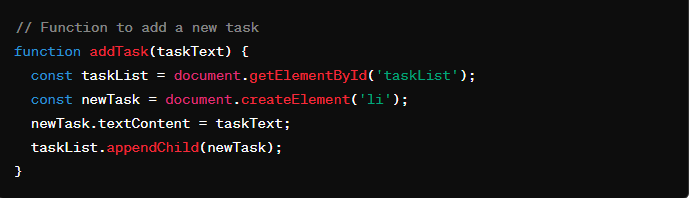


**===================================================**

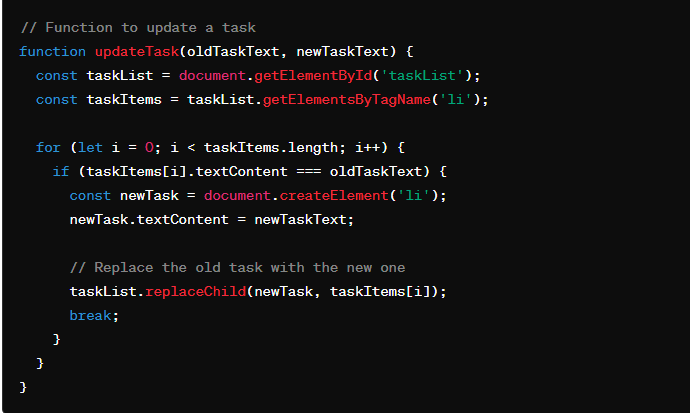
Reading the elements

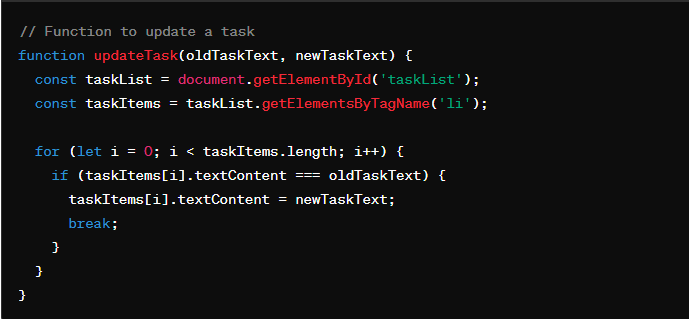


Create an element



Update the element





Delete the element



**Manipulating DOM Elements:**

- Accessing Properties: Changing properties like `innerHTML`, `textContent`, `innerText`, `value`, etc.

- Modifying Attributes: Changing attributes like `src`, `href`, `class`, `id`, etc.

- Adding and Removing Elements: Creating new elements with `createElement()`, appending with `appendChild()`, removing with `removeChild()`, etc.

- Modifying Styles: Changing CSS styles using `style` property or `classList`.



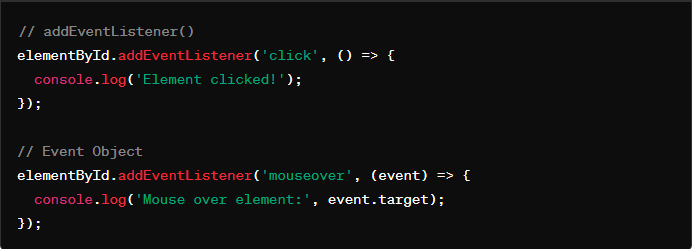
**Event Handling:**

- addEventListener(): Attaches an event handler to an element.

- Event Types: Various events like `click`, `mouseover`, `keydown`, etc.

- Event Object: Provides information about the event and its target element.

- Event Propagation: Understanding event capturing and bubbling phases.

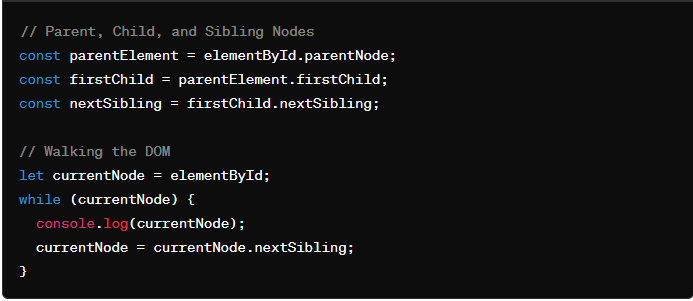


**DOM Traversal and Manipulation:**

- Parent, Child, and Sibling Nodes: Accessing and navigating through different parts of the DOM tree.

- Traversal Methods: `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`, etc.

- Walking the DOM: Techniques to iterate over DOM elements.



**Types of Events**

**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.

**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.

**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.

**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>

**More Explanation**

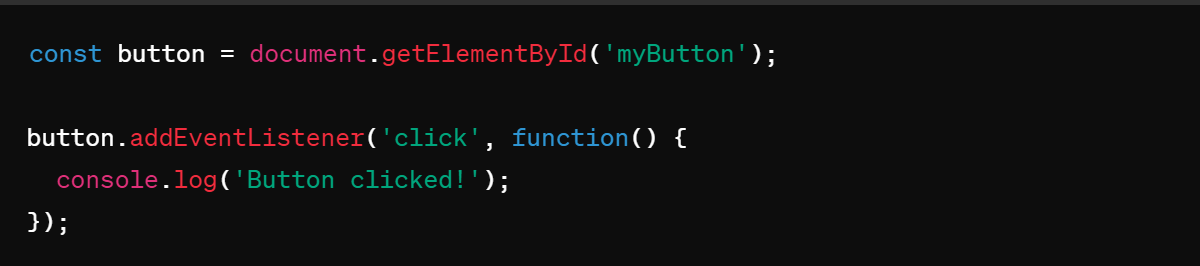
**Events and Event Handling:**

**1. Event Listener:**

- An event listener is a function that listens for a specific type of event on a given DOM element.

- It waits for an event to occur and then executes the specified callback function.

- Event listeners are added using the `addEventListener()` method.



**2. Event Handler:**

- An event handler is a function that is executed in response to an event being triggered.

- It can be defined inline as shown in the example above, or it can be a named function.

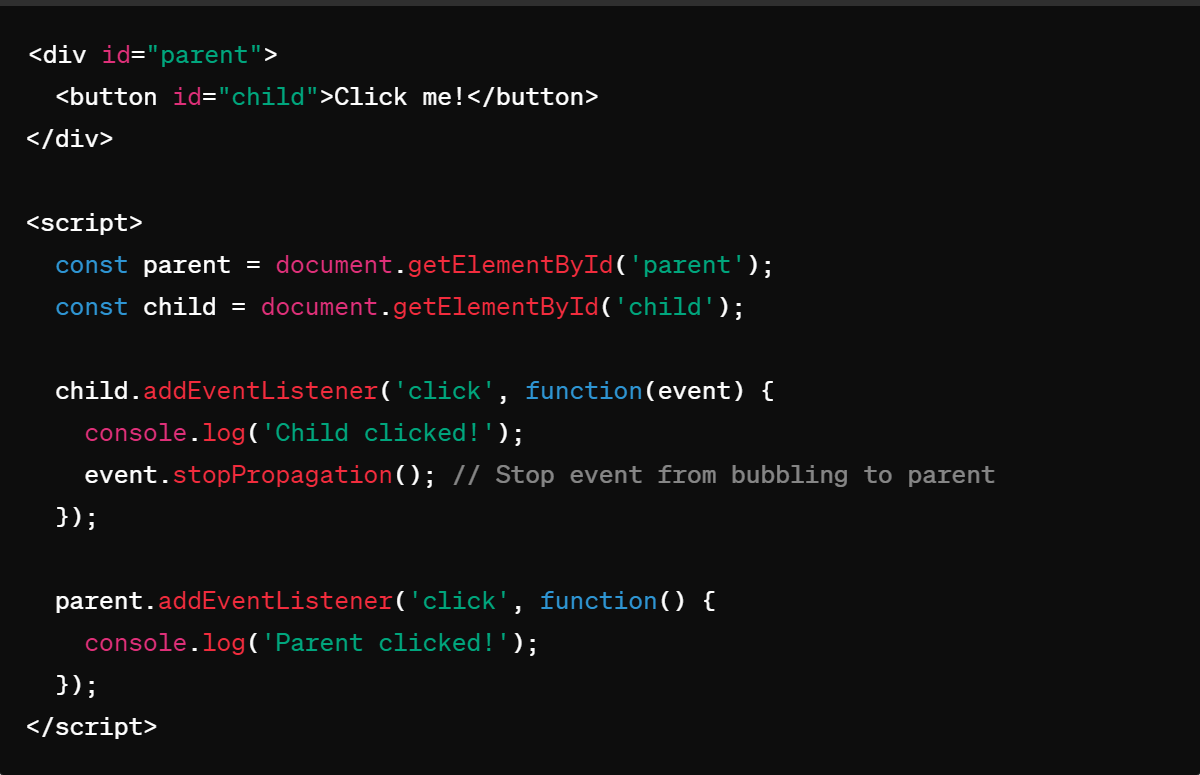


**3. Event Bubbling:**

- 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.

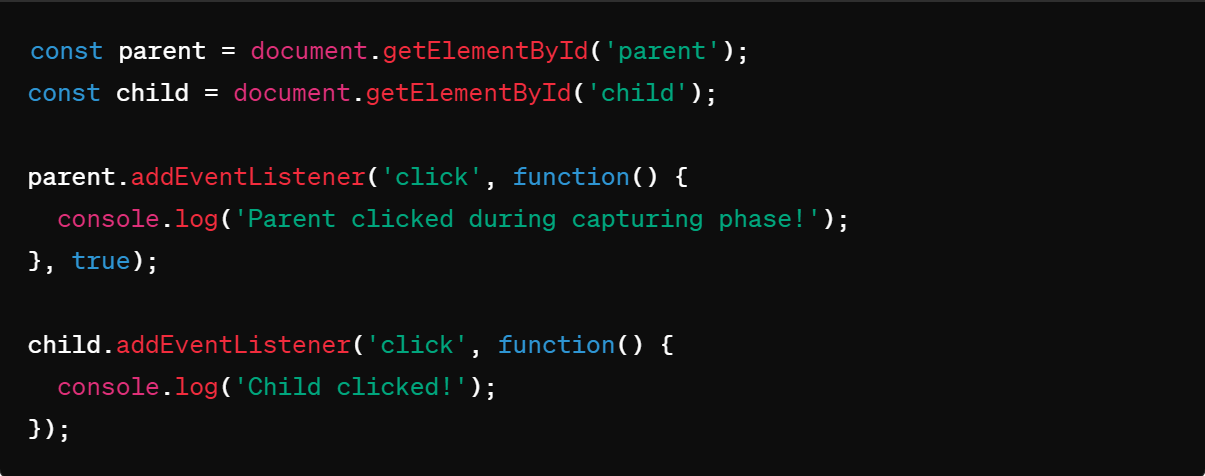
- You can stop the bubbling phase using `event.stopPropagation()`.



**4. 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()`.

**5. 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`.



**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.

**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.

**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"

**Methods**

console.log("Java Enabled:", navigator.javaEnabled());

// Example output: "Java Enabled: false" (Depends on the browser and user settings)

**Location objects**

It provides properties that allow you to access and manipulate different parts of the URL

// Properties

console.log("Full URL:", location.href);

// Example output: "Full URL:

https://www.example.com/path/?query=string"

console.log("Hostname and Port:", location.host);

// Example output: "Hostname and Port: www.example.com"

console.log("Hostname:", location.hostname);

// Example output: "Hostname: www.example.com"

console.log("Protocol:", location.protocol);

// Example output: "Protocol: https:"

console.log("Pathname:", location.pathname);

// Example output: "Pathname: /path/"

**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.

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);

**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)

**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** 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));

**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."

**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:**

We can destructure values individually

We can can skip elements in the array by using commas.

If the value at the specified position is undefined, you can assign a default value.

We can use rest operator (...) to collect the remaining elements into a new array.

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:**

We can destructure values individually

We can rename variables while destructuring by using a colon (:).

We can set default values for properties that might be undefined.

Destructuring can be used to extract values from nested objects.

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'

**Spread Syntax**:

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 and string

Spread operator is 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 = [...arr, 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]; // Shallow copy using the rest operator

shallowCopyArray[0] = 10; // Modify the shallow copy

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

console.log(shallowCopyArray); // Output: [10, 2, 3, 4, 5]

**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

**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 can be used in array/ object destructuring

  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)**

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](theme/theme1.xml)

         \* 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.# )

}

   }

**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);

Certainly! 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**

**Inheritance in JavaScript**:

**Base Class**: Defines common properties and methods.

**Subclass**: Inherits from the base class and can add or override properties and methods.

**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.

**Constructor Rules**:

In a subclass constructor, super must be called before accessing this.

Ensures proper initialization of the inherited fields.

**Method Inheritance**:

Subclasses inherit all methods from the base class.

Methods can be overridden in the subclass if needed.

**Code Reusability**:

Inheritance promotes code reusability by allowing subclasses to use and extend the functionality of base classes.

Reduces redundancy and simplifies maintenance.

**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.

**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 snippet, "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**

**Event Loop**: The event loop is a mechanism in JavaScript that continuously checks the call stack and the callback queue. It ensures that the execution of code is done in the right order, especially when dealing with asynchronous operations.

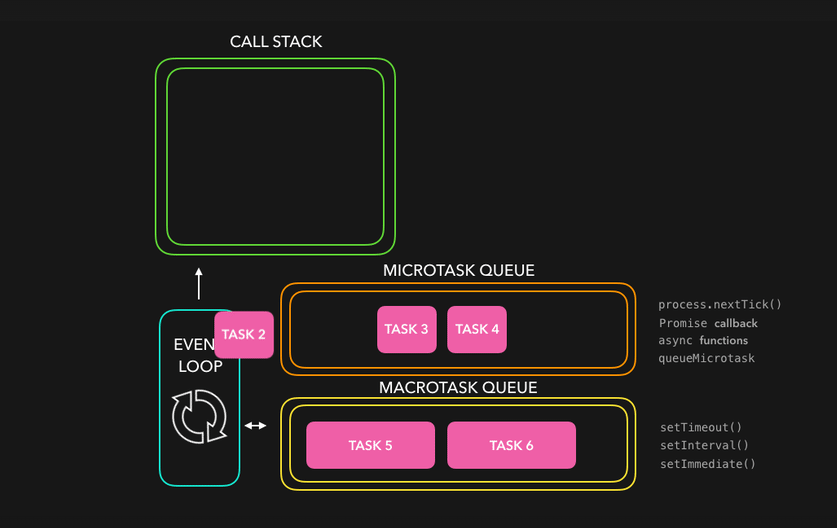
**Call Stack:** The call stack is a data structure that keeps track of function calls in the code. Whenever a function is called, it's added to the top of the call stack. When a function completes, it's removed from the stack.

**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.

**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



**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.

**Callbacks**

callback is a function that 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!");

      });

**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

Asynchronous Operations

Higher-Order Functions

Array Methods

**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 (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 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

**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);

            });

          });

        });

      });

To mitigate callback hell, several approaches have been developed:

Named functions

Promises

Async/await

**Promises**

Promises in JavaScript provide a cleaner and more structured way to handle asynchronous operations compared to traditional callbacks.It has three states: pending, fulfilled, or rejected.

**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")

      }

      });

**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

      let add = (val) =>

        new Promise((resolve, reject) => {

          resolve(val + 10);

        });

      let sub = (val) =>

        new Promise((resolve, reject) => {

          resolve(val - 10);

        });

      let mul = (val) =>

        new Promise((resolve, reject) => {

          resolve(val \* 5);

        });

      let div = (val) =>

        new Promise((resolve, reject) => {

          resolve(val / 2);

        });

      add(10)

        .then((addres) => sub(addres))

        .then((subres) => mul(subres))

        .then((mulres) => div(mulres))

        .then((divres) => console.log(divres))

        .catch((error) => console.error(error));

Promises aysnchronous

        let promise1= new Promise((resolve,reject)=>{

          console.log("promise 1");

          setTimeout(resolve, 2000, "promise 1 success")

        })

        let promise2= new Promise((resolve,reject)=>{

          console.log("promise 2");

          setTimeout(resolve, 1500, "promise 2 success")

        })

        let promise3= new Promise((resolve,reject)=>{

          console.log("promise 3");

          setTimeout(resolve, 1800, "promise 3 success")

        })

        let promise4= new Promise((resolve,reject)=>{

          console.log("promise 4");

          setTimeout(resolve, 500, "promise 4 success")

        })

        promise1.then((resolve)=>{console.log(resolve)})

        promise2.then((resolve)=>{console.log(resolve)})

        promise3.then((resolve)=>{console.log(resolve)})

        promise4.then((resolve)=>{console.log(resolve)})

//convert synchronous to aynchronous

      promise1

        .then((result) => {

          console.log(result);

          return promise2;

        })

        .then((result) => {

          console.log(result);

          return promise3;

        })

        .then((result) => {

          console.log(result);

          return promise4;

        })

        .then((result) => {

          console.log(result);

        });

**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 163 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. This block is often used for cleanup operations.

//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 = window.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.

**Prompting the User**:

num = window.prompt("entry");

**Error Handling with try...catch:**

try {

if (num <= 0) {

throw new Error("The number must be positive.");

}

} catch (err) {

console.error(err);

}

**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

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("%");

**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**.

**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.

**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:

**200 OK**: This status code indicates that the request was successful, and the server has returned the requested resource.

**201 Created**: Indicates that the request was successful, and a new resource has been created as a result.

**204 No Content**: The server successfully processed the request, but there is no content to return.

**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.

**401 Unauthorized**: Indicates that the client needs to authenticate itself to access the requested resource.

**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).

**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.

**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.

**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.

**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.

**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);

        });

**fetch("https://fakestoreapi.com/products")**: This line initiates a request to the specified URL, which returns a Promise representing the response to that request.

**.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.

**.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()**

**How to create a local json server**

Download and Install Node js

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**

**Get-ExecutionPolicy**

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

Install json server for api creation in local server

**npm install -g json-server@0**

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**

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);

        })

**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**")

**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.

**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

      }

**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();

**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);

        })