**JAVASCRIPT**

JavaScript is the most popular programming language known as scripting language.

Scripting language cannot run individually.

It needs some other language, in combination to execute or run the programs.

JavaScript is the programming language of the Web.

JavaScript is easy to learn.

JavaScript is one of the 3 languages all web developers must learn:

1. HTML to define the content of web pages

2. CSS to specify the layout of web pages

3. JavaScript to program the behavior of web pages

JavaScript is Case Sensitive

**Application of JavaScript**

JavaScript is used to create interactive websites. It is mainly used for:

Client-side validation,

Dynamic drop-down menus,

Displaying date and time,

Displaying pop-up windows and dialog boxes (like an alert dialog box, confirm dialog box and prompt dialog box),

Displaying clocks etc.

**JavaScript Comments**

Not all JavaScript statements are "executed".

Code after double slashes // or between /\* and \*/ is treated as a comment.

**JavaScript Identifiers / Names**

Identifiers are JavaScript names. Identifiers are used to name variables and keywords, and functions. The rules for legal names are the same in most programming languages.

A JavaScript name must begin with:

A letter (A-Z or a-z)

A dollar sign ($)

Or an underscore (\_)

Numbers

Subsequent characters may be letters, digits, underscore.

**1. Internal JavaScript**

Internal JavaScript is written inside the <script> tag within an HTML file.

* Inside the <head> section (not recommended for performance reasons).
* Inside the <body> section (recommended when interacting with the DOM).
* At the bottom of the <body> (best practice to ensure HTML loads first).

**2. External JavaScript**

* External JavaScript is written in a separate .js file and linked to an HTML file using the <script> tag with the src attribute.
* <script src="script.js"></script>
* Place this inside the <body> before closing the </body> tag or inside <head> with defer.

**What is a Variable?**

* A variable is a named container that stores data, which can be used and modified throughout a program. In JavaScript, variables are declared using var, let, or const.

**Types of Variables in JavaScript**

**1. var (Function-Scoped Variable)**

* Can be **redeclared** and **updated**.
* Function-scoped (accessible within the function it is declared in).
* Can be **hoisted** (moved to the top of their scope but not initialized).
* **Not recommended** due to scope issues.

**2. let (Block-Scoped Variable)**

* Can be **updated** but **not redeclared** in the same scope.
* Block-scoped (only available inside {} where it is declared).
* Hoisted but **not initialized** (cannot be used before declaration).
* **Recommended** for variables that need to change.

**3. const (Constant Variable - Block Scoped)**

* **Cannot be redeclared or updated**.
* Block-scoped like let.
* Hoisted but **must be initialized** at declaration.

**Data Types in JavaScript**

JavaScript has **two main categories** of data types:

1. **Primitive Data Types** (immutable, stored directly)
2. **Non-Primitive (Reference) Data Types** (mutable, stored as references)

**1. Primitive Data Types**

Primitive data types are basic types that store single values and are immutable (cannot be changed after creation).

**1.1 String**

* Represents textual data.
* Defined using single ('), double (") or backticks (` for template literals).

**1.2 Number**

* Represents both integers and floating-point numbers.
* JavaScript does not differentiate between int and float.

**1.3 Boolean**

* Represents true or false values (logical type).
* Used in conditions and decision-making.

**1.4 Undefined**

* A variable that has been declared but **not assigned a value**.
* Default value of uninitialized variables.

**1.5 Null**

* Represents an **intentional absence of value**.
* Used when a variable should have "no value".

**2. Non-Primitive (Reference) Data Types**

These store references to memory locations instead of actual values.

**2.1 Object**

* A collection of key-value pairs.
* Can store multiple types of data.

**2.2 Array**

* Ordered collection of multiple values.
* Index-based (starts from 0).

OPERATORS IN JAVASCRIPT:

* JavaScript has various types of operators that perform different kinds of operations on values. Here are some of the most commonly used categories of operators along with examples for each:

**1. Arithmetic Operators**

* These operators perform basic mathematical operations like addition (+), subtraction (-), multiplication (\*), division (/), modulus (%), increment (++), and decrement (--). They allow you to perform calculations on numbers.

**2. Assignment Operators**

* Assignment operators are used to assign values to variables. These include =, +=, -=, \*=, /=, and %= which also combine the operation with the assignment.

**3. Comparison Operators**

* Comparison operators are used to compare two values and return a boolean value (true or false). The most common ones are == (equal to), === (strict equal to), != (not equal to), !== (strict not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

**4. Logical Operators**

* Logical operators perform logical operations on boolean values. These include && (AND), || (OR), and ! (NOT), which evaluate expressions and return boolean results.

**5. Bitwise Operators**

* Bitwise operators work with binary representations of numbers and perform operations like & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift), and >>> (unsigned right shift).

**6. Ternary (Conditional) Operator**

* The ternary operator is a shorthand for an if-else statement, written as condition ? expr1 : expr2, where it evaluates the condition and returns expr1 if true, otherwise expr2.

**7. Unary Operators**

* Unary operators perform operations on a single operand. Common examples are + (unary plus), - (unary minus), ++ (increment), -- (decrement), typeof (to check the type of a variable), and delete (to remove a property from an object).

**8. Type Conversion Operators**

* Type conversion operators are used to convert data from one type to another. Examples include Number() to convert to a number, String() to convert to a string, and Boolean() to convert to a boolean.

**Control Flow and Conditional Statements in JavaScript**

Control flow refers to the order in which individual statements, instructions, or function calls are executed or evaluated. In JavaScript, conditional statements and loops are used to control the flow of a program, depending on whether a condition is true or false. Here's an overview of how control flow and conditional statements work in JavaScript.

**1. if Statement**

The if statement is used to execute a block of code if a specified condition is true.

javascript

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

In this example, the if statement checks if age is greater than or equal to 18. If it is, the message "You are an adult" is logged. Otherwise, the message "You are a minor" is logged.

**2. if-else Statement**

The if-else statement provides an alternative block of code that runs when the condition is false.

javascript

let time = 14;

if (time < 12) {

console.log("Good Morning");

} else {

console.log("Good Afternoon");

}

Here, if time is less than 12, it will print "Good Morning." If time is 12 or greater, it will print "Good Afternoon."

**3. if-else if-else Statement**

The if-else if-else statement is used when there are multiple conditions to check.

javascript

let temperature = 25;

if (temperature > 30) {

console.log("It's a hot day.");

} else if (temperature > 20) {

console.log("It's a warm day.");

} else {

console.log("It's a cool day.");

}

This checks multiple conditions. If the temperature is above 30, "It's a hot day" is printed. If it's between 20 and 30, "It's a warm day" is printed. If it's less than or equal to 20, "It's a cool day" is printed.

**4. switch Statement**

A switch statement is used to evaluate one expression against different cases. It can be more readable than using many if-else statements when dealing with multiple conditions.

javascript

let day = 3;

switch (day) {

case 1:

console.log("Monday");

break;

case 2:

console.log("Tuesday");

break;

case 3:

console.log("Wednesday");

break;

case 4:

console.log("Thursday");

break;

case 5:

console.log("Friday");

break;

case 6:

console.log("Saturday");

break;

case 7:

console.log("Sunday");

break;

default:

console.log("Invalid day");

}

In this example, the switch statement checks the value of day. If it is 3, the message "Wednesday" is logged. If none of the cases match, the default block runs.

**5. ternary (Conditional) Operator**

The ternary operator is a shorthand version of the if-else statement. It takes a condition followed by ?, then the result if true, and the result if false.

javascript

let age = 21;

let result = (age >= 18) ? "Adult" : "Minor";

console.log(result); // Output: "Adult"

This is equivalent to writing an if-else statement but in a more compact form.

**6. break and continue Statements**

* **break** is used to exit from a loop or switch statement immediately.
* **continue** is used to skip the current iteration of a loop and continue with the next iteration.

javascript

// Example of break

for (let i = 1; i <= 5; i++) {

if (i === 3) {

break; // Exit the loop when i equals 3

}

console.log(i); // Output: 1, 2

}

// Example of continue

for (let i = 1; i <= 5; i++) {

if (i === 3) {

continue; // Skip iteration when i equals 3

}

console.log(i); // Output: 1, 2, 4, 5

}

**Dialog Boxes in JavaScript**

Dialog boxes in JavaScript are used to interact with the user by displaying messages or collecting user input. JavaScript provides three built-in dialog methods:

1. **alert()**
2. **confirm()**
3. **prompt()**

These methods create small pop-up boxes in the browser that allow developers to communicate with users or collect information.

**alert()**

The alert() method is used to display an alert box with a specified message. This dialog box contains an OK button, and once the user clicks it, the alert box closes.

**Syntax:**

javascript

alert(message);

**2. confirm()**

The confirm() method displays a dialog box with a specified message, and it includes "OK" and "Cancel" buttons. It returns true if the user clicks "OK" and false if the user clicks "Cancel". This is useful for asking users to confirm their actions (like deleting something).

**Syntax:**

javascript

let result = confirm(message);

**3. prompt()**

The prompt() method is used to display a dialog box that asks the user for input. It provides a text field where the user can type in a response. It returns the input value as a string, or null if the user clicks "Cancel".

**Syntax:**

javascript

let userInput = prompt(message, defaultValue);

**JavaScript Functions**

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

Code reusability: We can call a function several times so it save coding.

Less coding: It makes our program compact. We don’t need to write many lines of code each time to perform a common task.

**1. Types of Scopes in JavaScript**

There are primarily **THREE types of scopes** in JavaScript:

1. **Global Scope**
2. **Local Scope (Function Scope)**
3. **Block Scope**

**Array Methods in JavaScript**

JavaScript provides a variety of built-in methods that you can use to manipulate arrays. Here's a summary of key array methods:

**1. push()**

* Adds one or more elements to the end of an array.
* Returns the new length of the array.

**2. pop()**

* Removes the last element from an array.
* Returns the removed element.

**3. shift()**

* Removes the first element from an array.
* Returns the removed element.

**4. unshift()**

* Adds one or more elements to the beginning of an array.
* Returns the new length of the array.

**5. concat()**

* Combines two or more arrays and returns a new array.

**6. join()**

* Combines all elements of an array into a single string with a specified separator.

**7. slice()**

* Returns a shallow copy of a portion of an array into a new array.

**8. splice()**

* Changes the contents of an array by removing, replacing, or adding elements.
* Returns an array of removed elements.

**9. forEach()**

* Executes a provided function once for each array element.
* Does not return a value.

**10. map()**

* Creates a new array populated with the results of calling a provided function on every element in the array.

**11. filter()**

* Creates a new array with all elements that pass the test implemented by the provided function.

**12. reduce()**

* Applies a function to accumulate array values into a single result, processing from left to right.

**13. reduceRight()**

* Similar to reduce(), but processes the array from right to left.

**14. some()**

* Tests if at least one element in the array passes the provided test function.
* Returns true or false.

**15. every()**

* Tests if all elements in the array pass the provided test function.
* Returns true or false.

**16. find()**

* Returns the first element in the array that satisfies the provided test function.
* Returns undefined if no elements satisfy the condition.

**17. find Index()**

* Returns the index of the first element that satisfies the provided test function.
* Returns -1 if no elements satisfy the condition.

**18. includes()**

* Checks if an array contains a specific element.
* Returns true or false.

**19. sort()**

* Sorts the elements of an array in place.
* Returns the sorted array.

**JavaScript String Methods:**

Javascript provides various methods to perform operations on the string. The following are the methods:

**1. slice()**

* **Purpose**: Extracts a section of a string and returns a new string.
* **Parameters**: start (the index at which to begin extraction), end (the index before which to end extraction).
* **Behavior**: It does not modify the original string. If end is omitted, it extracts from start to the end of the string.

**2. substring()**

* **Purpose**: Extracts characters from a string between two specified indices.
* **Parameters**: start (the index to begin extraction), end (the index at which to stop extraction).
* **Behavior**: If end is omitted, it extracts characters from start to the end of the string. It also swaps start and end if start > end.

**3. substr()**

* **Purpose**: Returns a substring starting at a specified index and continues for a specified length.
* **Parameters**: start (the index to begin extraction), length (the number of characters to extract).
* **Behavior**: If length is omitted, it extracts everything from start to the end of the string.

**4. replace()**

* **Purpose**: Replaces the first occurrence of a specified substring with a new substring.
* **Parameters**: searchValue (the substring to search for), replaceValue (the value to replace searchValue with).
* **Behavior**: This method only replaces the first match unless a regular expression with the global flag (g) is used.

**5. replaceAll()**

* **Purpose**: Replaces all occurrences of a specified substring with a new substring.
* **Parameters**: searchValue (the substring to search for), replaceValue (the value to replace searchValue with).
* **Behavior**: Unlike replace(), it replaces all matches in the string.

**6. toUpperCase()**

* **Purpose**: Converts all characters in a string to uppercase.
* **Returns**: A new string with all characters in uppercase.

**7. toLowerCase()**

* **Purpose**: Converts all characters in a string to lowercase.
* **Returns**: A new string with all characters in lowercase.

**8. concat()**

* **Purpose**: Combines two or more strings into a single string.
* **Parameters**: One or more strings to concatenate.
* **Returns**: A new string with the combined values.

**9. trim()**

* **Purpose**: Removes whitespace from both ends of a string.
* **Returns**: A new string with the whitespace removed from the beginning and end.

**10. trimStart()**

* **Purpose**: Removes whitespace from the beginning of a string.
* **Returns**: A new string with the leading whitespace removed.

**11. trimEnd()**

* **Purpose**: Removes whitespace from the end of a string.
* **Returns**: A new string with the trailing whitespace removed.

**12. padStart()**

* **Purpose**: Pads the beginning of a string with another string until it reaches a specified length.
* **Parameters**: targetLength (the length the string should reach), padString (the string to pad with).
* **Returns**: A new string with the padding added at the start.

**13. padEnd()**

* **Purpose**: Pads the end of a string with another string until it reaches a specified length.
* **Parameters**: targetLength (the length the string should reach), padString (the string to pad with).
* **Returns**: A new string with the padding added at the end.

**14. charAt()**

* **Purpose**: Returns the character at a specified index.
* **Parameters**: index (the position of the character to retrieve).
* **Returns**: A string representing the character at the specified index.

**15. charCodeAt()**

* **Purpose**: Returns the Unicode value of the character at a specified index.
* **Parameters**: index (the position of the character to retrieve the Unicode value of).
* **Returns**: A number representing the Unicode value of the character at the specified index.

**16. split()**

* **Purpose**: Splits a string into an array of substrings based on a specified delimiter.
* **Parameters**: separator (the delimiter to split the string by), limit (optional, the maximum number of splits).
* **Returns**: An array of substrings.

**Extracting String Parts**

JavaScript provides three methods for extracting parts of a string:

1. **slice(start, end)**: Extracts a part of the string between start and end indices.
2. **substring(start, end)**: Extracts a part of the string between start and end indices, but adjusts if start > end.
3. **substr(start, length)**: Extracts a substring starting from start and continues for a specified length.

LOOPING STATEMENTS:

In javascript looping statements are used to execute the set or block of statements continuously until the condition fails. There are three types of looping statements in javascript. 1.While 2. Do-while 3. for

Loops are handy, if you want to run the same code over and over again, each time with a different value.

FOR LOOP:

The JavaScript for loop iterates the elements for the fixed number of times. It should be used if number of iteration is known. The syntax of for loop is given below.

for(intialization;condition;inc/dec)

{

statements;

}

JavaScript while loop

The JavaScript while loop iterates the elements for the n number of times. It should be used if we want certain statements to iterate continuously until the condition gets failed. The syntax of while loop is given below.

while (condition)

{

Statements1;

}

JavaScript do while loop

The JavaScript do while loop iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false. The syntax of do while loop is given below.

do{

code to be executed

}while (condition);

**DOM Manipulation in JavaScript**

**DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of the document as a tree of nodes, where each node corresponds to a part of the page (elements, text, attributes, etc.). JavaScript can interact with the DOM to dynamically change the content, structure, and styling of a web page.

**DOM manipulation** refers to the process of using JavaScript to access and modify the elements of a web page. Through DOM manipulation, you can change the page dynamically, allowing for more interactive and responsive websites.

**Basic DOM Manipulation Concepts**

1. **Selecting Elements**: The first step in DOM manipulation is selecting the elements you want to modify.
2. **Modifying Elements**: Once an element is selected, you can change its content, attributes, or styling.
3. **Creating and Removing Elements**: You can dynamically create new elements and add them to the DOM, or remove existing ones.

**Selecting Elements**

You need to select the HTML elements that you want to work with. There are several methods for selecting DOM elements:

**1. getElementById()**

Selects an element by its ID.

javascript

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

**2. getElementsByClassName()**

Selects elements by their class name. It returns a live HTMLCollection of elements.

javascript

let elements = document.getElementsByClassName("myClass");

**3. getElementsByTagName()**

Selects elements by their tag name. It returns a live HTMLCollection of elements.

javascript

let elements = document.getElementsByTagName("div");

**4. querySelector()**

Selects the first element that matches the specified CSS selector.

javascript

let element = document.querySelector(".myClass");

**5. querySelectorAll()**

Selects all elements that match the specified CSS selector. It returns a static NodeList.

javascript

let elements = document.querySelectorAll("p.myClass");

**setAttribute()**

* **Description**: Sets the value of a specified attribute on an HTML element.
* **Purpose**: To modify an element’s attributes (e.g., href, src, class).
* **Use Cases**:
  + Changing an element's attribute, such as updating the src attribute of an image or changing the class of a button.
  + Dynamically modifying the behavior of form elements or links.

**getAttribute()**

* **Description**: Retrieves the value of a specified attribute from an HTML element.
* **Purpose**: To access the value of an attribute from an element.
* **Use Cases**:
  + Getting the value of the src attribute of an image or the href attribute of a link.
  + Fetching custom attributes used for data storage or other purposes.

**removeAttribute()**

* **Description**: Removes a specified attribute from an element.
* **Purpose**: To delete an attribute from an element.
* **Use Cases**:
  + Removing unnecessary attributes (e.g., disabled from a button once it is enabled).
  + Clearing specific attributes after a certain action (e.g., removing a selected attribute after a choice is made).

**Modifying Elements**

Once you have selected an element, you can modify various properties, including its content, attributes, or styling.

**1. Modifying Text Content**

Use textContent to change or retrieve the text content of an element.

javascript

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

element.textContent = "New Text Content";

**2. Modifying HTML Content**

Use innerHTML to change or retrieve the HTML content (including elements) of an element.

javascript

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

element.innerHTML = "<p>New HTML Content</p>";

**3. Modifying Attributes**

You can change or retrieve the attributes (like href, src, class, etc.) of an element using setAttribute() and getAttribute().

javascript

let link = document.getElementById("myLink");

link.setAttribute("href", "https://www.newurl.com");

To get the attribute value:

javascript

let link = document.getElementById("myLink");

let href = link.getAttribute("href");

console.log(href); // Output: https://www.newurl.com

**4. Modifying CSS Styles**

Use the style property to modify an element's inline CSS styles.

javascript

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

element.style.color = "red";

element.style.backgroundColor = "yellow";

**Creating and Removing Elements**

You can dynamically create new elements and remove existing ones from the DOM.

**1. Creating New Elements**

Use document.createElement() to create a new element.

javascript

let newDiv = document.createElement("div");

newDiv.textContent = "I am a new div!";

document.body.appendChild(newDiv);

**2. Removing Elements**

Use parentElement.removeChild() to remove an element from the DOM.

javascript

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

element.parentElement.removeChild(element);

Alternatively, you can use remove() directly on an element.

javascript

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

element.remove();

**Events and Event Listeners in JavaScript**

In JavaScript, **events** and **event listeners** are key concepts for building interactive websites. Events allow web pages to respond to user actions such as clicks, keypresses, mouse movements, etc. **Event listeners** are functions that listen for specific events and execute code when that event is triggered.

**What is an Event?**

An **event** is an action or occurrence that can be detected and handled by JavaScript. Events are typically triggered by user interactions with elements on the webpage, but they can also be triggered by the browser itself or the system.

**Common Event Types:**

1. **Mouse Events**:
   * click: Triggered when an element is clicked.
   * dblclick: Triggered when an element is double-clicked.
   * mouseenter: Triggered when the mouse enters an element.
   * mouseleave: Triggered when the mouse leaves an element.
   * mousemove: Triggered when the mouse moves over an element.
2. **Keyboard Events**:
   * keydown: Triggered when a key is pressed down.
   * keyup: Triggered when a key is released.
   * keypress: Triggered when a key is pressed and produces a character.
3. **Form Events**:
   * submit: Triggered when a form is submitted.
   * change: Triggered when the value of a form element changes.
   * input: Triggered when the value of an <input>, <textarea>, or <select> element is changed by the user.
4. **Focus Events**:
   * focus: Triggered when an element gains focus (e.g., when a user clicks on an input field).
   * blur: Triggered when an element loses focus (e.g., when the user clicks outside of an input field).
5. **Window Events**:
   * load: Triggered when the entire page (including images and other resources) has loaded.
   * resize: Triggered when the window is resized.
   * scroll: Triggered when the user scrolls in the browser window.

**What is an Event Listener?**

An **event listener** is a function that "listens" for a specific event to occur and then executes some code when that event happens. It is a way to **attach** an event handler to an element so that it responds to the user action.

**Attaching Event Listeners**

**1. addEventListener() Method**

The most common way to attach an event listener to an element is by using the addEventListener() method. It allows you to attach multiple event listeners to a single element and supports various event types.

**Syntax:**

javascript

element.addEventListener(event, function, useCapture);

* **event**: The type of the event (e.g., "click", "keydown").
* **function**: The function to be executed when the event occurs.
* **useCapture**: A Boolean value that determines whether the event should be captured during the capture phase (optional, default is false).

**Example:**

javascript

let button = document.getElementById("myButton");

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

alert("Button clicked!");

});

In this example, when the user clicks the button, the event listener is triggered, and the alert is displayed.

**2. Event Listener with Named Function**

Instead of using an anonymous function, you can use a named function to handle events. This is useful for reuse or easier debugging.

javascript

function handleClick() {

alert("Button clicked!");

}

let button = document.getElementById("myButton");

button.addEventListener("click", handleClick);

**3. Removing Event Listeners with removeEventListener()**

If you no longer want an event listener to be active, you can remove it using the removeEventListener() method. This method requires the same event type and function as the ones used when adding the listener.

javascript

let button = document.getElementById("myButton");

function handleClick() {

alert("Button clicked!");

}

// Adding the event listener

button.addEventListener("click", handleClick);

// Removing the event listener

button.removeEventListener("click", handleClick);

**classList in JavaScript**

The classList property provides a way to access and manipulate the classes of an HTML element. It allows you to easily add, remove, toggle, and check for classes, offering a simpler and more efficient approach than directly modifying the className property.

**Methods Available in classList**

1. **add()**
   * **Description**: Adds one or more classes to an element.
   * **Purpose**: To add one or more classes to the classList of an element. If the class already exists, it is not added again.
   * **Use Case**: Dynamically adding a class to an element, such as changing the background color when a button is clicked.
2. **remove()**
   * **Description**: Removes one or more classes from an element.
   * **Purpose**: To remove specific classes from an element. If the class does not exist, no error is thrown.
   * **Use Case**: Removing a class when an element is no longer in focus or when a specific condition is met.
3. **toggle()**
   * **Description**: Toggles between adding and removing a class. If the class exists, it is removed; if it doesn't, it is added.
   * **Purpose**: To change the class of an element based on a condition (i.e., add it if absent, remove it if present).
   * **Use Case**: Switching between two states, such as toggling visibility or active/inactive styles.
4. **contains()**
   * **Description**: Checks if an element has a particular class.
   * **Purpose**: To determine if an element has a specified class, returning true if the class is present, and false if it is not.
   * **Use Case**: Checking if an element has a class before performing an action, like adding or removing a class.
5. **replace()**
   * **Description**: Replaces an existing class with a new one.
   * **Purpose**: To replace a class in the element's class list with a new class.
   * **Use Case**: Replacing one class with another when toggling between different states or styles.
6. **Date Methods in JavaScript**
7. JavaScript provides a built-in Date object that allows you to work with dates and times. The Date object is used to represent dates and times, and it comes with several methods to manipulate, format, and retrieve different aspects of a date or time.

**Common Date Methods**

1. **new Date()**
   * **Description**: Creates a new Date object representing the current date and time.
   * **Purpose**: To instantiate a new Date object with the current system date and time.
   * **Use Case**: To get the current date and time, e.g., let currentDate = new Date();.
2. **getFullYear()**
   * **Description**: Returns the four-digit year of the specified date.
   * **Purpose**: To get the full year (e.g., 2025) of the date object.
   * **Use Case**: let year = date.getFullYear();
3. **getMonth()**
   * **Description**: Returns the month (0-11) of the specified date.
   * **Purpose**: To retrieve the month from a Date object (note: months are zero-indexed, so January is 0, February is 1, and so on).
   * **Use Case**: let month = date.getMonth();
4. **getDate()**
   * **Description**: Returns the day of the month (1-31) of the specified date.
   * **Purpose**: To get the day of the month (not the day of the week).
   * **Use Case**: let day = date.getDate();
5. **getDay()**
   * **Description**: Returns the day of the week (0-6) for the specified date (0 for Sunday, 1 for Monday, etc.).
   * **Purpose**: To retrieve the day of the week from the date object.
   * **Use Case**: let weekday = date.getDay();
6. **getHours()**
   * **Description**: Returns the hour (0-23) of the specified date.
   * **Purpose**: To retrieve the hour of the day in 24-hour format.
   * **Use Case**: let hours = date.getHours();
7. **getMinutes()**
   * **Description**: Returns the minutes (0-59) of the specified date.
   * **Purpose**: To retrieve the minute component of the time from the Date object.
   * **Use Case**: let minutes = date.getMinutes();
8. **getSeconds()**
   * **Description**: Returns the seconds (0-59) of the specified date.
   * **Purpose**: To retrieve the second component of the time from the Date object.
   * **Use Case**: let seconds = date.getSeconds();
9. **getMilliseconds()**
   * **Description**: Returns the milliseconds (0-999) of the specified date.
   * **Purpose**: To retrieve the milliseconds portion of the Date object.
   * **Use Case**: let milliseconds = date.getMilliseconds();
10. **getTime()**
    * **Description**: Returns the time in milliseconds since January 1, 1970 (the Unix epoch).
    * **Purpose**: To get the timestamp representing the number of milliseconds since the Unix epoch.
    * **Use Case**: let timestamp = date.getTime();
11. **setFullYear(year)**
    * **Description**: Sets the full year (e.g., 2025) of the specified date.
    * **Purpose**: To modify the year of a Date object.
    * **Use Case**: date.setFullYear(2025);
12. **setMonth(month)**
    * **Description**: Sets the month (0-11) of the specified date.
    * **Purpose**: To modify the month of a Date object (months are zero-indexed).
    * **Use Case**: date.setMonth(5); (June)
13. **setDate(day)**
    * **Description**: Sets the day of the month (1-31) of the specified date.
    * **Purpose**: To modify the day of the month for a Date object.
    * **Use Case**: date.setDate(15);
14. **setHours(hours)**
    * **Description**: Sets the hour (0-23) of the specified date.
    * **Purpose**: To modify the hour component of a Date object.
    * **Use Case**: date.setHours(10);
15. **setMinutes(minutes)**
    * **Description**: Sets the minutes (0-59) of the specified date.
    * **Purpose**: To modify the minutes of a Date object.
    * **Use Case**: date.setMinutes(30);
16. **setSeconds(seconds)**
    * **Description**: Sets the seconds (0-59) of the specified date.
    * **Purpose**: To modify the seconds component of a Date object.
    * **Use Case**: date.setSeconds(45);
17. **setMilliseconds(milliseconds)**
    * **Description**: Sets the milliseconds (0-999) of the specified date.
    * **Purpose**: To modify the milliseconds component of a Date object.
    * **Use Case**: date.setMilliseconds(500);

**Math Methods in JavaScript**

JavaScript provides a built-in Math object that includes a variety of methods for performing mathematical operations. These methods cover basic arithmetic, complex mathematical functions, and utilities for working with numbers in your code.

**Common Math Methods**

1. **Math.abs(x)**
   * **Description**: Returns the absolute value of a number x. This method removes the negative sign if the number is negative.
   * **Purpose**: To get the positive magnitude of a number, regardless of whether it is positive or negative.
   * **Use Case**: Math.abs(-10) returns 10.
2. **Math.ceil(x)**
   * **Description**: Returns the smallest integer greater than or equal to x (rounds up).
   * **Purpose**: To round a number upwards to the nearest integer.
   * **Use Case**: Math.ceil(4.3) returns 5.
3. **Math.floor(x)**
   * **Description**: Returns the largest integer less than or equal to x (rounds down).
   * **Purpose**: To round a number downwards to the nearest integer.
   * **Use Case**: Math.floor(4.7) returns 4.
4. **Math.round(x)**
   * **Description**: Rounds a number to the nearest integer. If the fractional part is 0.5 or greater, it rounds up.
   * **Purpose**: To round a number to the nearest whole number.
   * **Use Case**: Math.round(4.5) returns 5.
5. **Math.random()**
   * **Description**: Returns a pseudo-random floating-point number between 0 (inclusive) and 1 (exclusive).
   * **Purpose**: To generate random numbers for various applications like simulations or games.
   * **Use Case**: Math.random() might return a value like 0.6834045920195902.
6. **Math.max(...values)**
   * **Description**: Returns the largest of zero or more numbers provided as arguments.
   * **Purpose**: To find the largest number in a set of values.
   * **Use Case**: Math.max(3, 5, 1) returns 5.
7. **Math.min(...values)**
   * **Description**: Returns the smallest of zero or more numbers provided as arguments.
   * **Purpose**: To find the smallest number in a set of values.
   * **Use Case**: Math.min(3, 5, 1) returns 1.
8. **Math.pow(x, y)**
   * **Description**: Returns x raised to the power of y (x^y).
   * **Purpose**: To calculate the power of a number.
   * **Use Case**: Math.pow(2, 3) returns 8.
9. **Math.sqrt(x)**
   * **Description**: Returns the square root of x.
   * **Purpose**: To calculate the square root of a number.
   * **Use Case**: Math.sqrt(16) returns 4.
10. **Math.sin(x)**
    * **Description**: Returns the sine of x (x is in radians).
    * **Purpose**: To calculate the sine of an angle in radians.
    * **Use Case**: Math.sin(Math.PI / 2) returns 1.
11. **Math.cos(x)**
    * **Description**: Returns the cosine of x (x is in radians).
    * **Purpose**: To calculate the cosine of an angle in radians.
    * **Use Case**: Math.cos(Math.PI) returns -1.
12. **Math.tan(x)**
    * **Description**: Returns the tangent of x (x is in radians).
    * **Purpose**: To calculate the tangent of an angle in radians.
    * **Use Case**: Math.tan(Math.PI / 4) returns 1.
13. **Math.log(x)**
    * **Description**: Returns the natural logarithm (base e) of x.
    * **Purpose**: To compute the natural logarithm of a number.
    * **Use Case**: Math.log(10) returns approximately 2.302585.
14. **Math.log10(x)**
    * **Description**: Returns the base-10 logarithm of x.
    * **Purpose**: To compute the logarithm of a number with base 10.
    * **Use Case**: Math.log10(100) returns 2.
15. **Math.exp(x)**
    * **Description**: Returns e raised to the power of x (where e is Euler's number).
    * **Purpose**: To calculate the exponential value of a number.
    * **Use Case**: Math.exp(1) returns approximately 2.71828.
16. **Math.trunc(x)**
    * **Description**: Returns the integer part of a number by removing the fractional part.
    * **Purpose**: To truncate a number to its integer value (similar to rounding down).
    * **Use Case**: Math.trunc(4.9) returns 4.

**setInterval() and clearInterval() in JavaScript**

In JavaScript, setInterval() and clearInterval() are used to work with timed, repetitive actions. These methods are often used to repeatedly execute a function or code block at a specified time interval, and later stop the execution when it's no longer needed.

**setInterval()**

* **Purpose**: Executes a function repeatedly after every specified number of milliseconds.
* **Syntax**:

javascript

const intervalId = setInterval(function, milliseconds);

* + function: The function you want to run repeatedly.
  + milliseconds: The interval (in milliseconds) between each function call.
* **Returns**: It returns an interval ID (a unique identifier), which can be used later with clearInterval() to stop the interval.

**Use Case:**

You might use setInterval() for:

* Creating animations or effects that occur at regular intervals (e.g., moving an element across the screen every few milliseconds).
* Periodic updates in real-time applications (e.g., polling an API for new data at regular intervals).
* Repeating checks or actions (e.g., checking if a certain condition is met every second).

**clearInterval()**

* **Purpose**: Clears or stops the interval created by setInterval(). This halts the repeated execution of the function.
* **Syntax**:

javascript

clearInterval(intervalId);

* + intervalId: The ID returned by setInterval() when the interval was created. This ID identifies the specific interval you want to stop.
* **Returns**: It doesn't return anything; it simply stops the interval.

**Use Case:**

You might use clearInterval() for:

* Stopping a repeating task after a certain time (e.g., stopping a polling mechanism after a set amount of retries).
* Preventing unnecessary repeated actions after a user input or other condition is met (e.g., stopping an animation or timer after a user clicks a button).

**setTimeout() and clearTimeout() in JavaScript**

* In JavaScript, setTimeout() and clearTimeout() are used to handle delayed execution of functions or code. They allow you to schedule a function to run after a specified delay and cancel that scheduled execution if needed.

**setTimeout()**

* **Purpose**: Executes a function once after a specified number of milliseconds.
* **Syntax**:

javascript

const timeoutId = setTimeout(function, milliseconds);

* + function: The function that you want to execute after the delay.
  + milliseconds: The delay (in milliseconds) before executing the function.
* **Returns**: It returns a timeout ID (a unique identifier), which can be used later with clearTimeout() to cancel the execution.

**Use Case:**

You might use setTimeout() for:

* **Delaying a task**: For instance, waiting for a user to finish typing before submitting data.
* **Creating timed actions**: Delaying animations or transitions in web pages, like fading out an element after a certain time.
* **Triggering events**: You may use it for triggering events or functions after waiting for some condition to be met (e.g., a delay before showing a pop-up message).

**clearTimeout()**

* **Purpose**: Cancels a function that was previously scheduled using setTimeout(), preventing it from executing.
* **Syntax**:

javascript

clearTimeout(timeoutId);

* + timeoutId: The ID returned by setTimeout(). This is used to identify and cancel the scheduled function.
* **Returns**: It doesn't return anything; it simply cancels the scheduled timeout.

**Use Case:**

You might use clearTimeout() for:

* **Canceling delayed actions**: If a certain condition is met or a user interacts with a page, you can cancel a scheduled timeout (e.g., canceling a message from being shown if the user clicks a button).
* **Preventing unnecessary function calls**: When waiting for a function to be executed after a delay, but deciding to cancel the task based on user input or other criteria.