**What is java script:**

JavaScript is a programming language that allows you to make websites interactive. It runs in the web browser and works with HTML and CSS to build modern web pages.

**Variable** : var , let, const.

**Operator and expression:** + , - , / , \* , < = , > = , == , != , ===.

**Logical operator :** And && , OR || , logical !.

**Ternary operator :** condition ? expression if true : expression if false;

**Eg**: var result = age>=18? ”yes” : “No”;

**Strict equality (===):** its mean not only compare value also compare data type.

**Concatenate string:**

To concatenate string use (+) operator.

In javascript use camel case to to declear a variable (eg: myName)

**Example:**

var name=’bilal’+’ahmed’;

console.log(name); // result equals to bilalahmed.

**parseInt:** use to convert decimal points number into number;

**parseFloat :** use for decimal points.

**toFixed function :** use in expression for how many numbers you want after decimal point.

**eg**: var sum=0.1+0.2;

console.log(sum.toFixed(2)); // result = 0.30

**condition statement:** if else, else if ,nested if else or else if, switch statement.

**Scope**

* **var** is **function-scoped**.
* **let** is **block-scoped** (inside {} like if, for, while blocks).

**Example:**

function testVar() {

if (true) {

var x = 10;

}

console.log(x); // ✅ Works, x is accessible

}

function testLet() {

if (true) {

let y = 20;

}

console.log(y); // ❌ Error: y is not defined}

Spread operator:

The **spread operator (...)** is used to **expand** elements of an **array**, **object**, or **iterable** into individual elements. It makes copying, merging, or passing values much simpler and cleaner.

**Use Cases with Examples**

**1. Copying Arrays**

const original = [1, 2, 3];

const copy = [...original];

console.log(copy); // Output: [1, 2, 3]

**2. Merging Arrays**

const a = [1, 2];

const b = [3, 4];

const merged = [...a, ...b];

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

**3. Spreading in Function Calls**

const numbers = [5, 10, 15];

console.log(Math.max(...numbers)); // Output: 15

**4. Copying Objects**

const user = { name: "Bilal", age: 21 };

const clone = { ...user };

console.log(clone); // Output: { name: "Bilal", age: 21 }

**5. Merging Objects**

javascript

CopyEdit

const a = { x: 1 };

const b = { y: 2 };

const merged = { ...a, ...b };

console.log(merged); // Output: { x: 1, y: 2 }

**🔸 Important:**

* **Spread** is used to **expand** elements.

**Array:**

**Create Array using constructor:**

Let fruits= new Array(“apple”, “orange”, “banana”);

**Create Array using literals:**

Let fruits=[“apple”, “orange”, “banana”];

**For of loop:**

Const array=[“apple”, “orange”, “banana”];

For(let item of fruits){

Console.log(item);

}

**For in loop:**

Const array=[“apple”, “orange”, “banana”];

For(let item in fruits){

Console.log(item);

}

d/f between (for of) & (for in) loop:

for of loop print values in array.

For in loop print index of array element.

**1. forEach()**

**➤ Purpose:**

Executes a callback function once for each array element.

**➤ Syntax:**

array.forEach((element, index, array) => {

// code to run

});

**➤ Example:**

const numbers = [1, 2, 3];

numbers.forEach(num => {

console.log(num \* 2);

});

// Output: 2, 4, 6

* Does not return anything useful (undefined)
* Used for side effects (e.g., logging, updating DOM)

**✅ 2. map()**

**➤ Purpose:**

Creates a new array by applying a function to each element.

**➤ Syntax:**

const newArray = array.map((element, index, array) => {

return element \* 2;

});

**➤ Example:**

const numbers = [1, 2, 3];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // Output: [2, 4, 6]

* Returns a new array
* Used when you want to transform data

**🔄 Differences Between forEach and map**

| **Feature** | **forEach()** | **map()** |
| --- | --- | --- |
| **Return Value** | **undefined** | **Returns a new array** |
| **Usage** | **For side effects** | **For data transformation** |
| **Chaining** | **❌ Cannot be chained (returns undefined)** | **✅ Can be chained with other array methods** |
| **Performance** | **Slightly faster (no array creation)** | **Slightly slower (creates new array)** |

**🔁 Similarities**

* Both loop through each element of an array.
* Both take a callback function with up to 3 arguments: element, index, array.
* Both do not modify the original array (unless you do it manually).

**🔗 Chaining with map (and filter, etc.)**

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

const result = numbers

.map(num => num \* 2) // [2, 4, 6, 8, 10]

.filter(num => num > 5); // [6, 8, 10]

console.log(result);

**🧠 Summary:**

* Use forEach() when you’re doing something for each item (e.g., console.log).
* Use map() when you need a new transformed array.

**(CRUD) methods.**

**Push method:**

Add value at the end of an array.

Array.push(“value”) ;

**Pop method:**

remove value from the end of an array.

Array.pop () ;

**Unshift method:**

Add value at the start of an array.

Array.unshift (“value”) ;

**shift method:**

remove value from the start of an array.

Array.shift () ;

**Splice method:**

Array.splice(start no , deleting no, item that you want to add) //syntax

const studentsName=["bilal","ali","hamza","hasan","kashif"];

studentsName.splice(1,3,"noman","izhar","huzaifa");

console.log(studentsName);

**searching Methods in array:**

**indexof method:**

return starting index of an element.

Numbers.indexOf(5);

**lastIndexof method:**

return last index of an element.

Numbers.lastIndexOf(5);

**Includes method:**

If value exists return true otherwise false.

Numbers.includes(5);

**Find method:**

Find first element in array according to condition.

Numbers.find((curElm)=>{

Return curElm > 5;

});

**FindIndex method:**

Find index of first element in array according to condition.

Numbers.findIndex((curElm)=>{

Return curElm > 5;

});

**Filter:**

In JavaScript, filter() is an **array method** used to **create a new array** by selecting elements that **meet a specific condition**.

**🔹 Definition:**

array.filter(callbackFunction)

* array: the original array.
* callbackFunction: a function that tests each element. It returns true to keep the element, or false to skip it.

**🔹 Example:**

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(function(num) {

return num % 2 === 0;

});

console.log(evenNumbers);

**Sort:**

Numbers.sort();

Use compare to sort array in acending and decending order.

For acending:

Numbers.sort((a,b)=>{

If(a>b) return 1; //switch the order

If(b>a) return -1; // keep the order

});

For decending:

Numbers.sort((a,b)=>{

If(a>b) return -1; keep the order

If(b>a) return 1; switch the order

});

**Reduce method:**

In JavaScript, reduce() is an **array method** that **reduces an array to a single value** by executing a **callback function** on each element.

**🔹 Syntax:**

array.reduce(callbackFunction, initialValue)

* callbackFunction: a function that runs on each element.
* initialValue *(optional)*: the starting value for the accumulator.

**🔹 Callback Parameters:**

function callback(accumulator, currentValue, index, array)

* accumulator: stores the result from previous calls.
* currentValue: the current item being processed.
* index and array: optional.

**🔹 Example 1: Sum of Numbers**

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

const total = numbers.reduce((acc, curr) => acc + curr, 0);

console.log(total); // Output: 15

* acc starts at 0
* Each number is added to acc

**String**

**Index of method:**

Text.indexof(text,number);

**Last index of method:**

Text.lastIndexof(text,number);

**Convert string into array:**

Let text=”bilal ahmed”;

strArr=Array.from(text);

**search method:**

return index number else return -1.

Use regular expression:

Like: text.search(/Bilal/i); // ignore case sensitivity.

Let text=”bilal ahmed”;

Console.log(text.search(“bilal”));

**Match method :**

Return an array of matched value else return null:

Use regular expression:

Like: text.match (/Bilal/gi); // ignore case sensitivity.

Let text=”bilal ahmed”;

Console.log(text.match (“bilal”));

**MatchAll method :**

Return an array of matched value else return null:

Use regular expression:

Like: text.matchAll (/Bilal/gi); // ignore case sensitivity.

Let text=”bilal ahmed”;

Console.log(text.matchAll(“bilal”));

**Includes():**

Text.include(“b”);

**Startswith method:**

Return true if string start with given value else return false.

Text.startsWith(“bilal”);

**endswith method:**

Return true if string ends with given value else return false.

Text.endsWith(“bilal”);

**Slice method: (sub string)**

Return the extracted part in a new string.

Text=”hi I am bilal ahmed”;

Text.slice(starting index,ending index);

Text.slice(0,5); //output: hi i

**Replace method:**

Replace first occurrence.

let text="hi i am bilal ahmed";

console.log(text.replace("a","k"));

**ReplaceAll method:**

Replace all occurrence.

let text="hi i am bilal ahmed";

console.log(text.replaceAll("a","k"));

**charAt method:**

return single character else empty string.

text.charat(5);

**charCodeAt method:**

return asci value of character else empty string.

text.charCodeat(5);

**At method:**

return single character else empty string and also work with negative index.

text.charat(5);

**touppercase:**

convert string into uppercase:

text.toUpperCase();

**toLowercase:**

convert string into uppercase:

text.toLowerCase();

**trim method:**

remove white space from both end of the string.

Text.trim();

**Convert array into string:**

Use join method

Arrayname.join();

**String.fromCharCode() method:**

Convert asci code into character.

Code=97;

Console.log(String.fromCharCode(code)); //result = a;

**New set method :**

Remove duplicate values:

New Set (values);

**Math objects:**

**Math.random method:**

Rounds to the nearest integer.

Console.log(Math.round(4.5)); // output=5.

Console.log(Math.round(4.1)); // output=4.

**Math.floor method:**

Always Rounds down to the nearest integer.

Console.log(Math.floor (4.9)); // output=4.

Console.log(Math.floor(4.1)); // output=4.

**Math.ceil method:**

Always Rounds upto the nearest integer.

Console.log(Math.ceil (4.9)); // output=5.

Console.log(Math.ceil (4.1)); // output=5.

**Math.PI:**

Math.pi method give you pi value=3.14.

piVlaue=Math.PI;

console.log(piValue); // output=3.14159…

**Math.abs (absolute) method:**

How far the number is from zero 0 . it will be always positive

Console.log(Math.abs(-5)); output=5;

Console.log(Math.abs(3)); output=3;

**Math.trunc method:**

Return integer part of x(value). Eliminate decimal point.

Math.trunc(5.5) // output=5;

**Math.pow (power) method:**

Math.pow(2,3) // output=8

**Math.sqrt (square root) method:**

Math.pow(25) // output=5

**Math.log() method:**

Math.log(1)//output=0;

**DOM (Document Object Model):**

Example: Creating a <p> element with text and adding it to the page

// Step 1: Create the element

let para = document.createElement("p");

// Step 2: Add text to the element

para.textContent = "Hello, this is a paragraph created with JavaScript!";

// Step 3: Add the element to the page (e.g., inside the body)

document.body.appendChild(para);

para.setAttribute("class", "my-paragraph");

**Dom navigators:**

**ChildNodes / children:**

Navigate to child nodes or element.

**Syntax:**

let bodyElement=document.body;

let headElement=document.head;

console.log(bodyElement.childNodes);

console.log(headElement.children);

**firstChild / firstelEmentChild:**

Navigate to first child nodes or element.

1. let firChild=bodyElement.firstChild;

console.log(firChild);

1. let firElmChild=bodyElement.firstElementChild;

console.log(firElmChild);

**lastChild / lastElementChild:**

Navigate to last child nodes or element.

1. let lastChild=bodyElement.lastChild;

console.log(lastChild);

1. let lastElmChild=bodyElement.lastElementChild;

console.log(lastElmChild);

**nextelementsibling / prevoiusElemenetSibling:**

Navigate to nextsibling/previoussibling nodes or element.

let firstChild=document.body.firstElementChild;

let nextSibling=firstChild.nextElementSibling;

console.log(nextSibling);

let previousSibling=nextSibling.previousElementSibling;

console.log(previousSibling);

**parentelement/ parentNode:**

let parentNode=previousSibling.parentNode;

console.log(parentNode);

**Dom searching:**

**Search Element by id:**

let elementById=document.getElementById("intro");

console.log(elementById.innerHTML);

**Search Element by class:**

let elementByClass=document.getElementsByClassName("list");

console.log(elementByClass);

**Search Element by tag name:**

let elementByTagName=document.getElementsByTagName("h1");

console.log(elementByTagName);

**querySelector:**

let elementFind=document.querySelector("#intro");

console.log(elementFind.innerText);

**query Selector All:**

et findAll=document.querySelectorAll("li");

console.log(findAll);

for(elem of findAll){

    console.log(elem.innerText);

}

**Events in javascript:**

**Event listener:**

**Mouse Events:**

Event Occurs When

click The user clicks on an element

contextmenu The user right-clicks on an element

dblclick The user double-clicks on an element

mousedown A mouse button is pressed over an element

mouseenter The pointer is moved onto an element

mouseleave The pointer is moved out of an element

mousemove The pointer is moving over an element

mouseout The mouse pointer moves out of an element

mouseover The mouse pointer is moved over an element

onmouseup The mouse button is released over an element

**keyboard eventlistener:**

* keydown.
* Keyup.

**Input eventlistener:**

* input.
* change.

**Local storage:**

The localStorage object allows you to save key/value pairs in the browser.

The localStorage object stores data with no expiration date.

The data is not deleted when the browser is closed and are available for future sessions.

**1: setItem:**

Localstorage.setItem(“fruits”,”apple”);

**1: getItem:**

Localstorage.getItem(“fruits”);

**1: removeItem:**

Localstorage.removeItem(“fruits”);

**Json:**

**1. JSON.stringify()**

Converts a JavaScript object or array into a JSON string.

**✅ Example:**

const user = { name: "Bilal", age: 21 };

const jsonString = JSON.stringify(user);

console.log(jsonString); // Output: {"name":"Bilal","age":21}

**2. JSON.parse()**

Converts a JSON string into a JavaScript object.

**✅ Example:**

const jsonString = '{"name":"Bilal","age":21}';

const user = JSON.parse(jsonString);

console.log(user.name); // Output: Bilal

**Date & Time:**

new Date()

new Date(date string)

new Date(year,month)

new Date(year,month,day)

new Date(year,month,day,hours)

new Date(year,month,day,hours,minutes)

new Date(year,month,day,hours,minutes,seconds)

new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)

**examples:**

Mon May 05 2025 22:27:40 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2024","1"))

VM221:1 Thu Feb 01 2024 00:00:00 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2024","0"))

VM234:1 Mon Jan 01 2024 00:00:00 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2025","4"))

VM253:1 Thu May 01 2025 00:00:00 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2025","9","19"))

VM276:1 Sun Oct 19 2025 00:00:00 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2025","9","19","45"))

VM294:1 Mon Oct 20 2025 21:00:00 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date("2025","9","19","12","45","56"))

VM326:1 Sun Oct 19 2025 12:45:56 GMT+0500 (Pakistan Standard Time)

undefined

console.log(new Date().getTime())

VM378:1 1746466235955

**Javascript get date methods:**

**get year method:**

console.log(new Date().getFullYear())

output= 2025

**get month method:**

console.log(new Date().getMonth())

output= 4

**get day method:**

console.log(new Date().getDay())

output= 1 //return day based on week 0 to 6 Sunday=0,Monday=1;

**get date method:**

console.log(new Date().getDate ())

output= 5

**Date set methods:**

**Set year methods:**

let date=new Date();

date.setFullYear (2035);

**Set month methods:**

let date=new Date();

date.setMonth(5);

**Set Date methods:**

let date=new Date();

date.setDate (15);

**get time and date extra method:**

let date=new Date()

* let localString=date.toLocaleString()
* let localString=date.toLocaleDateString()
* let localString=date.toLocaleTimeString()

**Get Time Methods:**

Let date=new Date();

Console.log(date.getHours());

Output=22;

Console.log(date.getMinutes ());

Output=58;

Console.log(date.GetSeconds ());

Output=38;

Console.log(date.getTime());

Output= 2062000232138 //miliseconds;

**Time Set Methods:**

Date.setHours(10);

Date.setMinutes(56);

Date.setSeconds(36);

**Time based events:**

* **setTimeout()**
  + function printMessage(x){

    console.log("hi "+x);

}

let innfinite=setTimeOut (()=>printMessage(5),2000);

clearTimeOut (innfinite);

* clearTimeout()
* **setInterval()**
  + function printMessage(x){

    console.log("hi "+x);

}

let innfinite=setInterval(()=>printMessage(5),2000);

clearInterval(innfinite);

* clearInterval()

**objects in javacript:**

**define:**

Objects are a fundamental part of JavaScript, providing a way to group related data and functions together.

In JavaScript, an object is a collection of key-value pairs, where each key is a string (or a symbol) and

each value can be any data type, including other objects. Objects can have properties and methods, making

them versatile for various use cases.

**There are several ways to create objects in JavaScript. The most common one is using object literals:**

**Syntax:**

const product = {

  id: 1,

  pName: "laptop",

};

let person = {

  name: "Vinod",

  age: 30,

  isStudent: false,

  greet: function () {

    console.log("Welcome to World Best CSS Course");

  },

};

**Accessing Properties:**

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

console.log(person.age);

console.log(person.name);

console.log(person[`is'Student`]);

**Adding and Modifying Properties:**

You can add new properties or modify existing ones:

person["job"] = "web dev";

// person.age = 18;

person["age"] = 20;

**Methods:**

Methods in objects are functions associated with the object. They can be invoked using the same notation

as properties:

person.greet();

let idType = "studentId";

**We can add dynamic keys in an object:**

A dynamic key in JavaScript is a property name that is set using a variable or expression at runtime, instead of being hardcoded.

let student = {

  [`idType${1}`]: "A123456", // Dynamic key based on idType

  [`idType${2}`]:"cosc231101023",

  sName: "Vinod",

  sAge: 29,

  isStudent: true,

  greet: function () {

    console.log(

      `Hey, my ${idType} is ${student[idType]} and my name is ${student.sName}.`

    );

  },

};

**useCase**: when we want to get the user name and value in react

=================================

\* **Data Modeling:**

=================================

Data modeling is the process of creating a visual representation of either a whole information system

or parts of it to communicate connections between data points and structures. The goal is to illustrate

the types of data used and stored within the system, the relationships among these data types, the ways

the data can be grouped and organized and its formats and attributes.

Objects are excellent for modeling real-world entities. For instance, you might represent a car,

a user, or a product as an object with properties like color, brand, username, etc.

**example:**

let car = {

  brand: "Toyota",

  model: "Camry",

  year: 2022,

  start: function () {

    console.log("Engine started!");

  },

};

**Difference Between Passing by Reference and by Value in JavaScript**

In JavaScript:

* **Primitive types** (like number, string, boolean, null, undefined, symbol, bigint) are **passed by value**.
* **Objects** (including arrays and functions) are **passed by reference**.

**Pass by Value (Primitives):**

**When you pass a primitive, a copy of the value is passed. Changes do not affect the original.**

**✅ Example:**

**let a = 10;**

**function changeValue(x) {**

**x = 20;**

**}**

**changeValue(a);**

**console.log(a); // Output: 10 (original remains unchanged)**

**Pass by Reference (Objects):**

When you pass an **object**, its **reference** is passed. Changes inside the function **affect the original** object.

**✅ Example:**

let person = { name: "Bilal" };

function updateName(obj) {

obj.name = "Ahmed";

}

updateName(person);

console.log(person.name); // Output: "Ahmed" (original object is changed)

**In Short:**

* **Value**: A copy is passed. Original stays unchanged.
* **Reference**: A pointer is passed. Original can be changed.

**Object.assign() in JavaScript**

Object.assign() is a built-in method used to **copy properties** from one or more **source objects** into a **target object**.

**Syntax:**

Object.assign(target, ...sources);

 **target**: The object to receive properties.

 **sources**: One or more objects whose properties will be copied.

**Example: Cloning an Object**

const user = { name: "Bilal" };

const copy = Object.assign({}, user);

console.log(copy); // { name: "Bilal" }

**Example: Merging Multiple Objects**

const obj1 = { a: 1 };

const obj2 = { b: 2 };

const result = Object.assign({}, obj1, obj2);

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

**Important Notes:**

1. **Shallow Copy Only**  
   It copies only one level deep:

const original = { info: { age: 25 } };

const clone = Object.assign({}, original);

clone.info.age = 30;

console.log(original.info.age); // 30 (because nested object is shared)

1. **Target is modified**  
   If you use a non-empty object as the target, it will be updated.

**Use Cases:**

* Cloning objects
* Merging settings/configs
* Setting default values

**Comparison by Reference in JavaScript:**

In JavaScript, **objects (including arrays and functions)** are **compared by reference**, not by value.

**🔹 What does that mean?**

Two objects are **equal only if they reference the same memory location**, even if they contain the **same data**.

**✅ Example:**

let obj1 = { name: "Bilal" };

let obj2 = { name: "Bilal" };

let obj3 = obj1;

console.log(obj1 === obj2); // ❌ false (different references)

console.log(obj1 === obj3); // ✅ true (same reference)

* obj1 and obj2 look the same but are **not equal**.
* obj1 and obj3 point to the **same memory**, so they **are equal**.

**🧠 In Short:**

* Objects are compared by reference.
* Two objects are only equal if they point to the **exact same object in memory**.

**JSON (JavaScript Object Notation):**

**JSON** is a lightweight **data format** used for **storing and exchanging data**, especially between a server and a web application.

**🔹 Why Use JSON?**

* Easy to read and write.
* Language-independent (used in Python, Java, PHP, etc.).
* Widely used in APIs and databases.

**🔹 JSON Syntax:**

* Data is in **key/value** pairs.
* Keys are always **strings** (in double quotes).
* Values can be:
  + string, number, boolean, null, object, or array.

**✅ Example:**

json

CopyEdit

{

"name": "Bilal",

"age": 22,

"isStudent": true,

"skills": ["HTML", "CSS", "JavaScript"]

}

**🔹 Working with JSON in JavaScript**

**1. Convert JS Object → JSON (string):**

let obj = { name: "Bilal", age: 22 };

let jsonString = JSON.stringify(obj);

console.log(jsonString); // '{"name":"Bilal","age":22}'

**2. Convert JSON (string) → JS Object:**

let json = '{"name":"Bilal","age":22}';

let parsed = JSON.parse(json);

console.log(parsed.name); // Bilal

**🧠 In Short:**

**JSON** is used to **send and receive structured data** in a readable way, especially in **web APIs**.

**The this Keyword in JavaScript**

**📌 What is this?**

* this refers to **the object that is executing the current function**.
* It behaves differently depending on **how the function is called**.

**✅ When is this different?**

| **Situation** | **this refers to** |
| --- | --- |
| In a method (regular function) | The object calling the method |
| In a regular function (non-strict mode) | The global object (window in browsers) |
| In a regular function (strict mode) | undefined |
| In an arrow function | Lexical this (from surrounding scope) |
| In an event listener | The DOM element that triggered the event |
| Using call, apply, bind | The object you manually assign |

**✅ Example Code:**

**1. Global Function:**

function callme() {

console.log(this); // In browser: window object

}

callme();

**2. Regular Function in Object:**

const obj = {

name: "Kodyfier",

greet: function () {

console.log(this); // refers to obj

},

};

obj.greet();

**3. Arrow Function in Object:**

const obj = {

name: "Thapa",

greet: () => {

console.log(this); // refers to outer scope (not obj)

},

};

obj.greet(); // likely undefined or window

**✅ Object Methods Overview**

Assume this object:

const product = {

id: 1,

name: "Laptop",

price: 999.99

};

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| Object.keys() | Get all keys | Object.keys(product) → ["id", "name", "price"] |
| Object.values() | Get all values | Object.values(product) → [1, "Laptop", 999.99] |
| Object.entries() | Key-value pairs | Object.entries(product) → [["id",1],["name","Laptop"],["price",999.99]] |
| Object.hasOwnProperty() | Check if a key exists | product.hasOwnProperty("name") → true |
| Object.assign() | Merge/clone objects | Object.assign({}, product) |
| Object.freeze() | Make object read-only | Object.freeze(product) |

**✅ Interview Questions**

**1. Add Subject and Grade**

function addSubjectGrade(student, subject, grade) {

if (!student.hasOwnProperty("grades")) {

student.grades = {};

}

student.grades[subject] = grade;

}

**Example:**

let student = { name: "Ali" };

addSubjectGrade(student, "math", 95);

console.log(student);

**2. Compare Two Objects**

function areObjectsEqual(obj1, obj2) {

const keys1 = Object.keys(obj1);

const keys2 = Object.keys(obj2);

if (keys1.length !== keys2.length) return false;

for (let key of keys1) {

if (obj1[key] !== obj2[key]) return false;

}

return true;

}

**3. Transform Array to Object by ID**

function arrayToObj(arr) {

const result = {};

arr.forEach(item => {

result[item.id] = item;

});

return result;

}

**Example:**

const inputArray = [

{ id: 1, name: "Alice" },

{ id: 2, name: "Bob" },

{ id: 3, name: "Charlie" }

];

console.log(arrayToObj(inputArray));

**Output:**

{

'1': { id: 1, name: 'Alice' },

'2': { id: 2, name: 'Bob' },

'3': { id: 3, name: 'Charlie' }

}

**Ecma Script 2015 :**

* Let & const.
* Template string.
* Default arguments.
* Arrow function.
* Destructing.
* Object properties.
* Rest operators.
* Spread operators.

**1. let and const**

These are new ways to declare variables.

* let: Allows you to **change** the value later.
* const: Means **constant**, the value **cannot be changed** once set.

let name = "Bilal"; // can change later

const age = 20; // cannot change

**2. Template Strings**

Makes it easier to join strings and variables using **backticks (`)** instead of quotes.

let name = "Bilal";

console.log(`My name is ${name}`); // Output: My name is Bilal

**3. Default Arguments**

You can set a **default value** for function parameters.

function greet(name = "Guest") {

console.log(`Hello, ${name}`);

}

greet(); // Output: Hello, Guest

greet("Bilal"); // Output: Hello, Bilal

**4. Arrow Functions**

A **shorter way** to write functions. Also, it does not have its own this.

// Regular function

function add(a, b) {

return a + b;

}

// Arrow function

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

**5. Destructuring**

A way to **unpack** values from arrays or objects.

// Object destructuring

const person = { name: "Bilal", age: 21 };

const { name, age } = person;

// Array destructuring

const arr = [10, 20];

const [a, b] = arr;

2**: Ignoring elements:**

const [, , third] = numbers;

console.log(third);

**6. Object Properties (Shorthand)**

**If you have a variables and you want convert into objects and object key are same as variables you can use short hand method.**

If the **key** and **variable name** are the same, you can write it once.

let name = "Bilal";

let age = 21;

const person = { name, age }; // same as { name: name, age: age }

? **Renaming properties:**

const { name: fullName, age } = user; // Rename "name" to "fullName"

console.log(fullName);

**7. Rest Operator (...)**

Used to **collect** multiple values into one variable (usually in functions).

function sum(...numbers) {

return numbers.reduce((total, num) => total + num);

}

sum(1, 2, 3); // 6

**8. Spread Operator (...)**

Used to **spread or copy** values from arrays or objects.

// Arrays

const arr1 = [1, 2];

const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]

// Objects

const obj1 = { a: 1 };

const obj2 = { ...obj1, b: 2 }; // { a: 1, b: 2 }

**Ecma Script 2016 :**

* **Array .includes():**
* **Exponential operator(\*\*):**

Use \*\* for power of a number.

Syntax: (Base \*\* exponent)

5\*\*2=25;

**Ecma Script 2017 :**

String padding.

Object.values().

Object.entries().

Trailing commas in function parameter lists and calls.

Async functions.

**String padding:**

 String padding in JavaScript is a way to add extra characters (like spaces) to a string to make it a

specific length.

todo  Use Case: Makes formatting text easier and more predictable, especially for tables, alignments, and UI elements. No more messy, uneven lines disrupting your visual spells!

? Using padStart() to pad from the beginning:

const companyName = "Kodyfier";

? Using padEnd() to pad from the end:

todo Key points:

? Both padStart() and padEnd() create a new padded string without modifying the original one.

? They take two arguments:

?   - targetLength: The total length of the padded string.

?   - padString (optional): The string to use for padding (defaults to spaces).

? If the original string is already longer than or equal to targetLength, it's returned as-is.

**Trailing commas:**

This feature allows to have trailing commas in function declarations, functions calls, array literal & object literal:

// Object literal

const person = {

  firstName: "John",

  lastName: "Doe",

  age: 30,

};

**Ecma Script 2018:**

Spread/rest properties.

All ready done .

Promise.finally().

**Ecma Script 2019:**

Array.prototype.{flat,flatMap}.

Object.fromEntries().

String.prototype.{trimStart,trimEnd}.

Symbol.prototype.description.

Optional catch binding.

**Array.prototype.flat()**

* **What it does:** Flattens nested arrays into a single-level array.
* **Example:**

js

CopyEdit

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

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

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

**🔹 Array.prototype.flatMap()**

* **What it does:** Maps each element and flattens the result by one level.
* **Example:**

const arr = [1, 2, 3];

console.log(arr.flatMap(x => [x, x \* 2])); // [1, 2, 2, 4, 3, 6]

**🔹 Object.fromEntries()**

* **What it does:** Converts a list of key-value pairs into an object.
* **Example:**

const entries = [['name', 'Bilal'], ['age', 22]];

console.log(Object.fromEntries(entries)); // { name: 'Bilal', age: 22 }

**🔹 String.prototype.trimStart() and trimEnd()**

* **What they do:** Remove whitespace from the beginning (trimStart) or end (trimEnd) of a string.
* **Example:**

const str = ' Hello World! ';

console.log(str.trimStart()); // 'Hello World! '

console.log(str.trimEnd()); // ' Hello World!'

**🔹 Symbol.prototype.description**

* **What it does:** Gets the description of a symbol (added in ES2019).
* **Example:**

const sym = Symbol('mySymbol');

console.log(sym.description); // 'mySymbol'

**🔹 Optional catch binding**

* **What it does:** Allows catch without a parameter if the error object isn’t used.
* **Example:**

try {

// some code

} catch {

console.log('Something went wrong'); // No need for `catch (e)`

}

**Ecma Script 2020:**

BigInt

Nullish Coalescing Operator ??

Optional Chaining Operator ?.

promise.allSettled

**BigInt**

* **What it does:** Allows you to work with integers larger than Number.MAX\_SAFE\_INTEGER (2^53 - 1).
* **Syntax:** Add n to the end of an integer or use BigInt().
* **Example:**

const big1 = 9007199254740991n; // BigInt

const big2 = BigInt(123456789012345678901234567890);

console.log(big1 + 10n); // 9007199254741001n

⚠️ Note: You cannot mix BigInt and regular numbers in arithmetic operations directly.

**🔹 Nullish Coalescing Operator ??**

* **What it does:** Returns the right-hand value **only if** the left-hand value is null or undefined (not for false, 0, '').
* **Example:**

let name = null;

console.log(name ?? 'Guest'); // 'Guest'

let count = 0;

console.log(count ?? 10); // 0 (because 0 is \*not\* null/undefined)

**🔹 Optional Chaining Operator ?.**

* **What it does:** Safely accesses deeply nested properties without causing an error if something is null or undefined.
* **Example:**

const user = { profile: { name: 'Bilal' } };

console.log(user.profile?.name); // 'Bilal'

console.log(user.address?.street); // undefined (instead of error)

**Ecma Script 2021:**

String.prototype.replaceAll()

Logical Assignment Operators (||=, &&=, ??=)

Numeric Separators

Promise.any()

**String.prototype.replaceAll()**

* **What it does:** Replaces **all** occurrences of a substring or regex pattern in a string.
* **Example:**

const text = 'apple, banana, apple';

const newText = text.replaceAll('apple', 'orange');

console.log(newText); // 'orange, banana, orange'

⚠️ replace() replaces **only the first** match by default unless you use a global regex (/pattern/g), while replaceAll() replaces all without needing a regex.

**🔹 Logical Assignment Operators**

Used to assign values based on logical conditions, shorthand for common patterns:

**||= (OR assignment)**

Assigns value **if the variable is falsy** (e.g., false, 0, '', null, undefined).

let name = '';

name ||= 'Guest';

console.log(name); // 'Guest'

**&&= (AND assignment)**

Assigns value **only if the variable is truthy**.

let isLoggedIn = true;

isLoggedIn &&= 'User';

console.log(isLoggedIn); // 'User'

**??= (Nullish assignment)**

Assigns value **only if the variable is null or undefined** (not 0, false, etc.).

let age = 0;

age ??= 18;

console.log(age); // 0 (not null or undefined, so no change)

**🔹 Numeric Separators (\_)**

* **What it does:** Makes large numbers easier to read by using underscores as visual separators.
* **Example:**

const billion = 1\_000\_000\_000;

console.log(billion); // 1000000000

**Ecma Script 2022:**

**.**at() function for indexing

Object.hasOwn(obj, propKey)

**Array.prototype.at() (also works on strings)**

* **What it does:** Accesses an element at a specific index. Supports **negative indexing** to count from the end.
* **Example with array:**

const arr = [10, 20, 30, 40];

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

console.log(arr.at(-1)); // 40 (last element)

* **Example with string:**

const str = "hello";

console.log(str.at(0)); // 'h'

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

✅ More readable and safer than arr[arr.length - 1].

**🔹 Object.hasOwn(obj, propKey)**

* **What it does:** Checks if an object has a property as its **own (not inherited)**.
* **Syntax:**

Object.hasOwn(object, key)

* **Example:**

const person = { name: 'Bilal' };

console.log(Object.hasOwn(person, 'name')); // true

console.log(Object.hasOwn(person, 'toString')); // false (inherited)

✅ It's a modern and safer alternative to obj.hasOwnProperty().

**Ecma Script 2023:**

Array.findLast()

Array.findLastIndex()

Array.prototype.toReversed()

Array.prototype.toSorted(compareFn)

Array.prototype.toSpliced(start, deleteCount, ...items)

Array.prototype.with(index, value)

**Array.prototype.findLast()**

* **What it does:** Returns the **last** element that satisfies a condition.
* **Example:**

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

const lastEven = nums.findLast(x => x % 2 === 0);

console.log(lastEven); // 4

**🔹 Array.prototype.findLastIndex()**

* **What it does:** Returns the **index of the last** element that satisfies a condition.
* **Example:**

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

const index = nums.findLastIndex(x => x % 2 === 0);

console.log(index); // 3

**🔹 Array.prototype.toReversed()**

* **What it does:** Returns a **new array that is reversed**, without modifying the original.
* **Example:**

const arr = [1, 2, 3];

const reversed = arr.toReversed();

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

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

**🔹 Array.prototype.toSorted(compareFn)**

* **What it does:** Returns a **new sorted array**, leaving the original untouched.
* **Example:**

const arr = [3, 1, 2];

const sorted = arr.toSorted();

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

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

**🔹 Array.prototype.toSpliced(start, deleteCount, ...items)**

* **What it does:** Works like .splice(), but returns a **new array** without changing the original.
* **Example:**

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

const newArr = arr.toSpliced(1, 2, 9, 8); // removes 2 items at index 1, inserts 9 and 8

console.log(newArr); // [1, 9, 8, 4]

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

**🔹 Array.prototype.with(index, value)**

* **What it does:** Returns a new array where the element at index is replaced by value.
* **Example:**

const arr = [1, 2, 3];

const newArr = arr.with(1, 9); // replaces index 1 with 9

console.log(newArr); // [1, 9, 3]

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

**Event propagation:**

**Event propagation** in JavaScript refers to the way events travel through the **DOM (Document Object Model)** when an event occurs. It involves **three phases**:

**🔁 The Three Phases of Event Propagation:**

1. **Capturing Phase (Capture/Bubbling Down)**
   * The event starts from the **window** and goes **down the DOM tree** to the target element.
   * Also called "event capturing".
2. **Target Phase**
   * The event reaches the **target element** (the actual element that was clicked or interacted with).
3. **Bubbling Phase (Bubble Up)**
   * After reaching the target, the event **bubbles back up** from the target element to the **root** of the DOM (usually document or window).

**🧠 Example:**

<div id="parent">

<button id="child">Click Me</button>

</div>

<script>

document.getElementById("parent").addEventListener("click", () => {

alert("Parent clicked!");

});

document.getElementById("child").addEventListener("click", () => {

alert("Child clicked!");

});

</script>

If you click the button:

* **Child handler runs first**, then **parent** — because bubbling is the default phase used.

**🧩 addEventListener Third Parameter**

element.addEventListener("click", handler, true); // true = capturnding

* true → Use **capturing phase**
* false (default) → Use **bubbling phase**

**🛑 Stop Propagation**

* event.stopPropagation() → Prevents event from moving to the next element in the propagation chain.
* event.stopImmediatePropagation() → Stops all other listeners on the same element from being called.

**Event delegation:**

**Event Delegation** is a technique in JavaScript where **a single event listener is added to a parent element** instead of each child element. This works because of **event bubbling** — events triggered on child elements bubble up to their parent.

**✅ Why use event delegation?**

* **Performance:** Fewer event listeners = better performance.
* **Dynamic elements:** Useful when adding/removing elements dynamically.
* **Cleaner code:** One handler can manage many child elements.

**🧠 Example:**

Imagine this HTML:

<ul id="myList">

<li>JavaScript</li>

<li>HTML</li>

<li>CSS</li>

</ul>

Instead of this:

document.querySelectorAll('#myList li').forEach(item => {

item.addEventListener('click', () => {

console.log('Clicked:', item.textContent);

});

});

Use **event delegation**:

document.getElementById('myList').addEventListener('click', function (e) {

if (e.target.tagName === 'LI') {

console.log('Clicked:', e.target.textContent);

}

});

**🔍 How it works:**

* You add the listener to <ul> (the parent).
* When a <li> is clicked, the event bubbles up to <ul>.
* Inside the handler, you use e.target to check if the clicked element is an <li>.

**🚀 Real-World Benefit:**

If new <li> items are added dynamically later, you **don’t need to attach new listeners**. The parent still handles all clicks.

**1. First-Class Functions (Think: function = value)**

In JavaScript, **functions can be stored in variables** just like numbers or strings.

**🔹 Example:**

let sayHello = function() {

console.log("Hello!");

};

sayHello(); // Output: Hello!

**📌 What's happening?**

* We **store** a function inside the variable sayHello.
* Then we **call it** using sayHello().

👉 This is what we mean by "functions are first-class" — you can treat them like normal values.

**✅ 2. Callback Functions (Think: function passed to another function)**

A **callback** is a function that you **give to another function**, and it gets called **later**.

**🔹 Example:**

function greet(name, callback) {

console.log("Hi " + name);

callback(); // calling the function passed as an argument

}

function sayBye() {

console.log("Goodbye!");

}

greet("Bilal", sayBye);

**📌 What’s happening?**

* greet takes 2 things: a name and a function (called callback).
* We pass sayBye as the second thing.
* Inside greet, it says "Hi Bilal", then runs sayBye() — so it says "Goodbye!".

**✅ 3. Higher-Order Functions (Think: function that accepts or returns another function)**

A **higher-order function** is a function that:

* **Takes another function as input**, or
* **Gives back a function as output**

**🔹 Example: Takes a function as input**

function doTwice(action) {

action(); // run it once

action(); // run it again

}

function sayHello() {

console.log("Hello!");

}

doTwice(sayHello);

🧠 Output:

Hello!

Hello!

**🔹 Example: Returns a function**

function add(x) {

return function(y) {

return x + y;

};

}

let addFive = add(5); // gives back a new function that adds 5

console.log(addFive(3)); // 8

📌 What's happening?

* add(5) gives back a function.
* addFive(3) is like saying: 5 + 3 = 8

**🧠 Summary in Simple Words:**

| **Concept** | **Easy Meaning** |
| --- | --- |
| First-Class Function | Store function in variable like text or number |
| Callback Function | Give a function to another function as input |
| Higher-Order Function | A function that takes or gives another function |

**Callback Hell**

**What is it?**

**Callback Hell** happens when you write too many **nested callbacks** (functions passed as arguments to other functions), making your code hard to read and maintain.

**Example:**

getData(function(a) {

getMoreData(a, function(b) {

getEvenMoreData(b, function(c) {

processData(c, function(d) {

console.log(d);

});

});

});

});

This is hard to follow — it’s like a pyramid of doom 😵.

**Problem:**

* Hard to read
* Difficult to debug
* Unmanageable when scaling

**✅ Promises**

**What is it?**

A **Promise** is a cleaner way to deal with asynchronous operations. It represents a value that will be available **in the future**.

A Promise can be:

* **Pending** – still waiting
* **Resolved** – successful
* **Rejected** – failed

**Example:**

getData()

.then(a => getMoreData(a))

.then(b => getEvenMoreData(b))

.then(c => processData(c))

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

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

Much cleaner! 🎯

**Benefits:**

* Easy to read
* Chainable
* Better error handling using .catch()

**Summary Table**

| **Feature** | **Callbacks** | **Promises** |
| --- | --- | --- |
| Syntax | Nested functions | Chained .then() |
| Readability | Poor (Callback Hell) | Better |
| Error Handling | Manual (each level) | .catch() block |
| Scalability | Hard to maintain | Easy to manage |

**Promise.all()**

**✅ What it does:**

* Takes an array of promises.
* Waits for **all** promises to be resolved.
* If **any** promise fails (rejects), it throws an error.

**✅ Example:**

let p1 = Promise.resolve(1);

let p2 = Promise.resolve(2);

let p3 = Promise.resolve(3);

Promise.all([p1, p2, p3])

.then(results => console.log(results)) // [1, 2, 3]

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

If one promise fails:

let p1 = Promise.resolve(1);

let p2 = Promise.reject("Error!");

Promise.all([p1, p2])

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

.catch(error => console.log(error)); // "Error!"

**🏁 Promise.race()**

**🏁 What it does:**

* Takes an array of promises.
* Returns the **first** promise that settles (either resolves or rejects).

**🏁 Example:**

let p1 = new Promise(resolve => setTimeout(resolve, 1000, "One"));

let p2 = new Promise(resolve => setTimeout(resolve, 500, "Two"));

Promise.race([p1, p2])

.then(result => console.log(result)); // "Two" (faster one)

If the first promise to settle is a rejection:

let p1 = new Promise((\_, reject) => setTimeout(reject, 500, "Failed"));

let p2 = new Promise(resolve => setTimeout(resolve, 1000, "Success"));

Promise.race([p1, p2])

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

.catch(error => console.log(error)); // "Failed"

**🧘 Promise.allSettled()**

**🧘 What it does:**

* Takes an array of promises.
* Waits for **all** of them to settle (resolve or reject).
* Always resolves with an array of result objects: {status: "fulfilled"} or {status: "rejected"}.

**🧘 Example:**

let p1 = Promise.resolve("Success");

let p2 = Promise.reject("Error");

Promise.allSettled([p1, p2])

.then(results => console.log(results));

Output:

[

{ status: "fulfilled", value: "Success" },

{ status: "rejected", reason: "Error" }

]

This is useful when you want to know **all results**, even if some failed.

**🧠 Summary**

| **Method** | **Waits for All** | **Rejects Early** | **Gives All Results** |
| --- | --- | --- | --- |
| Promise.all() | ✅ | ❌ (fails if one fails) | ❌ |
| Promise.race() | ❌ (only first settles) | ✅ | ❌ |
| Promise.allSettled() | ✅ | ❌ | ✅ |

**What Are async and await?**

They are modern JavaScript keywords used to **work with Promises more easily** and write **asynchronous code** that looks like normal (synchronous) code.

**✅ async Keyword**

* You put async before a function to make it return a **Promise** automatically.

async function myFunction() {

return "Hello!";

}

myFunction().then(result => console.log(result)); // "Hello!"

**⏳ await Keyword**

* You use await **inside an async function**.
* It **waits for a Promise to resolve**, then gives you the result.
* It makes your code pause at that line until the Promise is done.

**🧪 Example with Delay:**

function delay() {

return new Promise(resolve => {

setTimeout(() => {

resolve("Waited 2 seconds!");

}, 2000);

});

}

async function myAsyncFunction() {

console.log("Start");

const result = await delay(); // Waits here

console.log(result); // After 2 seconds

console.log("End");

}

myAsyncFunction();

**Output:**

Start

(wait 2 seconds)

Waited 2 seconds!

End

**💥 Handling Errors**

Use try and catch inside async functions:

async function fetchData() {

try {

let response = await fetch("https://api.example.com/data");

let data = await response.json();

console.log(data);

} catch (error) {

console.log("Error occurred:", error);

}

}

**🔁 Summary Table**

| **Keyword** | **Purpose** | **Notes** |
| --- | --- | --- |
| async | Makes a function return a Promise | Must be used before using await |
| await | Pauses function until Promise resolves | Only used **inside** async functions |