JAVA SCRIPT

**Date:  11-09-2024**

**Day : wednesday  
Topic:  Javascript intro**

**Javascript**

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

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

.Lightweight

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

Cross-Platform

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

Single-Threaded

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

High-Level and Easy to Write

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

Interpreted Compiled

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

Scripting Language for Webpages

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

**The History of JavaScript**

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

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

**2. Early Naming and Initial Implementation**

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

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

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

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

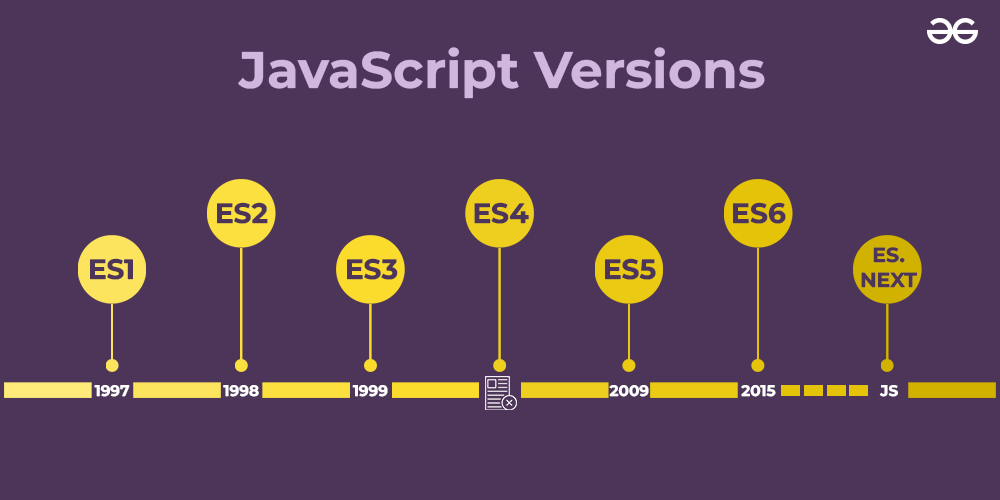
* **Launch of JScript:** In response to JavaScript’s growing popularity, Microsoft developed its own implementation of JavaScript called *JScript* in 1996.
* **Compatibility:** JScript was designed to be compatible with JavaScript but included certain proprietary features and enhancements. Microsoft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

**5. Standardization: ECMAScript**

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

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

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



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

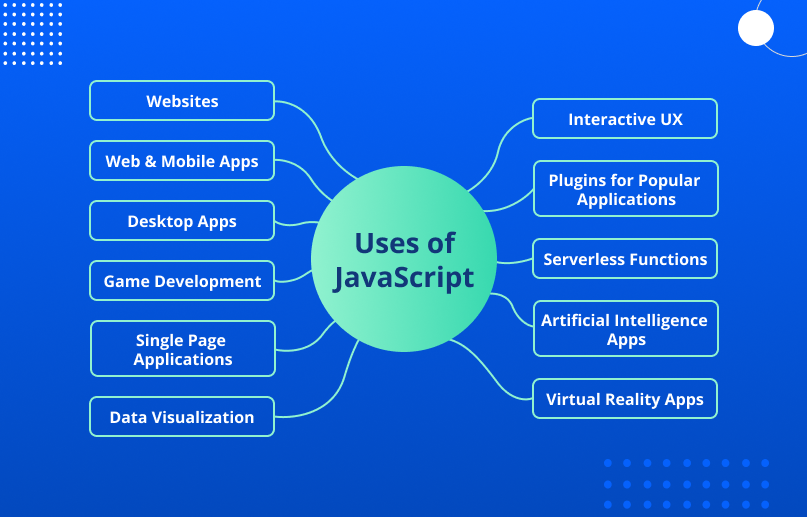
* **Annual Release Cycle:** Since ECMAScript 2015, the TC39 committee, which oversees the evolution of ECMAScript, adopted an annual release cycle. Each year, a new version of ECMAScript is released with incremental updates and new features.
* **ES7 (ES2016), ES8 (ES2017), ES9 (ES2018), etc.:** These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

**8. JavaScript's Role Today**

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

Some of the common uses of JavaScript are:

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

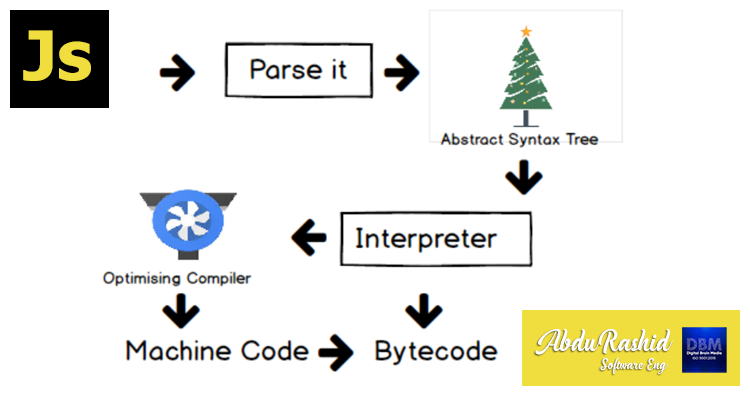




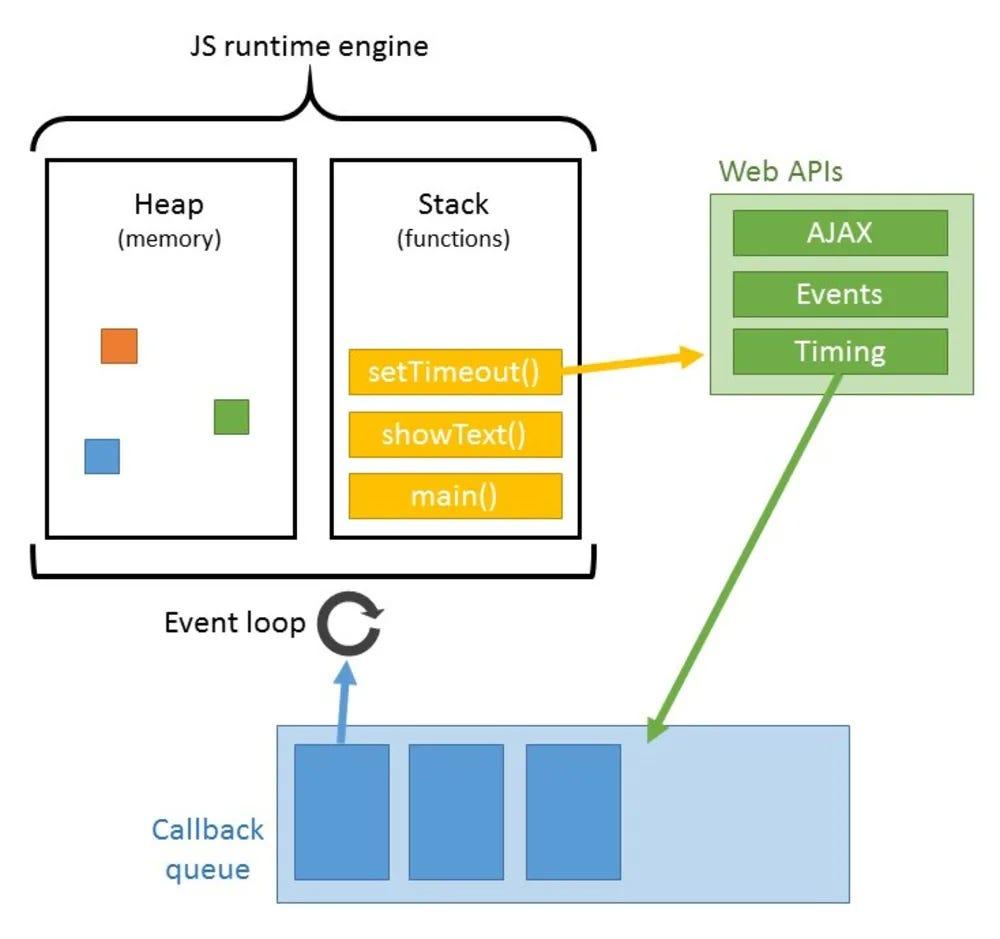
**How JavaScript Works: JS Engine & Runtime environment**

1. JavaScript in the Browser:
   * When you visit a webpage, your browser (like Chrome, Firefox, or Safari) loads the HTML and CSS to structure and style the page.
   * If the page includes JavaScript (either directly in the HTML file or linked as a separate .js file), the browser loads and runs that JavaScript code using its built-in JavaScript engine (like V8 in Chrome or SpiderMonkey in Firefox).
2. JavaScript Engine:
   * What It Is: A JavaScript engine is a program inside the browser that reads and executes JavaScript code. Every major web browser has a built-in JavaScript engine.
   * Examples: V8 (Chrome, Node.js), SpiderMonkey (Firefox), Chakra (Edge), and JavaScriptCore (Safari).
3. Execution Process:

* Parsing: The JavaScript engine first reads (or "parses") the JavaScript code. It checks the syntax to make sure everything is written correctly.
* Compilation: Unlike some languages that are compiled ahead of time (like C++), JavaScript is often compiled just before or even during its execution, in a process called Just-In-Time (JIT) compilation.
* Execution: After parsing and compiling, the engine runs the JavaScript code. It turns the code into machine-readable instructions that the computer's processor can understand and execute.



1. Event Loop and Asynchronous Behavior:
   * Single-Threaded: JavaScript runs in a single thread, meaning it can only do one thing at a time.
   * Event Loop: To handle multiple tasks (like user inputs, timers, or fetching data from a server), JavaScript uses an event loop. The event loop constantly checks if there are tasks (events) to be executed. If there are, it runs them one by one.
   * Asynchronous Tasks: JavaScript can perform certain tasks asynchronously, meaning it can start them, then continue with other work, and come back to finish the task once it’s done. Examples include making network requests, reading files, or waiting for user interactions.



1. Interacting with the Webpage (DOM):
   * DOM (Document Object Model): The DOM is a structured representation of the HTML elements on a webpage. JavaScript can interact with the DOM to change the content, structure, and style of the webpage dynamically.
   * How It Works: JavaScript can select elements on the page (like buttons or input fields), change their properties (like text, color, or visibility), and respond to events (like clicks or form submissions).

**Date:  12-09-2024**

**Day : Thursday  
Topic:  How to insert js, Variables and datatypes**

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

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

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

 <script>

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

 </script>

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

<!DOCTYPE html>

<html>

<head>

    <title>Inline JavaScript Example</title>

</head>

<body>

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

</body>

</html>

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

it allows you to reuse the script in multiple HTML pages

<!DOCTYPE html>

<html>

<head>

    <title>External JavaScript Example</title>

</head>

<body>

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

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

</body>

</html>

**Variables**

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

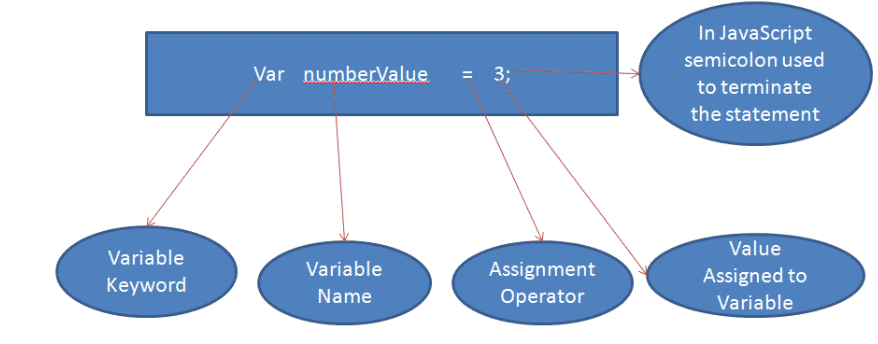
1. **Declaration**: Variables in JavaScript are declared using the **var**, **let**, or **const** keyword.
   * **var** has been traditionally used but has some scope-related issues.
   * **let** is block-scoped and is preferable for variable declaration.
   * **const** is also block-scoped but its value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

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

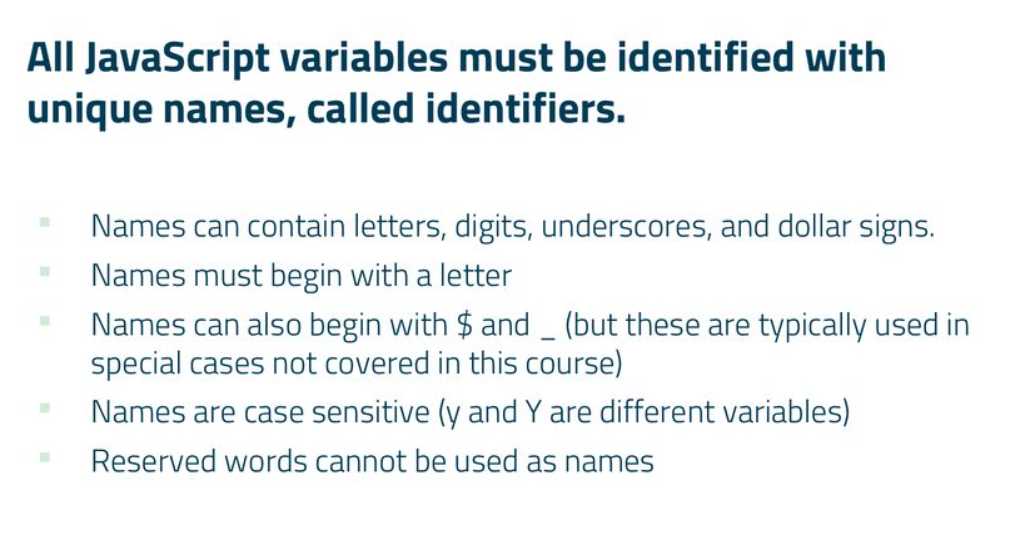
      var x = 10;

      let y = 20;

      const PI = 3.14;



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



      var myVariable = 5;

      var \_myVariable = 10;

      var $myVariable = 15;

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

      var myVariable = 5;

      var MyVariable = 10;

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

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

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

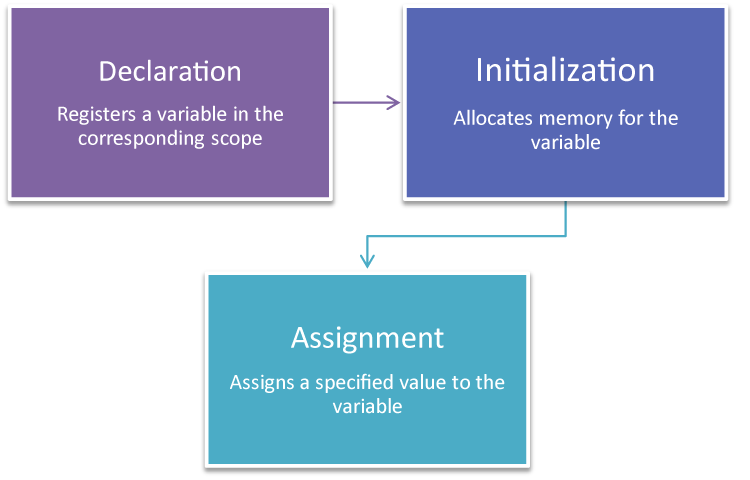
      var num = 5;

      var str = "Hello";

      var arr = [1, 2, 3];

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

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



**1. Declaration**

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

let myVariable;

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

**2. Initialization**

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

let myVariable = 10;

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

**3. Assignment**

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

let myVariable;

myVariable = 10;  // Assignment after declaration

myVariable = 20;  // Re-assignment

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

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

      myVar=5;

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

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

**Hoisting**

**In general terms Hoisting means ?**

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

**Technically ?**

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

Only the declarations are hoisted, not the initializations.

**2) var Declarations**:

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

•  The variable is hoisted and initialized with undefined

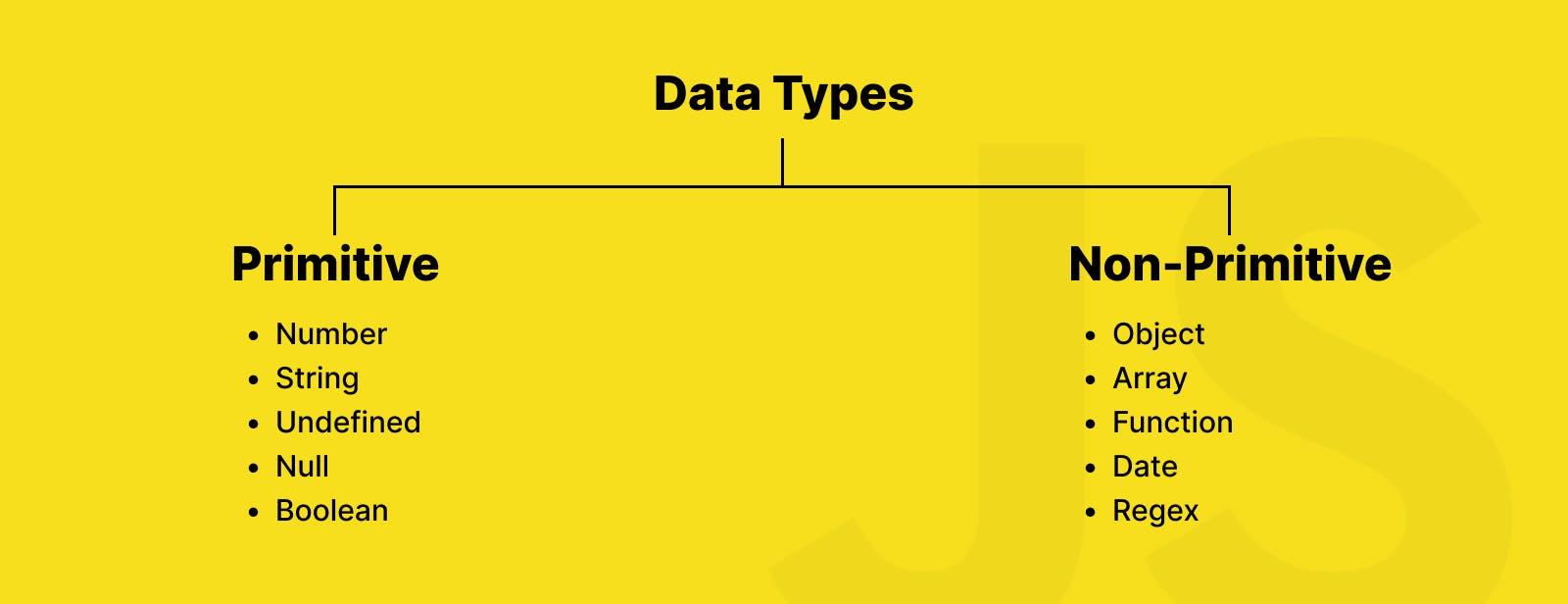
console.log(myVar);   // Output: undefined

var myVar = 10;

console.log(myVar);   // Output: 10

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

**Data types**



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

        let integer = 42;          // Integer

        let float = 3.14;          // Floating-point number

        let negative = -7;         // Negative number

        let infinity = Infinity;   // Infinity

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

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

        let singleQuoteString = 'Hello, world!';

        let doubleQuoteString = "Hello, world!";

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

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

        let isJavaScriptFun = true;

        let isTired = false;

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

    let notAssigned; // `notAssigned` is undefined

        console.log(notAssigned); // undefined

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

let emptyValue = null;

1. Composite Data Types or Non primitive:Non-primitive data types, also known as reference types, are more complex data structures that can hold multiple values and have methods and properties.non-primitive data types are mutable, meaning their values can be changed after they are created, and they are stored and accessed by reference rather than by value.

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

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

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

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

       let person = {

        name: 'John',

        age: 30,

        isEmployed: true

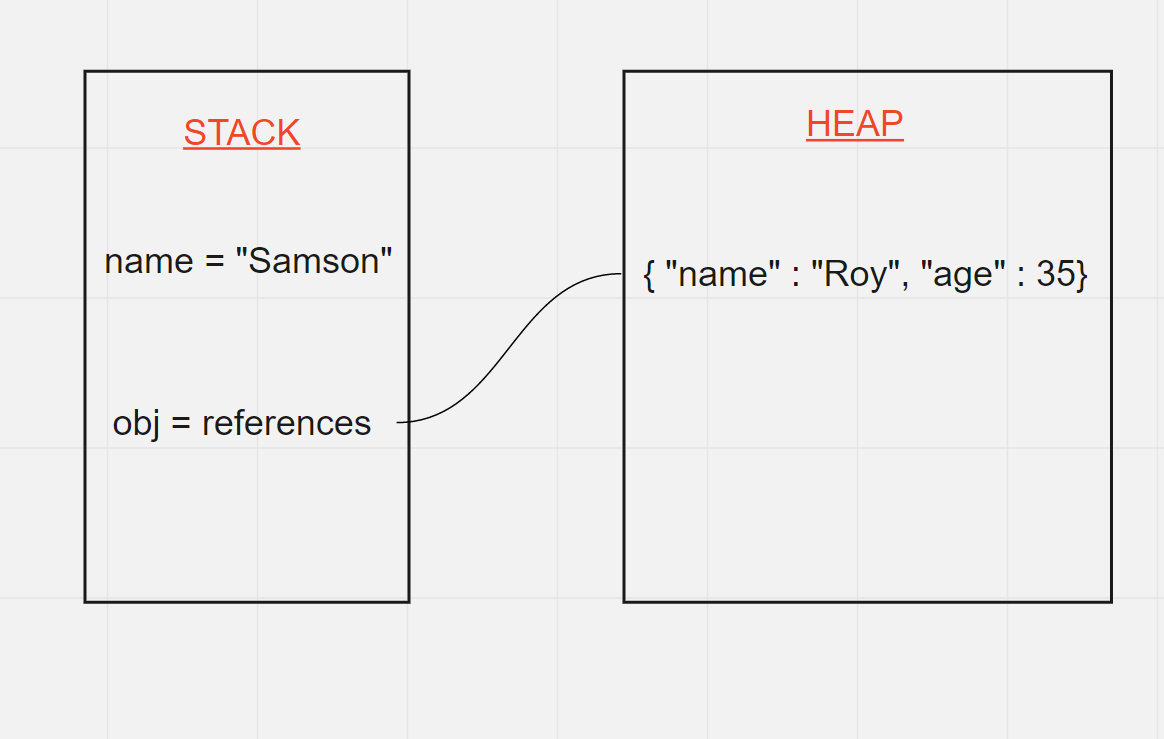
        };

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

    let sayHello = function(name) {

        return `Hello, ${name}!`;

    };



**Null & Undefined**

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

**Date:  13-09-2024**

**Day : Thursday  
Topic:  Type conversion and truthy , fasly values**

**Type Coercion**

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

**Types of Type Coercion**

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

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

**Explicit Type Conversion**

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

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

let num = 123;

let str = String(num);

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

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

let str = "123";

let num = Number(str);

 console.log(num); // Output: 123

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

let num = 0;

let bool = Boolean(num);

 console.log(bool); // Output: false  
**In JavaScript, values are categorized as either "truthy" or "falsy"**

**Falsy Values:**

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

**Truthy Values:**

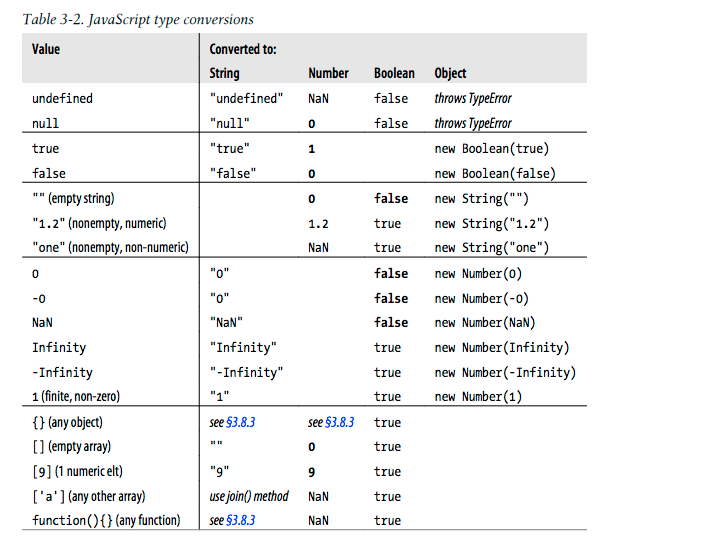
1. **true**: The boolean value true itself.
2. **Non-zero numbers**: Any number other than 0 (including negative numbers and decimals).
3. **Non-empty strings**: Any string with at least one character.
4. **Non-empty arrays**: Arrays with at least one element.
5. **Objects**: Any object (including functions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

      //program to find falsy and truthy values

      var a = [];

      var b = a ? true : false;

      console.log(b);



**Date:  16-09-2024**

**Day : Monday  
Topic:  Operators**

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

      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 and**

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

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

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

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

**//logical or**

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

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

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

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

* **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";

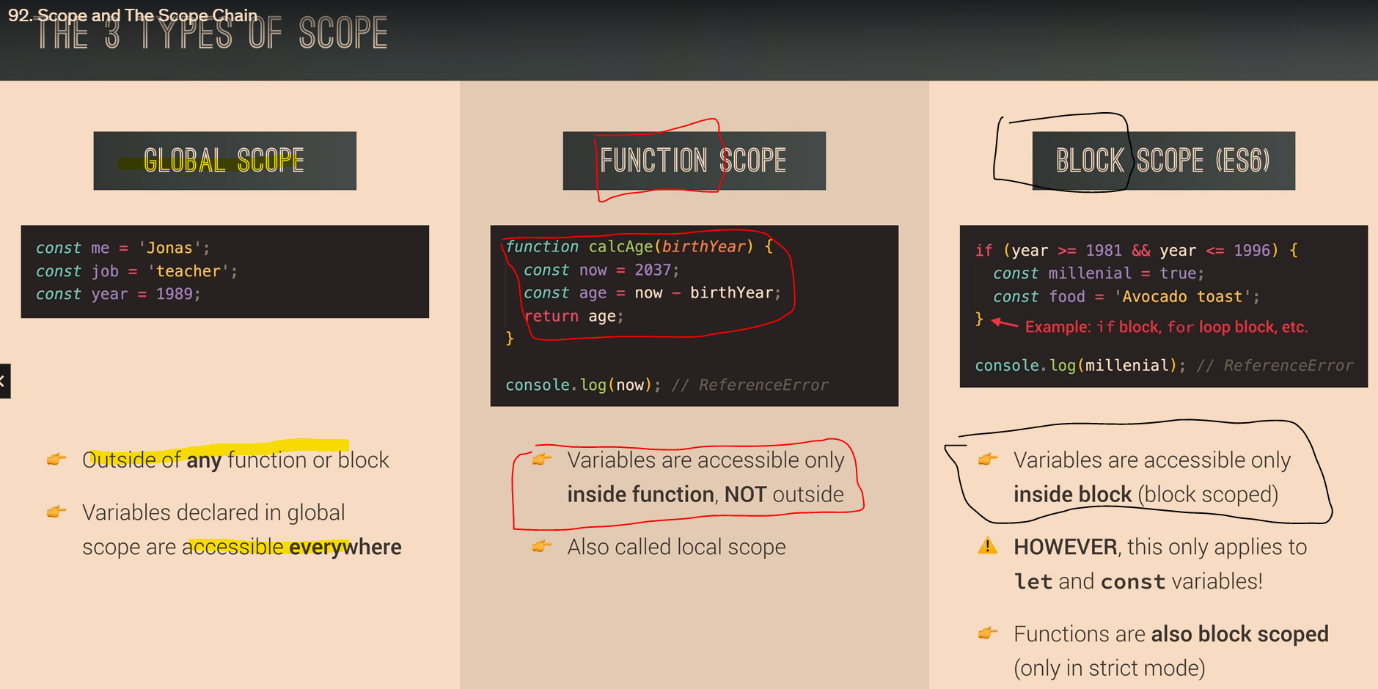
      console.log(status); // Output: 'Adult'

**Date:  18-09-2024**

**Day : Tuesday  
Topic:  Scopes and var , let , const difference**

**Scopes in javascript**:

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



**Global Scope**:

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

{

  var x = 2;

}

// x can be used here

**Block Scope**:

Variables declared within a block cannot be accessed from out

side of the block.

{

  let x = 2;

}

// x can NOT be used here

**Local Scope**:

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

function myFunction() {

  var a = "skills";

  // code here CAN use a

}

// code here can NOT use a

**VAR LET CONST**

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

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

{

  var x = 2;

}

// x CAN be used here

//because it has a global scope

2) variable declared with var can be redeclared

var x = "front end";

var x = 0;

//Variables defined with var can be redeclared.

3) variable declared with var can be re-assigned

var x;

x = 0;

x=10;

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

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

{

  let x = 2;

}

// x can NOT be used here

2) Variables defined with let can not be redeclared.

let x = "John Doe";

let x = 0;

//output – identifier has already been declared

3) Variable defined with let can be reassigned.

let x;

x = 0;

x=20;

//output- 20

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

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

{

 const x = 2;

}

// x can NOT be used here

2) Variables defined with const can not be redeclared.

Const x = "coding”;

Const x = 0;

output – identifier has already been declared

3) Variable defined with const cannot be reassigned.

Const x;

x = 0;

x=20;

//output-missing initializer in const declaration

**Date:  19-09-2024**

**Day : Thursday  
Topic:  Conditional statements and switch case**

**Conditional statements**

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

**1. if Statement:**

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

Syntax:

**if (condition) {**

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

**}**

      let x = 10;

      if (x > 0) {

        console.log("x is positive");

      }

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

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

**if (condition) {**

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

**} else {**

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

**}**

      let x = -5;

      if (x > 0) {

        console.log("x is positive");

      } else {

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

      }

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

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

**if (condition1) {**

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

**} else if (condition2) {**

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

**} else {**

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

**}**

      let x = -5;

      if (x > 0) {

        console.log("x is positive");

      } else if (x < 0) {

        console.log("x is negative");

      } else {

        console.log("x is zero");

      }

**Switch statements**

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

**switch (expression) {**

**case value1:**

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

**break;**

**case value2:**

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

**break;**

**// More cases...**

**default:**

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

**}**

**Key Points**

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

switch (grade) {

  case 'A': console.log('Excellent');

    break;

  case 'B':

  case 'C':console.log('Well done');

    break;

  case 'D':console.log('You passed');

 break;

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

    break;

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

}

**Date:  20-09-2024**

**Day : Friday  
Topic:  for loop**

**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);

      }

**Date:  23-09-2024**

**Day : Monday  
Topic:  Nested loop and continue & break**

**What are Nested Loops?**

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

**2. Basic Structure**

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

**JavaScript**

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

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

        // Code to execute

    }

}

**3. Multiplication Table**

Using nested loops to create a multiplication table:

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

    let row = "";

    for (let j = 1; j <= 10; j++) {

        row += (i \* j) + "\t";

    }

    console.log(row);

}

**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.

**Date:  24-9-2024**

**Day : Tuesday  
Topic:  while , do while , For in loop and for of loop**

* **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,array, string. It iterates over enumerable properties of an object, in an arbitrary order.

Syntax:

      for (ref in strname){

        console.log(ref);//indexes

      }

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

      // Iterate over the properties of an object

      const person = {

        name: "John",

        age: 30,

        gender: "male",

      };

      for (let prop in person) {

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

      }

1)  **Iterates over Properties**:

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

 2) **Order Not Guaranteed**:

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

3)  **Use with Objects**:

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

Syntax:

      for (ref of strname){

        console.log(ref);//values

      }

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

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

      // Iterate over elements of an array

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

      for (let num of numbers) {

        console.log(num);

      }

1)   **Iterates over Values**:

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

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

* Commonly used with arrays, strings, maps, sets, and other iterable objects.

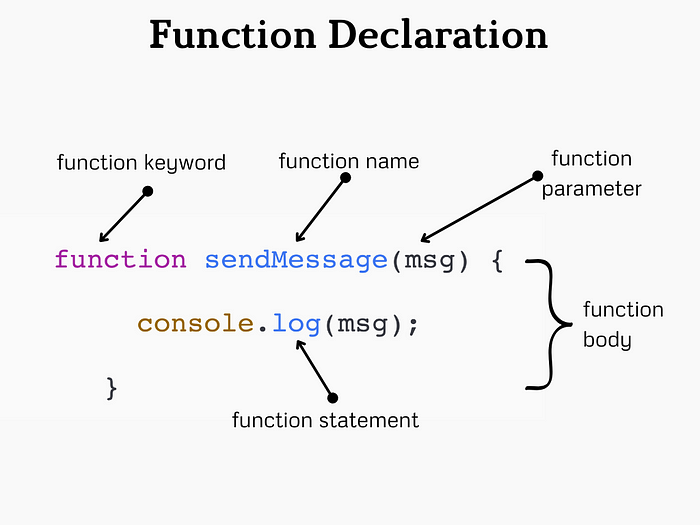
**Date:  25-9-2024**

**Day : wednesday  
Topic:  Functions**

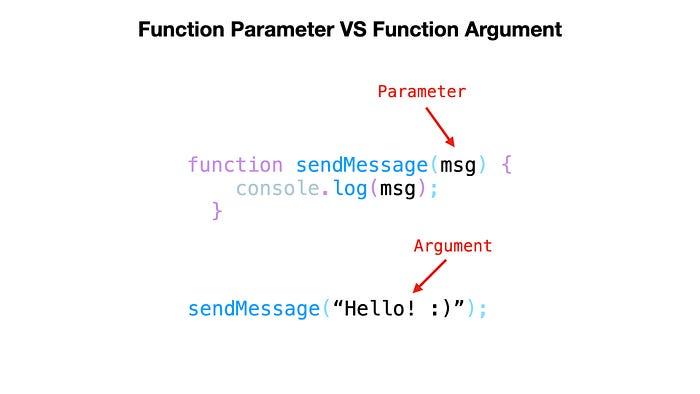
**Function:-**

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

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



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



Named functions can be hoisted

Ex 1:

        function myfun(a){

            return console.log(a);

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

        }

        myfun(20);

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

Points needs to be noted in functions

1) function name  stores function definition

2) log off  the  function calling stores return value

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

4) Named function can only hoisted

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

**Anonymous function:-**

  Anonymous function is a function that is defined without a name

*var* anonfun=function(){

            return "this is anonymous function";

        }

        console.log(anonfun());

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

**Arrow function:-**

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

        var arrowfun=()=>{

            return "this is arrow function";

        }

        console.log(arrowfun());

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

**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") ;

        })();

**Date:  26-9-2024**

**Day : Thursday  
Topic:  Functions**

**Callback Function**

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

        function hello1(){

            return "hello1 function is triggered by main function";

        }

        function hello2(j){

            var a=j();

         return a;

        }

        console.log(hello2(hello1));

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

**Global execution context:**

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

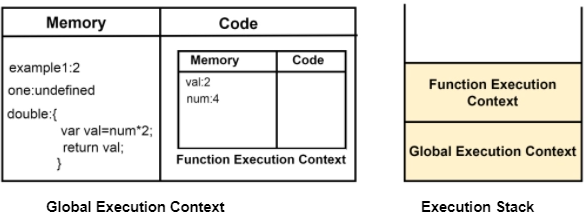
**Key Components**

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

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

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

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



**How it works on the functions**

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

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

**Closures**

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

function ahello() {

        var a = "variable inside a outer function";

        function ahi() {

          var b = "varaible inside inner function";

          console.log(a);

          console.log(b);

        }

        ahi();

      }

      ahello();

**Scope Chaining in JavaScript**

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

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

**Scope Chain**:

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

  var globalVar = "I am global";

function outerFunction() {

    var outerVar = "I am outer";

    function innerFunction() {

        var innerVar = "I am inner";

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

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

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

    }

    innerFunction();

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

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

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

}

outerFunction();

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

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

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

**Lexical Scoping:**

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

**Date:  30-9-2024**

**Day :  Monday  
Topic:  Closures, default parameters and intro to the dom**

**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();

**Default parameters**

Function with default parameters

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

            return a+b;

        }

        console.log(hello());

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

**DOM- Document Object Model**

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

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

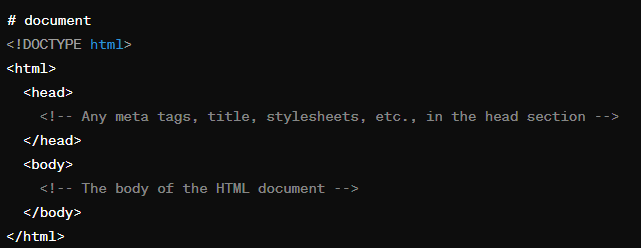
The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree. With them, you can change the document's structure, style, or content.

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

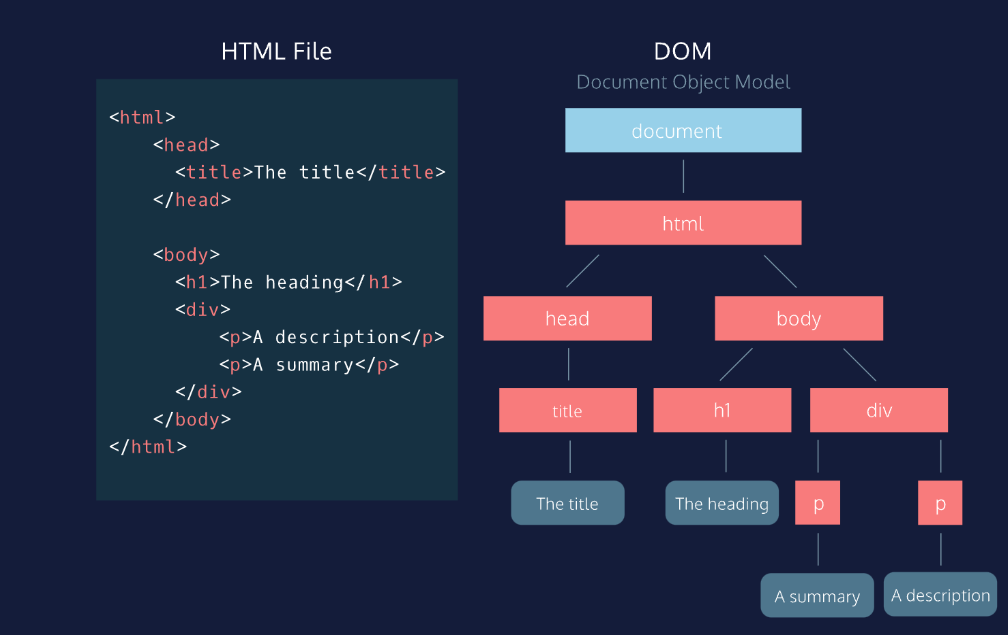
1. **Document**: The top-level object in the DOM hierarchy, representing the entire HTML document. It serves as an entry point to access and manipulate the document's content.

 console.log(document);

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



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

 **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

**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>

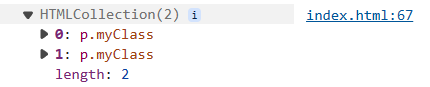
var elementsByClassName = document.getElementsByClassName("myClass");

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>

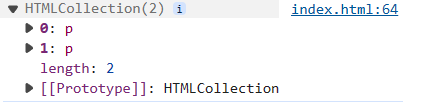
  var elementsByTagName = document.getElementsByTagName("p");

  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!";

**Date:  1-10-2024**

**Day :  Tuesday  
Topic:  dom styles, create element and append**

**How to apply styles using dom**

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

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

// Step 2: Apply styles

myDiv.style.backgroundColor="red";

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

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

    // Create a new paragraph element

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

    // Set innertext or other properties if needed

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

    // Append the paragraph to the document body

    document.body.appendChild(newParagraph);

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

**Appendchild and Append**

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

**appendChild**

**syntax :**

parentNode.appendChild(newChild);

**Parameters**:

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

**Behavior:**

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

Only accepts a single node.

**Append**

**syntax :**

parentNode.append(node1, node2, node3);

**Parameters**:

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

**Behavior:**

Can append multiple nodes and/or strings at once.

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

Allows appending a combination of nodes and text.

**How to create textNode**

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

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

// Step 2: Create a text node

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

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

myDiv.appendChild(textNode);

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

**DOM**

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

1) Field Names – document level object creation

2) Methods – element level object creation

**Create dom object by field names**

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

**Date:  3-10-2024**

**Day :  Thurs  day  
Topic: get attribute , set attribute, classlist and dom events and event handlers**

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

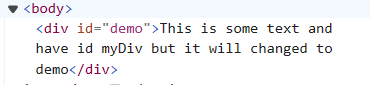
<div id="myDiv">This is some text and have id myDiv but it will changed to demo</div>

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

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

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

//output

****

**How to get attribute**

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

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

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

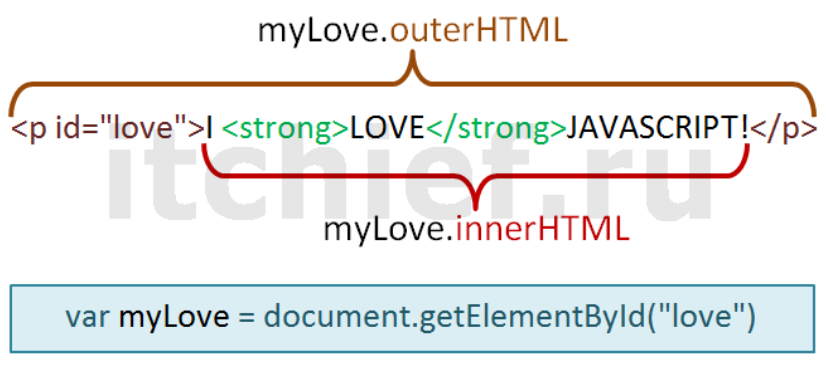
// To get the value of an attribute, such as "src" for an image element:

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

console.log(srcValue);

//output





 Classlist add and remove

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

**1. What is classList?**

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

**2. Syntax**

* The syntax for accessing classList is as follows:

element.classList

**classList methods include:**

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

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

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

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

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

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

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

**5. Common Use Cases**

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

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

element.classList.toggle('className');

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

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

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

**Types of Events**

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

**Event handlers:**

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

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

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

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

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

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

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

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

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

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

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

    console.log('Button clicked!');

});

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

function handleClick() {

    console.log('Button clicked!');

}

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

button.addEventListener('click', handleClick);

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

button.removeEventListener('click', handleClick);

**Date:  4-10-2024**

**Day :  Friday  
Topic: Form handling**

**Form Handling**

**1. Accessing Form Elements:**

    <form id="myForm">

        <input type="text" id="username">

        <input type="email" id="email">

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

    </form>

    <script>

        const form = document.getElementById('myForm');

        const usernameInput = document.getElementById('username');

        const emailInput = document.getElementById('email');

    </script>

**2. Setting Input Values:**

You can set input values using the value property:

    usernameInput.value = "John Doe";

    emailInput.value = "john@example.com";

**3. Getting Form Data:**

You can get form data when the form is submitted:

    form.addEventListener('submit', function(event) {

    event.preventDefault(); // Prevent form submission

    const formData = new FormData(form);

    // Access form data

    const username = formData.get('username');

    const email = formData.get('email');

    // Do something with the data

   });

**4. Changing Input Field Values**

You can change the value of input fields dynamically based on user actions or other events:

<input type="text" id="myInput">

<button onclick="changeInputValue()">Change Value</button>

<script>

    function changeInputValue() {

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

        input.value = 'New Value';

    }

</script>

**5.Accessing through values**

    document.forms[0].addEventListener('submit', function(event){

    event.preventDefault(); // Prevent form submission

    console.log(event);//stores the lot of form info including child tags of form

    console.log(event.target[0].value);//prints the value of first child in the first form

    })

In above program document.forms[0] get the first form,

e.preventdefault prevents the form default submission

console.log(event)  prints the lot of form data and event.target gets the childrens of the form in collection format.

**Date:  17-10-2024**

**Day :  Thursday  
Topic: String methods**

**Strings**

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

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

     //iterate the string using for loop

      var a ="hello";

      for(i=0;i<5;i++){

        console.log(a[i]);

      }

**String.length**

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

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

let str = "Hello, world!";

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

**charAt()**

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

let str = "Hello";

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

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

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

**at()**

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

let str = "Hello";

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

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

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

**charCodeAt()**

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

let str = "Hello";

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

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

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

**slice(start,end)**

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

let str = "Hello, world!";

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

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

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

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

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

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

**substring(start, end)**

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

let str = "Hello, world!";

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

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

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

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

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

**substr()**

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

let str = "Hello, world!";

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

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

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

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

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

**toUpperCase()**

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

let str = "Hello, world!";

let a= str.toUpperCase();

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

**toLowerCase()**

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

let str = "Hello, WORLD!";

let a = str.toLowerCase();

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

**concat()**

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

let str1 = "Hello";

let str2 = "world";

let str3 = "!";

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

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

**trim()**

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

let str = "   Hello, world!   ";

let trimmedStr = str.trim();

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

**repeat()**

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

let str = "Hello";

let repeatedStr = str.repeat(3);

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

**Split()**

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

let str = "Hello, world!";

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

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

let characters = str.split("");

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

**replace()**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**replaceAll()**

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

**string search methods**

**indexOf() and lastIndexOf**

    let str= "Hellow, world!";

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

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

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

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

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

**search()**

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

let str = "Hello, world!";

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

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

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

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

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

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

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

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

**includes()**

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

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

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

**Template literals**

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

    let a = 5;

    let b = 10;

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

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

**Date:  18-10-2024**

**Day :  Friday  
Topic: Array and array methods**

**Arrays**

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

**Features of array**

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

**Creating Arrays:**

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

let arr = [1, 2, “html”, “Css”];

**Accessing Elements:**

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

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

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

**Modifying Elements:**

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

arr[1] = 'js';

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

**Array Methods:**

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

**Array Length:**

**1. Array length:**

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

let arr = [1, 2, “html”, “Css”];

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

**2. Array at():**

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

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

let array1 = [1, 2, 3];

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

let newArray = array1.concat(array2);

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

**4. Array splice():**

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

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

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

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

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

**5. Array slice():**

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

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

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

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

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

If startIndex is negative:

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

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

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

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

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

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

**6.  Array pop():**

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

**7. Array push():**

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

**8. Array shift():**

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

**9. Array unshift():**

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

**Date:  22-10-2024**

**Day :  Tuesday  
Topic: Object methods**

**Objects**

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

**Creating Objects:**

1. **Literal notation:**

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

1. **Using the Object constructor:**

let person = new Object();

person.name = "John";

person.age = 30;

**Accessing Object Properties:**

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

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

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

**Adding and Modifying Properties:**

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

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

**Deleting Properties:**

delete person.age;

**Object Methods:**

Methods are functions stored as object properties.

let person = {

name: "John",

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

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

**Object Iteration:**

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

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

**Methods in js**

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

    const obj = { a: 1, b: 2, c: 3 };

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

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

    const obj = { a: 1, b: 2, c: 3 };

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

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

    const obj = { a: 1, b: 2, c: 3 };

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

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

    const target = { a: 1, b: 2 };

    const source = { b: 3, c: 4 };

    Object.assign(target, source);

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

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

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

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

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

    const obj = { a: 1, b: 2 };

    Object.freeze(obj);

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

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

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

    const obj = { a: 1, b: 2 };

    Object.seal(obj);

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

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

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

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

**How to itterate objects**

**for...in Loop**

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

for ( key in obj) {

  console.log( key + obj[key]);

}

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

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

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

  console.log(`${key}: ${obj[key]}`);

}

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

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

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

  console.log(value);

}

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

var obj={

  name:"johnn",

  age:20,

  city:{

    name:"new york",

  }

}

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

  console.log(`${key}: ${value}`);

}

**Date:  23-10-2024**

**Day :  Wednesday  
Topic: Number and math methods**

**Number methods**

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

let num = 123.45678;

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

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

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

let num1 = 150;

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

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

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

let str = "123";

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

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

// Example 1: Basic Parsing

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

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

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

// Example 2: Handling Leading and Trailing Whitespaces

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

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

// Example 3: Parsing with Non-Numeric Characters

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

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

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

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

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

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

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

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

// Example 6: Parsing Strings with Exponential Notation

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

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

// Example 8: Large and Small Numbers

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

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

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

**NUMBER** – False

**Not a Number** - True

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

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

**5. Number method**

The Number constructor converts a value to a number.

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

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

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

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

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

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

**Type Coercion:**

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

isNaN(NaN); // true

isNaN(undefined); // true

isNaN({}); // true

isNaN(true); // false

isNaN(false);//false

isNaN(null); // false

isNaN(37); // false

// Strings

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

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

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

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

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

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

// Dates

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

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

// Arrays

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

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

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

**Type Coercion**

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

**Types of Type Coercion**

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

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

**Explicit Type Conversion**

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

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

let num = 123;

let str = String(num);

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

**2.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

**Math methods**

**1. Math.abs()**

Returns the absolute value of a number.

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

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

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

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

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

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

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

**2. Math.ceil()**

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

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

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

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

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

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

**3. Math.floor()**

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

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

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

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

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

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

**4. Math.round()**

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

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

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

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

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

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

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

**5. Math.trunc()**

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

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

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

    console.log(Math.trunc(0)); // 0

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

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

**6. Math.max()**

Returns the largest of zero or more numbers

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

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

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

**7. Math.min()**

Returns the smallest of zero or more numbers.

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

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

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

**8. Math.pow()**

Returns the base raised to the power of the exponent.

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

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

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

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

**9. Math.sqrt()**

Returns the square root of a number.

    console.log(Math.sqrt(16)); // 4

    console.log(Math.sqrt(9)); // 3

    console.log(Math.sqrt(0)); // 0

**10. Math.random()**

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

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

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

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

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

**Date:  24-10-2024**

**Day :  Friday  
Topic: Date methods**

**Date Methods**

**Creating Dates**

Var now=new Date();

**Specific Date and Time**

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

**Getting Date Components**

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

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

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

**Day of the Week**

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

**Hours, Minutes, Seconds, Milliseconds**

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

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

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

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

**Setting Date Components**

now.setFullYear(2025);

now.setMonth(6); // July

 now.setDate(15);

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

now.setHours(15);

now.setMinutes(30);

now.setSeconds(45);

now.setMilliseconds(500);

**Formatting Date and Time**

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

1. **toDateString()**

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

**2.toTimeString()**

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

**3.toLocaleDateString()**

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

**4.toLocaleTimeString()**

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

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

**Age calculator**

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

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

    <title>Document</title>

  </head>

  <body>

    <input type="date" id="el1" />

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

    <script>

      function fun() {

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

        var olddate = new Date(d);

        var newdate = new Date();

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

        var milliseconds = newdate - olddate;

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

        console.log(days);

        console.log(year);

      }

    </script>

  </body>

</html>