**Advance JavaScript**

1. **Advance Concepts**
2. **Session Storage:** Session Storage is a temporary storage feature in web browsers that saves data only while a tab is open. It allows web pages to store information that remains available even if the page is refreshed. However, once the tab or browser is closed, all stored data is automatically deleted. String is only data type which is directly stored in session storage. If you want to store other types like object, array, numbers or booleans then you need to convert them to string.

**Point to Know about Session Storage:**

1. Storage Capacity: Typically, 5MB.
2. Persistence: Data lasts only as long as the session (tab or window) is open.
3. Scope: Session-specific storage per tab/window. If you open a new tab or window, a new session storage is created.
4. Browser Support: Supported by all major browsers.

**Methods of Session Storage:**

1. setItem() Method: It is used to store the value in a particular location with the name of the key.

sessionStorage.setItem(key, value)

1. getItem() Method: This method takes one parameter that is key which is used to get the value stored with a particular key name.

var user = sessionStorage.getItem("key");

1. removeItem() Method: This is method is used to remove the value stored in the memory in reference to key.

sessionStorage.removeItem(key)

1. clear() Method: This method is used to clear all the values stored in the session storage

sessionStorage.clear()

1. **Local Storage:** Local Storage allows websites to store key-value pairs persistently in a user’s browser, even after the browser is closed and reopened.String is only data type which is directly stored in local storage. If you want to store other types like object, array, numbers or booleans then you need to convert them to string.

**Points to Know about Local Storage:**

1. Storage Capacity: Up to 5MB or 10MB, depending on the browser.
2. Persistence: Data persists even when the browser or tab is closed.
3. Browser Support: Local Storage is supported by all major browsers.

**Methods of Local Storage:**

1. setItem: It is used to store the value in a particular location with the name of the key.

localStorage.setItem(key, value)

1. getItem() Method: This method takes one parameter that is key which is used to get the value stored with a particular key name.

localStorage.getItem(key)

1. removeItem() Method: This is method is used to remove the value stored in the memory in reference to key.

localStorage.removeItem(key)

1. clear() Method: This method is used to clear all the values stored in local storage.

localStorage.clear()

1. **Cookies:** Cookies are small pieces of data stored in a user’s browser, primarily used for session management, authentication, and tracking user activity.

**Points to Know about Cookies:**

1. Storage Capacity: Typically limited to 4KB.
2. Persistence: Expiration is defined when setting the cookie and can vary.
3. Scope: Cookies are domain-specific but can be shared between server and client.
4. **Browser Debugging**

**Different tabs in inspect element window:**

1. Elements: Shows the page's HTML and CSS structure.
2. Console: A place to run JavaScript manually. And displays errors, warnings, and logs from the page.
3. Sources: Contains all the files loaded by the web page (HTML, JS, CSS).
4. Network: Tracks all requests made by the page: images, scripts, API calls, etc. Shows status codes, loading times, request/response headers.
5. Application: Shows data stored by the site: Local Storage, Session Storage, Cookies.
6. Performance: Good for optimizing responsiveness and speed.
7. Memory: Helps monitor memory usage and detect leaks.
8. Lighthouse: Runs audits on your site: performance, accessibility, SEO, and best practices.
9. Security: Displays certificate information and any security warnings.

**Caching:** Caching is the practice of storing data in a temporary location, or cache, so it can be retrieved quickly. This boosts performance and improves user experience.

1. **OOJS study**
2. **What is OOJS?**

OOJS stands for Object-Oriented JavaScript. It uses objects to group related data (properties) and behaviour (methods). JavaScript is prototype-based, not class-based like Java or C++, but ES6 introduced class syntax to make it feel more familiar.

* Objects inherit from other objects, not from classes. This is done using something called a prototype.

**Core Concepts in OOJS:**

1. Encapsulation: Bundling data and methods inside objects to keep things organized.
2. Inheritance: Reusing properties and methods of one class in another.
3. Abstraction: Hiding complex details and exposing only essential features.
4. Polymorphism: Using the same method name in different ways across objects.
5. **Possible ways to implement class**
6. ES6 class Syntax: Introduced in ES6 (ES2015), this is now the standard way to write classes.

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello, I'm ${this.name}`);

}

}

const user = new Person('Kiran');

user.greet(); // Hello, I'm Kiran

1. Constructor Function + Prototype (Traditional): Before ES6, classes were simulated using constructor functions and the prototype chain.

function Person(name) {

this.name = name;

}

Person.prototype.greet = function () {

console.log('Hello, I\'m ' + this.name);

};

const user = new Person('Kiran');

user.greet(); // Hello, I'm Kiran

1. **Static Properties declaration**

* Declared using the static keyword inside a class.
* Belong to the class itself, not to instances created from the class.
* Used for utility functions or shared constants.
* JavaScript does not support making a whole class static

class Calculator {

static pi = 3.14159;

static add(a, b) {

return a + b;

}

}

console.log(Calculator.pi); // 3.14159

console.log(Calculator.add(2, 3)); // 5

const calc = new Calculator(); // calc.pi or calc.add would be undefined

**Declaring Properties in Classes (Instance vs. Static):**

1. Instance Properties: Defined inside the constructor or directly in the class body.
2. Static Properties: Declared with the static keyword and accessed via the class name.
3. **ECMAScript6**
4. **Difference between let, var & const**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **var** | **let** | **const** |
| **Scope** | Function-scoped | Block-scoped | Block-scoped |
| **Redeclare** | Yes | No | No |
| **Reassign** | Yes | Yes | No |
| **Hoisted** | Yes | No | No |

1. **JavaScript Classes**

JavaScript Classes are templates for JavaScript Objects.

Syntax:

class ClassName {  
  constructor() { ... }  
  method\_1() { ... }  
  method\_2() { ... }  
  method\_3() { ... }  
}

**Inheritance:** Classes can inherit from other classes using the extends keyword. This allows you to reuse and extend functionality.

1. **Arrow functions**

An arrow function is a shorthand syntax for writing function expressions. Instead of using the function keyword, you use the => (arrow) operator. Arrow functions do not have their own “this”. Arrow functions are not hoisted. They must be defined before they are used.

Syntax:

// Traditional function

function add(a, b) {

return a + b;

}

// Arrow function

const add = (a, b) => a + b;

const square = x => x \* x;

1. **Import, Export, async, await Functions**

**Import & Export:** JavaScript modules let you split your code into reusable pieces. You use export to share code and import to bring it into another file.

Code:

// utils.js

export const add = (a, b) => a + b;

export default function greet(name) {

return `Hello, ${name}`;

}

* export lets you export multiple named items.
* export default allows one default export per file.

// main.js

import greet from './utils.js'; // default import

import { add } from './utils.js'; // named import

console.log(greet("Kiran")); // Hello, Kiran

console.log(add(2, 3)); // 5

**Async & Await:** These keywords simplify working with Promises, making asynchronous code look synchronous.

* async makes a function return a Promise
* await makes a function wait for a Promise, await can only be used inside an async function

Code:

async function getData() {

const result = await fetch("https://api.example.com");

const data = await result.json();

console.log(data);

}

**Promise:** A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

JavaScript is single-threaded and uses asynchronous programming (like setTimeout, HTTP requests, etc.). Promises help manage and chain these asynchronous tasks more cleanly than traditional callback functions (avoiding "callback hell").

1. **Extra Points**

**Difference between == & ===, != & !===**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Allows Type conversion** | **Example** | **Result** |
| == | Loose equality | Yes | 5 == '5' | true |
| === | Strict equality | No | 5 === '5' | false |
| != | Loose inequality | Yes | 5 != '5' | false |
| !== | Strict inequality | No | 5 !== '5' | true |

Code:

console.log(0 == false); // true

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

console.log(null == undefined); // true

console.log(null === undefined); // false

console.log("5" != 5); // false

console.log("5" !== 5); // true

* **What does type="module" mean in <script type="module">?**

1. This script is an ES6 module, not a classic script.
2. Enables support for import and export statements directly in the browser.
3. The script runs in strict mode automatically (no need for "use strict").
4. Variables and functions inside this script are module-scoped (not global).
5. The script is deferred by default, so it runs after the HTML parsing is done.

* **Use of window object**

1. Global Variables and Functions
2. Interact with or manipulate the browser window
3. Access to the DOM
4. The window object handles global events
5. Storage Access
6. Dialogs and Alerts

* **What happen if we don’t write new keyword at time of object creation in JavaScript**

Without new: this refers to global object.

function Person(name) {

this.name = name;

}

const p = Person("Alice"); // forgot "new"

console.log(p); // undefined

The function runs, and this.name = name assigns "Alice" to the name property of the **global object** (window in browsers).

In strict mode, this will be undefined and trying to assign this.name = name will throw a **TypeError**.

* **Comparison between indexOf, search and match method of string**

| **Feature** | **indexOf()** | **search()** | **match()** |
| --- | --- | --- | --- |
| Accepts Regex | No | Yes | Yes |
| Accepts String | Yes | Yes (converted to regex) | Yes (converted to regex) |
| Returns | Index (number or -1) | Index (number or -1) | Array (or null if no match) |
| Global Matches | No | No | Yes (with /g flag) |
| Capturing Groups | No | Limited | Yes (very useful for groups) |
| Case-sensitive | Yes | Yes (use /i to ignore) | Yes (use /i to ignore) |
| Use Case | Find position of substring | Find position via pattern | Extract matching content |

* **String Methods**

1. length: Returns length of string. Length is property of string not method.

console.log("Hello".length); // 5

1. charAt: Returns character at the given index. If the index is negative or out of range then it returns empty string.

console.log("Hello".charAt(1)); // 'e'

1. concat: Combines the string and returns new string. It does not change original string.

console.log("Hello".concat(" World")); // 'Hello World'

1. at: Returns character at the given index, If the index is negative than it starts from the end of string. And if index is out of range, then it will return undefined.

console.log("Hello".at(-1)); // 'o'

1. slice: Used to extract some part of string. The first index is inclusive and second index id exclusive in this method. If index is negative then it starts from the end of string.

console.log("Hello World".slice(0, 5)); // 'Hello'

1. substring: Used to extract some part of string. The first index is inclusive and second index is exclusive in this method. If index is negative then it will treat it as 0 and it swaps start and end if needed.

console.log("Hello World".substring(0, 5)); // 'Hello'

1. toUpperCase: Converts all characters of string to uppercase and returns new string.

console.log("hello".toUpperCase()); // 'HELLO'

1. toLowerCase: Converts all characters of string to lowercase and returns new string.

console.log("HELLO".toLowerCase()); // 'hello'

1. trim: Trims whitespace from the beginning and end of the string.

console.log(" hello ".trim()); // 'hello'

1. trimStart: Trims whitespace from the beginning of the string.

console.log(" hello".trimStart()); // 'hello'

1. trimEnd: Trims whitespace from the end of the string.

console.log("hello ".trimEnd()); // 'hello'

1. padStart: Pads the string with a given string (repeated and/or truncated, if needed) from beginning so that the resulting string has a given length.

console.log("5".padStart(3, "0")); // '005'

1. padEnd: Pads the string with a given string (repeated and/or truncated, if needed) at ending so that the resulting string has a given length.

console.log("5".padEnd(3, "0")); // '500'

1. repeat: Constructs and returns a new string which contains the specified number of copies of this string, concatenated together.

console.log("ha".repeat(3)); // 'hahaha'

1. replace: Replaces first occurrence of substring or pattern with replacement and returns new string. And also replaces all occurrence by using global flag /g with regex.

console.log("apple banana apple".replace("apple", "orange")); // 'orange banana apple'

1. replaceAll: Replaces all occurrence of substring or pattern and returns new string.

console.log("apple banana apple".replaceAll("apple", "orange")); // 'orange banana orange'

1. split: The split method of String values takes a pattern and divides this string into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and returns the array.

console.log("a,b,c".split(",")); // ['a', 'b', 'c']

1. indexOf: Find and returns index of first occurrence of substring. And returns -1 if substring is not found. And doesn’t support regex.

console.log("banana".indexOf("a")); // 1

1. search: Find and returns index of first occurrence of substring or pattern. And returns -1 if pattern is not found. It supports regex.

console.log("banana".search(/a/)); // 1

1. match: The match method of String values retrieves the result of matching this string against a regular expression. You can find all occurrences using global flag /g.

console.log("banana".match(/a/g)); // ['a', 'a', 'a']

1. lastIndexOf: Find and returns index of last occurrence of substring. And returns -1 if substring is not found. And doesn’t support regex.

console.log("banana".lastIndexOf("a")); // 5

1. includes: Checks substring present or not in given string and returns true or false.

console.log("banana".includes("nan")); // true

1. stratsWith: Checks string starts with given substring or not and returns true or false accordingly.

console.log("banana".startsWith("ban")); // true

1. endsWith: Checks string is ends with given substring or not and returns true or false accordingly.

console.log("banana".endsWith("na")); // true

* **Date Object**
* **Creating Date Objects**

1. new Date(): Creates a new Date object with the current date and time.

let now = new Date();

console.log(now); // Current date and time

1. new Date(milliseconds): Creates a Date from milliseconds since January 1, 1970.

let date = new Date(0);

console.log(date); // Thu Jan 01 1970

1. new Date(dateString): Parses a date string into a Date object.

let date = new Date("2025-08-07T12:00:00Z");

console.log(date.toUTCString()); // Thu, 07 Aug 2025 12:00:00 GMT

1. new Date(year, monthIndex, day, hour, minute, second, millisecond): Creates a date by specifying each component. Year and month component is compulsory and other components are optional.

let date = new Date(2025, 7, 7, 10, 30);

console.log(date); // Thu Aug 07 2025 10:30:00

* **Date Methods**

1. Date.now(): Returns the current timestamp in milliseconds.

console.log(Date.now()); // e.g., 1691609420000

1. Date.parse(dateString): Parses a date string and returns the timestamp.

console.log(Date.parse("2025-08-07")); // e.g., 1754524800000

1. Date.UTC(year, month, day, hour, minute, second, millisecond): Returns the timestamp in milliseconds for the UTC date.

console.log(Date.UTC(2025, 7, 7)); // August is 7 (0-indexed), returns timestamp

1. getFullYear(): Returns the 4-digit year.

const d = new Date();

console.log(d.getFullYear());

1. getMonth(): Returns the month index (Starts with January with index 0).

console.log(d.getMonth()); // 0 = January

1. getDate(): Returns the day of the month (1-31).

console.log(d.getDate()); // Day of month

1. getDay(): Returns the day index of the week.

console.log(d.getDay()); // 0 = Sunday, 6 = Saturday

1. getHours(): Returns hour of date.

console.log(d.getHours());

1. getMinutes(): Returns minutes of date.

console.log(d.getMinutes());

1. getSeconds(): Returns seconds of date.

console.log(d.getSeconds());

1. getMilliseconds(): Returns milliseconds of date.

console.log(d.getMilliseconds());

1. getTime(): Returns the time value in milliseconds since January 1, 1970.

console.log(d.getTime()); // milliseconds since epoch

1. getTimezoneOffset(): Returns the time zone difference from UTC in minutes.

console.log(d.getTimezoneOffset()); // in minutes

1. getUTCFullYear(): returns the year for the date according to universal time.

console.log(d.getUTCFullYear());

1. getUTCMonth(): Returns the month for the date according to universal time, as a zero-based value (where zero indicates the first month of the year).

console.log(d.getUTCMonth());

1. getUTCDate():  Returns the day of the month for the date according to universal time.

console.log(d.getUTCDate());

1. getUTCDay(): Returns the day of the week for the date according to universal time, where 0 represents Sunday.

console.log(d.getUTCDay());

1. getUTCHours(): Returns the hours for the date according to universal time.

console.log(d.getUTCHours());

1. getUTCMinutes(): Returns the minutes for the date according to universal time.

console.log(d.getUTCMinutes());

1. getUTCSeconds(): Returns the seconds in the specified date according to universal time.

console.log(d.getUTCSeconds());

1. getUTCMilliseconds(): Returns the milliseconds for the date according to universal time.

console.log(d.getUTCMilliseconds());

1. setFullYear(): Changes the year, month, and/or day of month for this date according to local time. Year argument is compulsory and other is optional.

d.setFullYear(2022, 11, 25);

console.log(d.toString());

1. setMonth(): Changes the month and/or day of the month for this date according to local time. Month argument is compulsory and other are optional.

d.setMonth(5, 15); // June 15

console.log(d.toString());

1. setDate(): Changes the day of the month for this date according to local time.

d.setDate(10);

console.log(d.toString());

1. setHours(): Changes the hours, minutes, seconds, and/or milliseconds for this date according to local time. Hour argument is compulsory and other are optional.

d.setHours(14, 30, 0, 0);

console.log(d.toString());

1. setMinutes(): Changes the minutes for this date according to local time. Minute argument is compulsory and other are optional.

d.setMinutes(45, 30);

console.log(d.toString());

1. setSeconds(): Changes the seconds and/or milliseconds for this date according to local time. Second argument is compulsory and other are optional.

d.setSeconds(50, 500);

console.log(d.toString());

1. setMilliseconds(): Changes the milliseconds for this date according to local time.

d.setMilliseconds(123);

console.log(d.toString());

1. setTime(): Changes the timestamp for this date, which is the number of milliseconds since the epoch, defined as the midnight at the beginning of January 1, 1970, UTC.

d.setTime(1754524800000); // Set to specific timestamp

console.log(d.toString());

1. setUTCFullYear(): Changes the year for this date according to universal time.

d.setUTCFullYear(2030, 0, 1);

console.log(d.toUTCString());

1. setUTCMonth(): Changes the month and/or day of the month for this date according to universal time.

d.setUTCMonth(6, 20); // July 20 UTC

console.log(d.toUTCString());

1. setUTCDate(): Changes the day of the month for this date according to universal time.

d.setUTCDate(12);

console.log(d.toUTCString());

1. setUTCHours(): Changes the hours, minutes, seconds, and/or milliseconds for this date according to universal time.

d.setUTCHours(10, 15, 0, 0);

console.log(d.toUTCString());

1. setUTCMinutes(): Changes the minutes for this date according to universal time.

d.setUTCMinutes(30, 0, 0);

console.log(d.toUTCString());

1. setUTCSeconds(): Changes the seconds and/or milliseconds for this date according to universal time.

d.setUTCSeconds(45, 999);

console.log(d.toUTCString());

1. setUTCMilliseconds(): Changes the milliseconds for this date according to universal time.

d.setUTCMilliseconds(250);

console.log(d.toUTCString());

1. toString(): Returns full string representation in local time.

console.log(d.toString()); // Local time

1. toISOString(): Returns ISO 8601 formatted string.

console.log(d.toISOString()); // ISO format

1. toUTCString(): Returns a string formatted to UTC.

console.log(d.toUTCString());

1. toJSON(): Returns a string representing the date in the same ISO format as toISOString(). Called automatically when a Date object is serialized to JSON. Internally, it calls toISOString().

console.log(d.toJSON()); // Same as toISOString()

1. toLocaleString(): Converts a Date object to a string, using locale conventions.

console.log(d.toLocaleString()); // e.g., "8/7/2025, 2:00:00 PM"

1. toLocaleDateString(): Returns the date portion of a Date object as a string, using locale conventions.

console.log(d.toLocaleDateString()); // e.g., "8/7/2025"

1. toLocaleTimeString(): Returns the time portion of a Date object as a string, using locale conventions.

console.log(d.toLocaleTimeString()); // e.g., "2:00:00 PM"

1. toDateString(): Returns a string representing the date portion of this date interpreted in the local timezone.

console.log(d.toDateString()); // e.g., "Thu Aug 07 2025"

1. toTimeString(): Returns a string representing the time portion of this date interpreted in the local timezone.

console.log(d.toTimeString()); // e.g., "14:00:00 GMT+0530 (India Standard Time)"

1. valueOf(): The valueOf() method of Date instances returns the number of milliseconds for this date since the epoch, which is defined as the midnight at the beginning of January 1, 1970, UTC. It’s used internally by JavaScript when the Date is used in a numeric context.

console.log(d.valueOf()); // Same as getTime()

* **Math Object**

The Math namespace object contains static properties and methods for mathematical constants and functions.

* **Static properties**

1. Math.E: Euler's number and the base of natural logarithms; approximately 2.718.
2. Math.LN10: Natural logarithm of 10; approximately 2.303.
3. Math.LN2: Natural logarithm of 2; approximately 0.693.
4. Math.LOG10E: Base-10 logarithm of E; approximately 0.434.
5. Math.LOG2E: Base-2 logarithm of E; approximately 1.443.
6. Math.PI: Ratio of a circle's circumference to its diameter; approximately 3.14159.
7. Math.SQRT1\_2: Square root of ½; approximately 0.707.
8. Math.SQRT2: Square root of 2; approximately 1.414.

* **Static methods**

1. Math.round(x): Returns x rounded to its nearest integer

console.log(Math.round(4.6)); // Output: 5

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

1. Math.ceil(x): Returns x rounded up to its nearest integer

console.log(Math.ceil(4.1)); // Output: 5

console.log(Math.ceil(-4.1)); // Output: -4

1. Math.floor(x): Returns x rounded down to its nearest integer

console.log(Math.floor(4.9)); // Output: 4

console.log(Math.floor(-4.1)); // Output: -5

1. Math.trunc(x): Returns the integer part of x

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

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

1. Math.sign(x): Math.sign(x) returns if x is negative, zero or positive.

console.log(Math.sign(10)); // Output: 1

console.log(Math.sign(-10)); // Output: -1

console.log(Math.sign(0)); // Output: 0

1. Math.pow(x, y): Math.pow(x, y) returns the value of x to the power of y

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

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

1. Math.sqrt(x): Math.sqrt(x) returns the square root of x

console.log(Math.sqrt(9)); // Output: 3

console.log(Math.sqrt(16)); // Output: 4

1. Math.cbrt(x): Returns the cubic root of x

console.log(Math.cbrt(8)); // Output: 2

console.log(Math.cbrt(27)); // Output: 3

1. Math.abs(x): Math.abs(x) returns the absolute (positive) value of x

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

console.log(Math.abs(5)); // Output: 5

1. Math.sin(x): Math.sin(x) returns the sine (a value between -1 and 1) of the angle x (given in radians).

console.log(Math.sin(0)); // Output: 0

console.log(Math.sin(Math.PI / 2)); // Output: 1

1. Math.cos(x): Math.cos(x) returns the cosine (a value between -1 and 1) of the angle x (given in radians).

console.log(Math.cos(0)); // Output: 1

console.log(Math.cos(Math.PI)); // Output: -1

1. Math.max(): Returns the largest of the numbers given as input parameters, or -Infinity if there are no parameters.

console.log(Math.max(3, 7, 2, 9)); // Output: 9

console.log(Math.max()); // Output: -Infinity

1. Math.min(): Returns the smallest of the numbers given as input parameters, or Infinity if there are no parameters.

console.log(Math.min(3, 7, 2, 9)); // Output: 2

console.log(Math.min()); // Output: Infinity

1. Math.random(): Math.random() returns a random number between 0 (inclusive), and 1 (exclusive)

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

1. Math.log(x): Math.log(x) returns the natural logarithm of x.

console.log(Math.log(Math.E)); // Output: 1 (natural log of Euler's number)

console.log(Math.log(1)); // Output: 0

1. Math.log2(x): Math.log2(x) returns the base 2 logarithm of x.

console.log(Math.log2(8)); // Output: 3

console.log(Math.log2(1)); // Output: 0

1. Math.log10(x): Math.log10(x) returns the base 10 logarithm of x.

console.log(Math.log10(100)); // Output: 2

console.log(Math.log10(1)); // Output: 0

1. Math.imul(): Returns the result of the C-like 32-bit multiplication of the two parameters.

console.log(Math.imul(2, 4)); // Output: 8

console.log(Math.imul(-2, 4)); // Output: -8

1. Math.hypot(): The Math.hypot() static method returns the square root of the sum of squares of its arguments.

console.log(Math.hypot(3, 4)); // Output: 5

console.log(Math.hypot(3, 4, 12)); // Output: 13

* **Function in JS**

A function in JavaScript is a **block of reusable code** that performs a specific task. You can define it once and call it multiple times.

1. Function Declaration

function sayHello() {

console.log("Hello!");

}

1. Function Expression

const sayHi = function() {

console.log("Hi!");

};

1. Arrow Function (ES6+)

const add = (a, b) => a + b;

1. Anonymous Function

setTimeout(function() {

console.log("Anonymous function");

}, 1000);

1. Constructor Function

function Person(name) {

this.name = name;

}

const p = new Person("Alice");

console.log(p.name); // Alice

* **Array in JS**

An array is an ordered, indexed collection of elements. These elements can be of any type -strings, numbers, objects, functions, even other arrays (nested arrays).

* **Array Methods**

1. Array.length: Returns the number of elements in an array.

const arr = [1, 2, 3];

console.log(arr.length); // 3

1. Array.toString(): Converts an array to a comma-separated string.

const arr = [1, 2, 3];

console.log(arr.toString()); // "1,2,3"

1. Array.at(index): Returns the element at the given index (supports negative indexing).

const arr = [10, 20, 30];

console.log(arr.at(1)); // 20

1. Array.join(separator): Joins all elements into a string, with an optional separator.

const arr = ["a", "b", "c"];

console.log(arr.join("-")); // "a-b-c"

1. Array.pop(): Removes the last element and returns it.

const arr = [1, 2, 3];

arr.pop(); // returns 3

1. Array.push(...items): Adds elements to the end and returns new length.

const arr = [1, 2];

arr.push(10); // returns 3

1. Array.shift(): Removes the first element and returns it.

const arr = [1, 2, 3];

arr.shift(); // returns 1

1. Array.unshift(...items): Adds elements to the beginning and returns new length.

const arr = [2, 3];

arr.unshift(1); // returns 3

1. delete array[index]: Removes an item at index but does not change array length (leaves undefined).

const arr = [1, 2, 3];

delete arr[1];

console.log(arr); // [1, empty, 3]

1. Array.concat(...arrays): Merges arrays and returns a new one.

const a = [1, 2];

const b = [3, 4];

console.log(a.concat(b)); // [1, 2, 3, 4]

1. Array.copyWithin(target, start, end?): Copies elements within the array, modifying it.

const arr = [1, 2, 3, 4];

arr.copyWithin(1, 2);

console.log(arr); // [1, 3, 4, 4]

1. Array.flat(depth?): Flattens nested arrays.

const arr = [1, [2, [3]]];

console.log(arr.flat(2)); // [1, 2, 3]

1. Array.slice(start, end?): Returns a shallow copy (doesn’t modify original).

const arr = [1, 2, 3, 4];

console.log(arr.slice(1, 3)); // [2, 3]

1. Array.splice(start, deleteCount, ...items): Adds/removes items in place.

const arr = [1, 2, 3, 4];

arr.splice(1, 2, 'a', 'b');

console.log(arr); // [1, 'a', 'b', 4]

1. Array.toSpliced(start, deleteCount, ...items): Immutable version of splice() — returns new array.

const arr = [1, 2, 3, 4];

const newArr = arr.toSpliced(1, 2, 'a');

console.log(newArr); // [1, 'a', 4]

1. Array.indexOf(item): Returns first index of item or -1.

const arr = [1, 2, 3, 2];

console.log(arr.indexOf(2)); // 1

1. Array.lastIndexOf(item): Returns last index of item or -1.

const arr = [1, 2, 3, 2];

console.log(arr.lastIndexOf(2)); // 3

1. Array.includes(item): Checks if item exists.

const arr = [1, 2, 3];

console.log(arr.includes(2)); // true

1. Array.find(callback): Returns the first element that matches the condition.

const arr = [5, 10, 15];

const result = arr.find(x => x > 10);

console.log(result); // 15

1. Array.findIndex(callback): Returns index of first matching item.

const arr = [5, 10, 15];

console.log(arr.findIndex(x => x > 10)); // 2

1. Array.findLast(callback): Returns last element matching the condition.

const arr = [5, 10, 15, 20];

console.log(arr.findLast(x => x > 10)); // 20

1. Array.findLastIndex(callback): Returns index of last matching item.

const arr = [5, 10, 15, 20];

console.log(arr.findLastIndex(x => x > 10)); // 3

1. Array.sort([compareFn]): Sorts the array in place. Converts elements to strings by default.

const arr = [10, 2, 30];

arr.sort();

console.log(arr); // [10, 2, 30] → actually ["10", "2", "30"] lexically

// Numeric sort:

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

console.log(arr); // [2, 10, 30]

1. Array.reverse(): Reverses the array in place.

const arr = [1, 2, 3];

arr.reverse();

1. Array.toSorted([compareFn]): Immutable version of sort() — returns new array.

const arr = [3, 1, 2];

const sorted = arr.toSorted();

1. Array.toReversed(): Immutable version of reverse().

const arr = [1, 2, 3];

const rev = arr.toReversed();

1. Array.forEach(callback): Loops over each item - no return value.

[1, 2, 3].forEach(item => console.log(item)); // logs: 1, 2, 3

1. Array.map(callback): Returns a new array by transforming each item.

const nums = [1, 2, 3];

const squared = nums.map(n => n \* n);

console.log(squared); // [1, 4, 9]

1. Array.flatMap(callback): Maps and flattens one level.

const arr = [1, 2];

const result = arr.flatMap(x => [x, x \* 2]);

console.log(result); // [1, 2, 2, 4]

1. Array.filter(callback): Returns new array with items that match condition.

const nums = [1, 2, 3, 4];

const even = nums.filter(n => n % 2 === 0);

console.log(even); // [2, 4]

1. Array.reduce(callback, initialValue): Reduces array to a single value.

const nums = [1, 2, 3];

const sum = nums.reduce((total, num) => total + num, 0);

console.log(sum); // 6

1. Array.every(callback): Returns true if all items match condition.

const nums = [2, 4, 6];

console.log(nums.every(n => n % 2 === 0)); // true

1. Array.some(callback): Returns true if any item matches condition.

const nums = [1, 3, 4];

console.log(nums.some(n => n % 2 === 0)); // true

1. Array.from(iterable): Creates array from iterable (like string or Set).

console.log(Array.from("hello")); // ['h', 'e', 'l', 'l', 'o']

1. Array.keys(): Returns an iterator of keys (indexes).

const arr = ["a", "b"];

const keys = arr.keys();

console.log([...keys]); // [0, 1]

1. Array.entries(): Returns an iterator of [index, value] pairs.

const arr = ["a", "b"];

for (const [i, val] of arr.entries()) {

console.log(i, val);

}

1. Spread ... : Expands array elements.

const arr = [1, 2, 3];

const newArr = [...arr, 4];

console.log(newArr); // [1, 2, 3, 4]

1. Rest ... : Collects remaining elements.

const [a, ...rest] = [1, 2, 3, 4];

console.log(a); // 1

console.log(rest); // [2, 3, 4]