Q. What is JavaScript?
JavaScript (JS) is a programming language used to make websites interactive and dynamic.
JavaScript runs inside the browser (Chrome, Edge, Firefox) and controls what the user sees and what happens when they click or type.
Easy to learn
Runs in every browser
Event-based (reacts to clicks, typing)
Asynchronous (can fetch data in background)
Huge ecosystem (npm packages)

Q. Is JavaScript compiled or interpreted?
JavaScript is neither purely compiled nor purely interpreted.
It is a Just-In-Time (JIT) compiled language.

Modern browsers (Chrome, Edge, Node.js) use engines like:
V8 Engine (Chrome, Node.js)
SpiderMonkey (Firefox)
They use JIT Compilation (Just-In-Time).

How JS runs internally
Step 1: You write JS
Step 2: Engine reads code
Step 3: Converts to bytecode
Step 4: Frequently used code → compiled into machine code
Step 5: Runs very fast

Q. How does JavaScript run in the browser?
Step 1: Browser loads the page
The browser downloads:
HTML file
CSS files
JavaScript files

Step 2: HTML is parsed → DOM is created
The browser reads HTML top → bottom and converts it into a structure called:
DOM (Document Object Model)
It's basically a tree of objects the JavaScript can control.

Step 3: JavaScript Engine starts (Very Important ⍰)
Inside the browser there is a JavaScript Engine.

Chrome uses V8 engine.
The engine has 2 main parts:

Memory Heap

Stores variables and objects.

```
let name = "Nikhil";
```

Call Stack

Where functions execute.

```
function greet(){
  console.log("Hello");
}
greet();
```

Step 4: Problem — JavaScript is Single-Threaded

JavaScript can do only ONE task at a time.

So what about:

API calls

setTimeout

button clicks

fetch requests

They take time ▯

If JS waited for them → the page would freeze.

This is where the browser helps.

Step 5: Browser Web APIs (Secret helpers)

The browser provides special background features called:

Web APIs

setTimeout

fetch

DOM events (click, input)

Example:

```
setTimeout(() => {
  console.log("Hello");
}, 3000);
```

JS does NOT wait 3 seconds.

Instead:

JS sends timer to browser

Browser handles waiting

JS continues executing next code

Step 6: Callback Queue

When the timer finishes (3s), the browser sends the function to:
 Callback Queue
It waits there.

Step 7: Event Loop
The Event Loop constantly checks:
“Is the Call Stack empty?”
If NO → wait
If YES → push callback from queue into stack
Then the function finally runs.
This is why JS feels asynchronous even though it is single-threaded.

Q.What is ECMAScript?

CMAScript is the official standard (rule book) of JavaScript.
JavaScript is the language you write,
ECMAScript is the specification that defines how the language must work.

Q.What are the features of JavaScript?

1� High-Level Language
You don’t manage memory manually (like C/C++).
You simply write:
let name = "Nikhil";
JS automatically allocates and frees memory (garbage collection).

2� Interpreted + JIT Compiled
JavaScript runs directly in the browser.
Modern engines compile frequently used code using JIT (Just-In-Time) compilation → faster performance.

3� Dynamically Typed
You don’t declare data types.
let x = 10;   // number
x = "hello";   // now string (allowed)

4� Prototype-Based Object Oriented
Instead of traditional classes (like Java), JavaScript uses prototypes.

```
function Person(name){
  this.name = name;
}
```

Objects can inherit from other objects.
(ES6 later introduced class syntax, but internally it still uses prototypes.)

5️⃣ First-Class Functions (VERY IMPORTANT □)
Functions are treated like variables.
You can:
store in a variable
pass as argument
return from another function

```
const greet = function(){
  console.log("Hello");
};
```

This is why callbacks, promises, and React all work.

6️⃣ Event-Driven Language
JavaScript reacts to user actions:
click,typing,scrolling,submit
Websites feel interactive because of this.

7️⃣ Asynchronous & Non-Blocking
JavaScript does not stop the page while waiting for slow tasks.
It uses:
Callbacks
Promises
async/await

```
fetch("/api/data")
  .then(res => res.json())
  .then(data => console.log(data));
```

The UI keeps working while data loads.

8️⃣ Cross-Platform
JavaScript runs everywhere:
Browser (Frontend)
Server (Node.js)
Mobile apps (React Native)
Desktop apps (Electron)
One language → many platforms.

9️⃣ Case-Sensitive

Variables with different casing are different:
let name = "Nikhil";
let Name = "Rahul";

Q.What are primitive data types in JavaScript?
Primitive data types are basic, single values stored directly in memory (immutable=>You cannot change them directly.).
1.Number
Represents all numbers (integer + decimal).

2.String
Text data (written in quotes).

3.Boolean
Only two values → true or false
let isLoggedIn = true;
let isAdmin = false;
Mostly used in conditions.

4.Undefined
A variable declared but not assigned a value.
let x;
console.log(x); // undefined

5.Null
Represents intentional empty value (you assign it).
let data = null;
Meaning: “I purposely set no value”.

6.BigInt
Used for very large numbers (beyond normal number limit).
let bigNumber = 12345678901234567890n;

7.Used to create unique identifiers (mostly advanced / frameworks).
let id = Symbol("userId");
Every Symbol is unique.

Q.What are reference data types?
Reference data types are values that are stored in memory by reference (address), not by actual value.
Instead of holding the real data, the variable stores a pointer to a location in memory.
Also called: Non-primitive types ,Objects

JavaScript mainly has 3 common reference types:

Object

A collection of key-value pairs.

```
let user = {
  name: "Nikhil",
  age: 21
};
```

user does not store the object directly →
it stores the address where the object exists in memory.

Array

A list-like object storing multiple values.

```
let colors = ["red", "green", "blue"];
```

Arrays are actually a special type of object in JavaScript.

Function

Functions are also objects in JS.

```
function greet(){
  console.log("Hello");
}
```

Yes — in JavaScript, a function is a value you can store in a variable.

Example:

```
Primitive                    Reference
let a = 10;                  let obj1 = { name: "Nikhil" };
let b = a;                   let obj2 = obj1;
b = 20;                      obj2.name = "Rahul";
console.log(a); // 10        console.log(obj1.name); // Rahul
```

Q. What is dynamic typing?

Dynamic typing is a feature of JavaScript where variables are not bound to a fixed data type and their type is determined and can change during runtime based on the assigned value.

```
let value = 10;      // number
value = "Hello";     // now string
value = true;        // now Boolean
```

Same variable → different types ▯
This is dynamic typing.

Q. What is weak typing?

Weak typing means JavaScript automatically performs type coercion, converting values from one data type to another during operations instead of throwing type errors.

```
JS code:                output
console.log(5 + "5");        "55"
```

Number 5 is automatically converted to string "5"
→ then concatenation happens.
JS silently changed the type = weak typing

Q. Difference between var, let, and const
All three are used to declare variables in JavaScript, but they behave very differently.

Var
```
var name = "Nikhil";
```
Properties
- Function scoped (not block scoped)
- Can be redeclared
- Can be reassigned
- Hoisted and initialized with undefined
- Causes many bugs

```
if(true){
  var x = 10;
}
console.log(x); // 10  (still accessible)
```

Let
```
let age = 21;
```
Properties
- Block scoped { }
- Cannot be redeclared in same scope
- Can be reassigned
- Hoisted but not initialized (Temporal Dead Zone)

```
if(true){
  let y = 20;
}
console.log(y); //  ReferenceError
```

Const
```
const pi = 3.14;
```
Properties
- Block scoped
- Cannot be redeclared
- Cannot be reassigned

- Must be initialized immediately

Feature	Var	Let	Const
Scope	Function	Block	Block
Redeclare	Yes	No	No
Reassign	Yes	Yes	No
Hoisting	Yes(undefined)	Yes(TDZ)	Yes(TDZ)
Recommended	No	Yes	Best

Q. What is hoisting?
Hoisting is JavaScript's behavior where variable and function declarations are moved to the top of their scope during the creation phase, allowing them to be referenced before their actual declaration (with var as undefined, while let and const remain in the temporal dead zone).

Example 1 — var hoisting   Output

```
console.log(a);            //undefined
var a = 5;
```

Why not error?
Because JavaScript internally treats it like:

```
var a;         // hoisted
console.log(a); // undefined
a = 5;
```

So var is hoisted and initialized with undefined.

Example 2 — let and const

```
console.log(b);                  Output
let b = 10;                    //ReferenceError
```

But wait… let is also hoisted!
Yes — but not initialized(Because b exists but is uninitialized and protected)
This period is called:
Temporal Dead Zone (TDZ)

Example 3 — const

```
console.log(c);
const c = 20;
```

Also:

ReferenceError
Because const also lives in the TDZ.

Function Hoisting

Function Declaration   Output:

```
greet();

function greet(){
  console.log("Hello");
}                               Hello
```

Why?
Because entire function is hoisted.
Internally:

```
function greet(){
  console.log("Hello");
}
greet();
```

Function Expression   **Output**

```
greet();                        TypeError
var greet = function(){
  console.log("Hello");
}
```

Because only the variable greet is hoisted, not the function.

Q. Temporal Dead Zone (TDZ) — what is it?

The Temporal Dead Zone is the period between entering a scope and the declaration of a let or const variable where the variable is hoisted but cannot be accessed until it is initialized.

```
console.log(a);   //  TDZ
let a = 10;       // initialization happens here
console.log(a);   //  works
```

Q. What is global scope?

Global scope is the outermost scope in JavaScript.
A variable declared outside any function or block is accessible from anywhere in the program.
In simple words:

If a variable is created at the top level of the file, every function and block can use it.

```
let age = 21;
function one(){
  console.log(age);
}
function two(){
  console.log(age);
}
one();
two();
```

Q. What is Function scope?
Function scope means a variable declared inside a function can only be accessed within that function, not outside it.

```
Example:                  Output

function test(){            Hello
   let msg = "Hello";        Reference Error
   console.log(msg);
}

test();
console.log(msg);


let a = 5;   // Global
function show(){
   console.log(a);
}
show();
```

Q. What is Block Scope?
Block scope means a variable is accessible only inside the pair of curly braces { } where it is declared.

A block is any code written inside { }, like:
if, for, while ,switch, standalone { }

let and const are Block Scoped
They stay confined to the block where they are created.

```
{
   let x = 5;
   console.log(x); // □ works
}

console.log(x); //  error
```

var is NOT Block Scoped (Very Important)

```
if(true){
   var y = 50;
}
console.log(y); // 50
```

var ignores { } blocks and escapes outside.
This is one of the biggest problems with var.

Q. Can const object properties be modified?
Yes □ — properties of a const object CAN be changed.
But the object itself cannot be reassigned.

Because const protects the reference (memory address), not the internal data.

```
const user = {
  name: "Nikhil",
  age: 21
};
user.age = 22;    □ allowed
console.log(user)
```

Output

```
{ name: "Nikhil", age: 22 }
```

Q. What happens if you redeclare let?
You get a SyntaxError. JavaScript does NOT allow the same let variable to be declared again in the same scope.

```
let a = 10;
let a = 20;
SyntaxError: Identifier 'a' has already been declared
```

Why JavaScript forbids this
To prevent accidental overwriting bugs that used to happen with var.
With var:

```
var x = 5;
var x = 10;   // □ allowed
console.log(x);
```

Output:

```
10
```

Q. What is variable shadowing?

Variable shadowing happens when a variable declared in an inner scope has the same name as a variable in an outer scope, hiding (shadowing) the outer variable inside that block.

```
let name = "Nikhil";   // outer (global)
function show(){
  let name = "Rahul"; // inner (function scope)
  console.log(name);
}

show();
console.log(name);
```

Output

```
Rahul
Nikhil
```

Q. Difference between null and undefined

undefined means a variable exists but no value has been assigned yet (automatic by JavaScript).

null means intentional absence of value (given by the programmer).
You are explicitly saying:
“I know this variable should have a value, but right now it has none.”

Features	Undefined	Null
Who assigns	JavaScript	Programmer
Meaning	Value not assigned	Intentionally empty
Type	Undefined	Object(historical bug)
Default value	Yes	No
Strict equality	different	Different

Q. Difference between == and ===

Operator	Name	What it checks
= =	Loose equality	value only (after type conversion)

= = =	Strict equality	value AND type

Q. What is NaN?
NaN is a special numeric value in JavaScript representing the result of an invalid or undefined mathematical operation, and it is of type number but not equal to itself

```
console.log(0 / 0);      // NaN
console.log("hello" * 5);   //NaN
```

Q. Type coercion — what is it?
Type coercion is JavaScript's automatic or explicit conversion of values from one data type to another during operations or comparisons to make them compatible.

```
console.log(5 + "5");  //"55"
```

Q. What is typeof operator?
The typeof operator is used to determine the data type of a variable or value in JavaScript and returns the type as a string.

```
let name = "Nikhil";
console.log(typeof name);        "string"
```

Q. What is the in operator?
The in operator is used to check whether a specified property exists in an object (including its prototype chain) and returns a boolean value.

```
const user = {                          output
  name: "Nikhil",                         true
  age: 21
};

console.log("name" in user);
```

Q. What is a function declaration?
A function declaration is a named function defined using the function keyword that is hoisted and can be invoked before its definition in the code.
Function declarations are fully hoisted.
You can call them before writing them.

```
function add(a, b){                 output
   return a + b;                  5
}
```

```
console.log(add(2,3));
```

Q. What is a function expression?
A function expression is a function that is created and stored inside a variable.
Instead of giving the function a direct declaration, we assign it to a variable.
Function expressions are NOT fully hoisted.

```
const greet = function(){                    output
  console.log("Hello Nikhil");                 Hello Nikhil
};
greet();
```

Q. What are arrow functions?
Arrow functions are a concise way to write functions introduced in ES6. They use the => syntax and do not have their own this; instead, they inherit this from the surrounding scope.

Normal function:

```
function add(a, b){
  return a + b;
}

const add = (a, b) => {
  return a + b;
}
```

Arrow functions do NOT have their own this
They use this from their parent (lexical scope).

Where Arrow Functions Are Used
map(),filter(),reduce(),event callbacks,promises

Example:

```
const nums = [1,2,3];
const doubled = nums.map(n => n * 2);
```

Q. Arrow vs normal function
Arrow functions are a shorter syntax for functions and inherit this from their surrounding scope, whereas normal functions have their own this, support hoisting, the arguments object, and can be used as constructors.

Feature	Normal Function	Arrow Function
Syntax	Longer	Short
`this`	Dynamic	Lexical
Hoisting	Yes (declaration)	No
arguments	Available	Not available
Constructor	Yes	No
Best use	Objects, methods	Callbacks, short functions

Q. What is a callback function?
A callback function is a function passed as an argument to another function that is executed after the completion of a task, commonly used for asynchronous operations in JavaScript.

```
function greet(name){
  console.log("Hello " + name);
}

function processUser(callback){
  let username = "Nikhil";
  callback(username);
}

processUser(greet);
```

```
Output:
   Hello Nikhil
```

greet → callback function
processUser → main function
greet runs later inside it

Problem with callack function is that it creates call back hell that's why we need async/await

Q. What is a higher-order function?
A higher-order function is a function that takes another function as a parameter or returns a function as its result, enabling reusable and flexible code.

Function passed as argument (callback)

```
function greet(name){
  return "Hello " + name;
}
function processUser(callback){
  console.log(callback("Nikhil"));
}
```

```
processUser(greet);
```

greet → callback
processUser → higher-order function
Because it receives a function.

Function returning another function

```
function multiplier(x){
  return function(y){                       output:10
    return x * y;
  };
}

const double = multiplier(2);
console.log(double(5));
```

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

const result = nums.map(function(n){
  return n*2;                          Output:[2,4,6]
});

console.log(result);
```

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

const even = nums.filter(n => n%2===0);            output:[2,4]

console.log(even);
```

Q. What is a pure function?

A pure function is a function that:
Always returns the same output for the same input
Does NOT modify anything outside the function (no side effects)

```
function add(a, b){
 return a + b;
}
add(2,3); // 5
add(2,3); // always 5
```

Why pure?
Depends only on inputs a and b
Does not change any external variable

Example of impure function:

```
let total = 0;
function addToTotal(x){
  total += x;
  return total;
}
```

Why impure?
It modifies an outside variable (total)
Output changes every time

Q.What is an IIFE?
An IIFE (Immediately Invoked Function Expression) is a function expression that executes immediately after it is defined, commonly used to create a private scope and avoid polluting the global namespace.

```
(function(){                    Output
   let secret = 123;              123
   console.log(secret);
})();

console.log(secret);              Reference error
```

Q.What are default parameters?
Default parameters allow function parameters to take a predefined value when an argument is not provided or is undefined, improving code safety and readability.

```
function greet(name = "Guest"){
  console.log("Hello " + name);
}                               Output:

greet();                          Hello Guest
```

When you pass a value

```
greet("Nikhil");                    Hello Nikhil
```

Default value is used only when the argument is undefined.

Q.What are rest parameter?
The rest parameter (...) allows a function to accept a variable number of arguments and stores them as an array in a single parameter. or

The rest parameter allows a function to accept any number of arguments and collect them into a single array.

```
function sum(...numbers){
   console.log(numbers);                    Output:[1,2,3,4]
}

sum(1,2,3,4);


function sum(...nums){
  let total = 0;
  for(let n of nums){
     total += n;
  }                                   Output:15
  return total;
}

console.log(sum(1,2,3,4,5));
```

Q.What are spread parameter?
The spread operator (...) expands elements of an iterable (like an array or object) into individual values, commonly used for copying, merging, and passing arguments.

```
const arr = [1,2,3];
console.log(...arr);            // 1 2 3
------------------------------------------------
const a = [1,2];
const b = [3,4];
const c = [...a, ...b];
console.log(c);                 //[1,2,3,4]
-------------------------------------------------
const user = { name:"Nikhil", age:21 };
const updatedUser = {...user, city:"Ranchi"};
console.log(updatedUser);

output
{ name:"Nikhil", age:21, city:"Ranchi" }
```

Q.What is recursion?
Recursion is a programming technique where a function calls itself repeatedly until a base condition is met to solve a problem in smaller subproblems.

```
function factorial(n){
   if(n === 1) return 1;        // base case
   return n * factorial(n-1);   // recursion
}

console.log(factorial(5));                  //120

function print(n){
 if(n===0) return;
 console.log(n);
 print(n-1);
}
```

Q.What is currying?

Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each take one argument at a time.

Instead of:	It becomes:
f(a, b, c)	f(a)(b)(c)

Normal Function

```
function add(a, b){
  return a + b;
}

console.log(add(2,3));                //5
--------------------------------------------------------
function add(a){
  return function(b){
     return a + b;
  }
}

console.log(add(2)(3));                //5
------------------------------------------------------------
const add = a => b => a + b;

console.log(add(5)(4));                //9
-----------------------------------------------------------
const multiply = a => b => a * b;

const double = multiply(2);
const triple = multiply(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Q.What is function borrowing?
Function borrowing means using a method of one object with another object.
In JavaScript, objects can “borrow” a function from another object and execute it as if it were their own.
We do this using: call(),apply(),bind()

```
const person1 = {
  name: "Nikhil",
  greet: function(){
     console.log("Hello " + this.name);
  }
};

const person2 = {
  name: "Rahul"
};
//Using call()                                    // Output
person1.greet.call(person2);                      // Hello Rahul
```

Q.What is memoization?
Memoization is an optimization technique where function results are stored and reused for the same inputs to avoid repeated computations and improve performance.

Q.What does this refer to in global scope?

this is a special keyword that refers to the object that is currently executing the function.

```
const user = {
  name: "Nikhil",
  greet(){
    console.log(this.name);
  }
};

user.greet();          //Nikhil
```

this in Global Scope
In the global scope (browser), this refers to the global object → window.

```
console.log(this);         //window
```

Q.What is this inside an object method?

```
const user = {
  name: "Nikhil",
  greet: function(){
    console.log(this.name);
  }
};

user.greet();              //Nikhil
```

this refers to the object that called the method (user).

Q.What is this inside an Arrow Function?
Arrow functions do not have their own this.
They inherit this from the surrounding scope (lexical this).

```
const user = {
  name: "Nikhil",
  greet: () => {
    console.log(this.name);
  }
};

user.greet();             //undefined
```

Q.What is this inside an Event Listener?

```
button.addEventListener("click", function(){
  console.log(this);
});     //this refers to the HTML element that was clicked.
```

Q.What is call()?
The call() method is a predefined JavaScript method. It can be used to invoke (call) a method with an owner object as an argument (parameter). This allows borrowing methods from other objects, executing them within a different context, overriding the default value, and passing arguments.

JavaScript Function Call Examples
Example 1: In this example, we defines a product() function that returns the product of two numbers. It then calls product() using call() with `this` as the context (which is typically the global object), passing 20 and 5 as arguments. It logs the result, which is 100

```
// function that returns product of two numbers
function product(a, b) {
   return a * b;
}

// Calling product() function
```

```
let result = product.call(this, 20, 5);                //Output 100

console.log(result);
```

Example 2: This example we defines an object "employee" with a method "details" to retrieve employee details. Using call(), it invokes "details" with "emp2" as its context, passing arguments "Manager" and "4 years", outputting the result.

```
let employee = {
   details: function (designation, experience) {
      return this.name
         + " "
         + this.id
         + designation
         + experience;
   }
}

// Objects declaration
let emp1 = {
   name: "A",
   id: "123",
}
let emp2 = {
   name: "B",
   id: "456",
}
let x = employee.details.call(emp2, " Manager ", "4 years");
console.log(x);
```

Output

```
B 456 Manager 4 years
```

Q.What is apply()?
The apply() method is used to write methods, which can be used on different objects. It is different from the function call() because it takes arguments as an array.

```
let student = {
   details: function (section, rollnum) {
      return this.name + this.class
         + " " + section + rollnum;
   }
}
```

```
let stud1 = {
  name: "Dinesh",
  class: "11th",
}
let stud2 = {
  name: "Vaibhav",
  class: "11th",
}

let x = student.details.apply(stud2, ["A", "24"]);
console.log(x);
```

Output:

```
Vaibhav
11th A
24
```

Q.What is bind()?
bind() creates a new function with a fixed this value, but it does NOT execute the function immediately.

Call() vs apply()

Feature	call()	apply()
Invokes function	Immediately	Immediately
`this` control	Yes	Yes
Arguments	comma-separated	array
Usage	normal params	when arguments already in array

Q.What is a Closure?
A closure is created when a function remembers and can access variables from its outer (parent) scope even after the outer function has finished executing.

```
function outer(){
 let name = "Nikhil";

 function inner(){
  console.log(name);
 }

 return inner;
}
```

```
const fn = outer();
fn();                              //Nikhil
```

Q.Why are Closures used?
Closures are used whenever we need:
persistent memory
private variables
function factories
callbacks

Q.What are Advantages of Closures?
□ Data privacy (private variables)
□ Persistent state (remember values)
□ Function factories
□ Useful in callbacks & async code
□ Used heavily in React hooks

Q. What are Disadvantages of Closures?
Extra memory usage
Can cause memory leaks if misused
Harder to debug sometimes
Variables not garbage-collected quickly

Q.What are objects in JS?
An object is a collection of related data and functions stored as key–value pairs.

```
const user = {
  name: "Nikhil",
  age: 21,
  isStudent: true,
  greet(){
    console.log("Hello");
  }
};
```

Key	Value
name	"Nikhil"
age	21
isStudent	true
greet	function(method)

Q.How to create an object?
1.Object Literal (Most Common)

```
const user = {
  name: "Nikhil",
  age: 21,                              ✅ Simplest
  greet(){                              ✅ Most used
    console.log("Hello");
  }
};
```

2.Using new Object()

```
const user = new Object();
user.name = "Nikhil";
user.age = 21;                          Same result, but rarely used.
```

3.Constructor Function

```
function Person(name, age){
  this.name = name;
  this.age = age;
}

const p1 = new Person("Nikhil",21);
```

Here new:
creates a new object
binds this to it

4.Using ES6 Class

```
class Person{
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
}

const p1 = new Person("Nikhil",21);          //This is modern and commonly used.
```

5.Object.create()

```
const userProto = {
  greet(){
    console.log("Hello");
  }
};

const user = Object.create(userProto);
user.name = "Nikhil";
```

Q.What are object methods?

Object methods are functions defined as properties of an object that operate on the object’s data and are invoked using dot notation.

```
const user = {
  name: "Nikhil",
  age: 21,

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

user.greet();                         //Hello Nikhil
```

Q.What are optional chaining?
Optional chaining (?.) is a JavaScript operator that safely accesses nested object properties and returns undefined instead of throwing an error when a reference is null or undefined.

Why it is useful
Very common when handling:
API responses
database data
optional fields

```
const user = {
  profile: {
    contact: {
      email: "nikhil@gmail.com"
    }
  }
};

console.log(user.profile?.contact?.email);
```

Q.What is object freezing?
Object freezing is the process of making an object immutable using Object.freeze(), which prevents adding, deleting, or modifying its properties (shallowly)

```
const user = {                       Output:
  name: "Nikhil",                      { name: "Nikhil", age: 21 }
  age: 21
};
Object.freeze(user);
user.age = 25;     // ignored
user.city = "Patna"; // ignored
delete user.name;    // ignored
console.log(user);
```

JavaScript does not throw an error in normal mode —
it just silently ignores changes.

(In strict mode, it throws an error.)

Object.freeze() only freezes the top level, not nested objects.

```
const user = {
  name: "Nikhil",
  address: {
    city: "Ranchi"
  }
};

Object.freeze(user);

user.address.city = "Delhi"; // □ STILL CHANGES
console.log(user.address.city);
```

Because the inner object wasn’t frozen.
This is called shallow freezing.
To fully freeze → you need deep freeze (freeze nested objects manually).

Q.Difference between Object.freeze() and Object.seal()
Object.seal()
Sealed object = structure locked, but values can still change.
You cannot: add, delete
But you can update existing properties

```
const user = {                          Output
  name: "Nikhil",                         { name: "Nikhil", age: 25 }
  age: 21
};

Object.seal(user);
user.age = 25;       // □ allowed
user.city = "Patna"; // □ not added
delete user.name;    // □ not deleted
console.log(user);
```

Check Methods

```
Object.isSealed(obj);
Object.isFrozen(obj);
```

Feature	Object.seal()	Object.freeze()
Add new properties	Not allowed	Not allowed
Delete properties	Not allowed	Not allowed
Modify existing properties	Allowed	Not allowed
Most restrictive	No	Yes

Q.Shallow copy vs deep copy?
Shallow Copy vs Deep Copy
When you copy an object or array, you might:
copy only the reference (problem)
copy the entire nested data (safe)

Shallow Copy
A shallow copy copies only the first level of an object.
Nested objects are still shared (same memory reference).

```
const user = {
  name: "Nikhil",                          Output
  address: { city: "Ranchi" }                 Delhi
};
const copy = { ...user };   // shallow copy
copy.name = "Rahul";
copy.address.city = "Delhi";
console.log(user.address.city);
```

Why?
Because:
address object was NOT copied
both objects point to same memory
So changing nested data changes the original.

Deep Copy
A deep copy duplicates the entire object including nested objects.
No shared references.

```
const user = {                         Output
  name: "Nikhil",                        Ranchi
  address: { city: "Ranchi" }
};
```

```
const copy = JSON.parse(JSON.stringify(user));
copy.address.city = "Delhi";
console.log(user.address.city);
```

Feature	Shallow copy	Deep copy
Copies first level	⍰	⍰
Copies nested object	⍰	⍰
Shares memory	⍰	⍰
Safe for state updates	⍰	⍰

Q.Object.keys vs Object.values vs Object.entries
All three are static methods of Object used to extract data from an object.

```
const user = {
  name: "Nikhil",
  age: 21,
  city: "Ranchi"
};
```

Object.keys()
Returns an array of all property names (keys).

```
Object.keys(user);
```

Output: ["name", "age", "city"]

Object.values()
Returns an array of all property values.

```
Object.values(user);
```

Output: ["Nikhil", 21, "Ranchi"]

Object.entries() ⍰
Returns an array of key–value pairs.

```
Object.entries(user);
```

```
Output: [
 ["name","Nikhil"],
 ["age",21],
```

 ["city","Ranchi"]
]

Q.What is property descriptor?
A property descriptor is an object that defines the behavior and permissions of an object’s property (not just its value).
In simple words:
Every property in JavaScript has hidden settings like:
can it be changed?
can it be deleted?
can it appear in loops?
These settings are called the property descriptor.

Viewing a Property Descriptor
We use:
Object.getOwnPropertyDescriptor(object, propertyName);
Example
```
const user = {
  name: "Nikhil"
};
console.log(Object.getOwnPropertyDescriptor(user, "name"));
```

Output:
```
{
  value: "Nikhil",
  writable: true,
  enumerable: true,
  configurable: true
}
```

The 4 Descriptor Properties ⍰
1.value
The actual stored data.
value: "Nikhil"

2.writable
Can the value be changed?
user.name = "Rahul";
If writable: false → it won’t change.

3. enumerable
Will it appear in loops?
```
for(let key in user){
  console.log(key);
```

}
If enumerable: false → it will be hidden.

4. configurable
Can the property be deleted or redefined?
delete user.name;
If configurable: false → cannot delete or modify descriptor.

Q.How to clone an object?
Cloning an object means creating a copy of it instead of copying its reference.
cause this is wrong:
const a = {name:"Nikhil"};
const b = a;   // NOT a clone
Both point to the same memory → changing b changes a.

1.Spread Operator (Most Common) ?
const user = { name:"Nikhil", age:21 };
const copy = { ...user };
copy.name = "Rahul";
console.log(user.name);

Output:
Nikhil ?

2.Object.assign()
const user = { name:"Nikhil", age:21 };
const copy = Object.assign({}, user);
Also a shallow copy.

3.Deep Clone using structuredClone() (Best Modern Way) ?
```
const user = {
  name:"Nikhil",
  address:{ city:"Ranchi" }
};
const copy = structuredClone(user);
copy.address.city = "Delhi";
console.log(user.address.city);
```

Output:
Ranchi ?

4.Deep Clone using JSON (Older trick)
const copy = JSON.parse(JSON.stringify(user));

Works, but has limitations:
removes functions
removes Date
removes undefined

Q.Difference between map and forEach
forEach()
Used when you only want to do something (side-effects) with each element.
It does NOT return a new array.

```
const numbers = [1,2,3,4];
numbers.forEach((num) => {
  console.log(num * 2);
});
```

Output in console:

```
2 4 6 8
```

Why?
Because forEach() returns nothing.
Use cases,Printing,Updating DOM,API calls,Modifying external variables

Example:

```
let sum = 0;
numbers.forEach(n => sum += n);
console.log(sum); // 10
```

This is called a side effect.

map()
Used when you want to transform data and create a new array.
It always returns a new array.

```
const numbers = [1,2,3,4];
const doubled = numbers.map(num => num * 2);
console.log(doubled);
```

Output:

```
[2,4,6,8]
```

Important:
Original array is unchanged
New array is created

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

Q.What id filter()?
filter() is an array method used to select elements based on a condition.

It returns a new array containing only those elements that pass (true) the condition.

```
array.filter((element, index, array) => {
  return condition;
});
```

If condition → true □ element kept
If condition → false □ element removed

```
const numbers = [1,2,3,4,5,6];
const even = numbers.filter(num => num % 2 === 0);
console.log(even);
```

Output:

```
[2,4,6]
```

Notice:
Original array unchanged
New array created

```
const users = [
 {name:"Nikhil", age:21},
 {name:"Rahul", age:16},
 {name:"Aman", age:25}
];

const adults = users.filter(user => user.age > 18);
console.log(adults);
```

Output:

```
[
 {name:"Nikhil", age:21},
 {name:"Aman", age:25}
]
```

------------- ---------------------------------

```
const result = [1,2,3,4,5,6]
            .filter(n => n % 2 === 0)
            .map(n => n * 10);

console.log(result);                    //[20,40,60]
```

Q.What is reduce?
reduce() is used to convert an entire array into a single value.
That single value can be:

number (sum, max)
object
string
array
anything
It “reduces” many values ⍰ into one final result.

```
array.reduce((accumulator, currentValue) => {
  return updatedAccumulator;
}, initialValue);
```

Terms (VERY IMPORTANT)
accumulator (acc) → stores the result after each iteration
currentValue (curr) → current element in the array
initialValue → starting value of accumulator

Examples:

```
const numbers = [1,2,3,4];
const sum = numbers.reduce((acc, curr) => {
   return acc + curr;
}, 0);

console.log(sum);              //10
-----------------------------------------------------------------
const nums = [5,12,8,20,1];
const max = nums.reduce((acc, curr) => {
   return curr > acc ? curr : acc;
}, nums[0]);

console.log(max);         //20
-----------------------------------------------------------------
```

Why reduce() is powerful
Because it can actually replace map + filter + loops.
Example:

```
const arr = [1,2,3,4,5];
const result = arr.reduce((acc, curr) => {
  if(curr % 2 === 0){
    acc.push(curr*2);
  }
  return acc;
}, []);
```

```
console.log(result);
```

Output:

```
[4,8]
```

Q.What is find()?
find() in JavaScript
find() is an array method used to get the FIRST element that matches a condition.
It returns:
the element itself ⍰
or undefined if not found

```
array.find((element) => condition);

const numbers = [5, 12, 8, 130, 44];
const result = numbers.find(num => num > 10);
console.log(result);                        //12

const users = [
 {id:1, name:"Nikhil"},
 {id:2, name:"Rahul"},
 {id:3, name:"Aman"}
];
const user = users.find(u => u.id === 2);

console.log(user);                      //{id:2, name:"Rahul"}
```

Backend use

```
const product = products.find(p => p.id === req.params.id);
```

Q.What is some and every?
some() and every() in JavaScript
Both are array methods used to test a condition on elements.
They always return a boolean (true / false).

some()
Checks: “Is at least ONE element satisfying the condition?”
If one match is found → true
If none → false

Example

```
const numbers = [1,3,5,8,9];
const result = numbers.some(n => n % 2 === 0);
console.log(result);
```
Output:
```
true
```

Why?
Because 8 is even → only one needed ⍰ , when noting matches its flase

every()
Checks: “Are ALL elements satisfying the condition?”
If one element fails → false immediately

Example
```
const numbers = [2,4,6,8];
const result = numbers.every(n => n % 2 === 0);
console.log(result);
```

Output:
```
true
```

All numbers are even ⍰

Failure case
```
[2,4,6,7].every(n => n % 2 === 0);
```
Output: false

Q.Difference between slice and splice
slice() — Copy part of an array
It extracts elements and returns a new array.
Syntax
```
array.slice(start, end);
```
start → included
end → excluded

Example
```
const arr = [10,20,30,40,50];
const result = arr.slice(1,4);
console.log(result);
console.log(arr);
```
Output:
```
[20,30,40]
[10,20,30,40,50]
```

Original array unchanged

So slice() = non-destructive.

splice() — Modify the array
It edits the original array.
Syntax
```
array.splice(start, deleteCount, newItems)
```
start → where to begin
deleteCount → how many to remove
newItems → (optional) items to insert

Remove elements
```
const arr = [10,20,30,40,50];
arr.splice(1,2);
console.log(arr);
```

Output:
```
[10,40,50]
```

```
const arr = [1,2,5];
arr.splice(2,0,3,4);
console.log(arr);
```
Output:
```
[1,2,3,4,5]
```

Q.How to remove duplicates from array?
Best & Modern Method — Set (MOST IMPORTANT)
JavaScript Set stores only unique values.
```
const arr = [1,2,2,3,4,4,5];
const unique = [...new Set(arr)];
console.log(unique);
```

Output:
```
[1,2,3,4,5]
```
Why it works
Set automatically removes duplicates
... (spread operator) converts it back to array

Q.How to flatten an array?
Flatten an array = convert a nested array into a single-level array.
Example:
[1, [2,3], [4,[5,6]]]
→ [1,2,3,4,5,6]

Easiest (Modern JS) — flat()

JavaScript already has a built-in method.

```
const arr = [1,[2,3],[4,[5,6]]];
console.log(arr.flat(2));
```

Output:

```
[1,2,3,4,5,6]
```

Flatten completely

```
arr.flat(Infinity);
```

We can flatten an array using the built-in flat() method (arr.flat(Infinity)), or by using recursion to iterate through nested arrays and collect elements into a single array.

Q.What is Array.from()?

Array.from() is used to convert array-like or iterable values into a real array.

Syntax

```
Array.from(source, mapFunction)
```

source → thing you want to convert
mapFunction (optional) → modify each element while creating array

Why we need it?

Some things in JavaScript look like arrays but are NOT real arrays.

Examples:

string,Set,Map,arguments,NodeList (DOM querySelectorAll)

They don’t have array methods like .map(), .filter().

So we convert them → real array.

Convert a String to Array

```
const str = "hello";
const arr = Array.from(str);
console.log(arr);
```

Output:

```
['h','e','l','l','o']
```

```
const set = new Set([1,2,2,3,4]);
const arr = Array.from(set);
console.log(arr);          //  [1,2,3,4]
```

Q.What is synchronous vs asynchronous?

Synchronous (Blocking)

Synchronous code executes line-by-line.
Next line waits until the previous line finishes.

One task at a time.

Example

```
console.log("Start");
console.log("User data loading...");
console.log("End");
```

Output:

```
Start
User data loading...
End
```

Asynchronous (Non-Blocking)
Asynchronous code does NOT wait.
It starts a task in the background and moves to the next line immediately.
JavaScript uses:
setTimeout,setInterval
fetch / API calls
Promises
async/await

Example

```
console.log("Start");
setTimeout(() => {
  console.log("Loading finished");
}, 2000);
console.log("End");
```

Output:

```
Start
End
Loading finished
```

Why?
Because setTimeout is asynchronous — JS schedules it and continues.

Q.What is the call stack?
The call stack is a data structure (stack — LIFO: Last In, First Out) that JavaScript uses to keep track of which function is currently executing.
In simple words:
The call stack is where JavaScript remembers “which function is running right now”.

How it works
Whenever a function is called:
JS pushes it onto the stack
When the function finishes:
JS pops it off the stack

```
function first(){
  console.log("First");
}

function second(){
  first();
  console.log("Second");
}
second();
```

```
output:First
Second
```

Q.What is the event loop?

The event loop is a mechanism that allows JavaScript to perform asynchronous (non-blocking) operations by coordinating the call stack and the callback queue.

In simple words:
Event Loop decides WHEN async code gets executed.

```
console.log("Start");
setTimeout(()=>{
  console.log("Timer done");
},2000);
console.log("End");
```

```
Output:
Start
End
Timer done
```

Step 1 — console.log("Start")
Goes to call stack → executes immediately.

Step 2 — setTimeout
JS sees async function → NOT executed in call stack
It is sent to Web APIs (browser timer).
Call stack does NOT wait.

Step 3 — console.log("End")
Runs immediately.
So far output:
Start

End

Step 4 — Timer finishes
After 2 seconds:
Callback goes into Callback Queue
(Not to call stack yet)

Step 5 — Event Loop checks
Event loop continuously checks:

"Is the call stack empty?"
If YES → it pushes callback from queue → to stack.
Now it runs:
Timer done

Q.What is callback Queue?
The callback queue (also called task queue / macrotask queue) is a place where asynchronous callback functions wait before getting executed.
It stores functions that are ready to run but are waiting for the call stack to become empty.
JavaScript does NOT execute async functions immediately.
They first go to background → then queue → then stack.

Who puts functions into the callback queue?
These async operations:
setTimeout,
setInterval,
DOM events (click, keypress),
some browser APIs

```
console.log("Start");
setTimeout(()=>{
  console.log("Hello");
},1000);
console.log("End");
```

Output:
```
Start
End
Hello
```

Step-by-Step Flow
1. console.log("Start")
→ goes to Call Stack → executes.

2. setTimeout
JS sends it to Web APIs (browser timer)
Call stack does not wait.
After 1 second → its callback:
()=> console.log("Hello")
goes into the Callback Queue.

3. console.log("End")
Executes immediately.
Call stack becomes empty.

4. Event Loop
Now the event loop checks:
“Is the call stack empty?”
YES ⍰

So it takes the callback from callback queue
and pushes it into the call stack.
Now it runs:
Hello

Q.What is the microtask queue?
The microtask queue is a special high-priority queue that stores Promise callbacks and some other async operations.
It executes before the callback queue when the call stack becomes empty.
So JavaScript has TWO queues:
Microtask Queue (HIGH priority)
Callback Queue / Macrotask Queue (LOW priority)

Who goes into Microtask Queue?
.then() of Promises
.catch()
.finally()
queueMicrotask()
MutationObserver

Example (VERY IMPORTANT)

```
                                    Output
console.log("Start");                 Start
setTimeout(()=>{                     End
  console.log("Timeout");                Promise
},0);                                    Timeout
Promise.resolve().then(()=>{
  console.log("Promise");
});
console.log("End");
```

```
Call Stack Empty
     ↓
Microtask Queue  (Promises)  ← runs FIRST
     ↓
Callback Queue   (setTimeout)
```

Q.What is callback hell?

Callback hell is a situation where multiple nested callbacks make asynchronous code difficult to read, debug, and maintain, forming a pyramid-like structure. It is solved using Promises and async/await.

Too many nested callbacks → callback hell

Q. What is a promise? States of promise

A Promise is an object that represents the result of an asynchronous operation that will complete in the future.
A Promise is a guarantee that JavaScript gives — "I will return a value later (success or failure)."
Example:
API request, Database query ,File reading ,Payment processing
Because these take time, JS does not block → it returns a Promise.

States of promise:

State	**Meaning**
Pending	Operation still running
Fulfilled	Operation successful
Rejected	Operation failed

Pending

Initial state.
The async task is still happening.
Example:
fetch("server.com/users");
Server hasn't responded yet.

Fulfilled (Resolved)

Operation completed successfully → resolve() called.

```
resolve("User data");
```
Then .then() runs:
```
promise.then(data => console.log(data));
```

Rejected

Operation failed → reject() called.
```
reject("Network error");
```
Then .catch() runs:
```
promise.catch(err => console.log(err));
```

```
fetch("https://api.com/users")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.log("Error:", err));
```

Q.What is then?
What is catch?
What is finally?

.then()

It runs when the Promise is successful (resolved / fulfilled).
It receives the value passed inside resolve().

Example
```
const p = new Promise((resolve, reject)=>{
  resolve("Data loaded");
});

p.then(result => {
  console.log(result);
});
```

Output:
```
Data loaded
```

So:
.then() handles the success result of a Promise.

Chaining (Very Important)
```
fetch("https://api.com/users")
  .then(res => res.json())
  .then(data => console.log(data));
```

Each .then() gets the result of the previous one.

.catch()
It runs when the Promise fails (rejected).
It receives the value passed inside reject().

Example
```
const p = new Promise((resolve, reject)=>{
  reject("Network Error");
});

p.catch(err => {
  console.log(err);
});
```

Output:
```
Network Error
```

So:
.catch() handles errors in a Promise.
Also catches errors thrown inside .then().

.finally()
It runs always — whether success or failure.
Used for cleanup tasks:
stop loader
close DB
hide spinner

Example
```
const p = new Promise((resolve, reject)=>{
  resolve("Success");
});

p
 .then(res => console.log(res))
 .catch(err => console.log(err))
 .finally(()=> console.log("Done"));
```

Output:
```
Success
Done
```

Even if rejected:
Error

Done

So: .finally() executes no matter what.

Interview One-Line Answer

then() → executes when promise is fulfilled
catch() → executes when promise is rejected
finally() → executes regardless of result

Q. What is promise chaining?
Promise chaining means attaching multiple .then() methods one after another so that the output of one async operation becomes the input of the next.
Instead of nested callbacks (callback hell), we write a sequence of steps.

```
new Promise((resolve, reject)=>{
  resolve(2);
})
.then(num => {
  console.log(num);   // 2
  return num * 2;
})
.then(num => {
  console.log(num);   // 4
  return num * 2;
})
.then(num => {
  console.log(num);   // 8
});
```

Output
```
2   4   8
```

Q.What is promise.all() ?
Promise.all() is used when you want to run multiple asynchronous operations at the same time and wait until all of them finish.
It takes an array of promises and returns a single Promise.

Syntax: Promise.all([promise1, promise2, promise3])

```
const p1 = Promise.resolve("User data");
const p2 = Promise.resolve("Orders");
const p3 = Promise.resolve("Payment");
```

```
Promise.all([p1,p2,p3])
  .then(results => console.log(results));
```

Output:

```
["User data","Orders","Payment"]
```

Q.What is promise.race() ?

Promise.race() runs multiple promises at the same time and returns the result of the first promise that finishes.
It does NOT wait for all promises.
Whichever promise settles first (resolve OR reject) wins the race.

Syntax: Promise.race([promise1, promise2, promise3])

```
const p1 = new Promise(res => setTimeout(()=>res("Fast"),1000));
const p2 = new Promise(res => setTimeout(()=>res("Slow"),3000));

Promise.race([p1,p2])
  .then(result => console.log(result));
```

Output:

```
Fast    //because p1 finished first
```

race() cares only about who finishes first, not success only.
Even if the first promise fails, race ends.

```
const p1 = new Promise((res,rej)=> setTimeout(()=>rej("Error"),1000));
const p2 = new Promise(res => setTimeout(()=>res("Success"),3000));
Promise.race([p1,p2])
 .then(res=>console.log(res))
 .catch(err=>console.log(err));
```

Output:

```
Error
```

Promise.all()	Promise.race()
waits for all promises	waits for first promise
fails if one fails	first result decides
returns array	returns single value

Q. What is Promise.any()?
Promise.any() runs multiple promises in parallel and returns the first SUCCESSFUL (resolved) promise.
It ignores rejected promises.
It resolves as soon as any one promise resolves.

Syntax
```
Promise.any([promise1, promise2, promise3])
```

```
const p1 = Promise.reject("Server 1 failed");
const p2 = Promise.resolve("Server 2 success");
const p3 = Promise.resolve("Server 3 success");

Promise.any([p1,p2,p3])
  .then(result => console.log(result))
  .catch(err => console.log(err));
```

Output: Server 2 success

Q.What is async/await?
async/await is syntax built on top of Promises that lets you write asynchronous code in a synchronous-looking way (top-to-bottom).
It makes Promise code easier to read and avoids long .then() chains.

async keyword
When you put async before a function:
```
async function test(){
  return "Hello";
}
```

JavaScript automatically returns a Promise.

```
test().then(res => console.log(res));
```

Output:
Hello
Even though you returned a string, JS wrapped it inside a Promise.

await keyword
await pauses the function execution until the Promise resolves.
It can only be used inside an async function.

Example

```
function getData(){
  return new Promise(resolve=>{
    setTimeout(()=>resolve("Data received"),2000);
  });
}

async function showData(){
  const result = await getData();
  console.log(result);
}
showData();
```

Output (after 2s):

```
Data received
```

Q.Error handling in async/await

In async/await, errors are handled using try...catch because a rejected promise behaves like a thrown exception inside an async function.

```
async function load(){
  try{
    const user = await getUser().catch(()=>null);
    const posts = await getPosts().catch(()=>[]);
    console.log(user, posts);
  }catch(err){
    console.log("Unexpected error");
  }
}
```

Promises (`then/catch`)	**async/await**
Uses chaining	Uses sequential style
Harder to read for many steps	Looks like normal synchronous code
Callback-like	Straight top-to-bottom flow
Error via `.catch()`	Error via `try...catch`
Better for parallel tasks	Better for step-by-step logic
Can create callback hell chains	Avoids nesting

Q. setTimeout vs setInterval
Why setTimeout(0) still delayed?

setTimeout vs setInterval
Both are asynchronous timer functions (Web APIs).
They don’t run in the call stack immediately — they are handled by the browser/Node timer system.

setTimeout()
Runs a function only once after a delay.

Syntax
```
setTimeout(callback, delay);
```

Example
```
setTimeout(()=>{
  console.log("Hello");
},2000);
```

After 2 seconds → prints:
```
Hello
```

Used for:
API retry
delay animation
debounce

setInterval()
Runs a function again and again repeatedly after every delay.

Syntax
```
setInterval(callback, delay);
```
Example
```
setInterval(()=>{
  console.log("Running...");
},2000);
```

Output every 2 seconds:
```
Running...
Running...
Running...
```

Used for:

clock,live data refresh ,polling server

You may think:
setTimeout(()=>console.log("Hi"),0);
console.log("Hello");

Output should be:
Hi
Hello

But actual output:
Hello
Hi
Reason → Event Loop

0 ms does NOT mean immediate execution.
It means:
“Execute this callback after the current call stack becomes empty.”

What actually happens
JS executes synchronous code first
setTimeout goes to Web APIs
Callback goes to Callback Queue
Event loop waits
When call stack empty → then runs

JavaScript handles concurrency using the event loop, where asynchronous tasks are delegated to Web APIs and their callbacks are queued and executed when the call stack is empty, enabling non-blocking behavior despite being single-threaded.

Microtasks (like Promise callbacks) have higher priority and execute immediately after the call stack is empty, whereas macrotasks (like setTimeout) execute afterward in the next event loop cycle.

Promise = VIP line
setTimeout = normal line

Q.What is DOM?
DOM = Document Object Model
The DOM is a programming interface that represents an HTML webpage as a tree of objects so JavaScript can read and modify the page.

In simple words:
DOM connects JavaScript ↔ HTML

Without DOM, JavaScript cannot change anything on a webpage.

```
<html>
  <body>
    <h1 id="title">Hello</h1>
  </body>
</html>
```

Browser converts this HTML into a tree structure in memory.

```
Document
  └── html
       └── body
            └── h1
```

Why DOM is Needed
Because JavaScript does NOT understand HTML directly.
JS only understands objects.
So browser converts HTML → DOM objects.
Now JS can manipulate:
text, styles ,image ,buttons ,forms

Accessing DOM

```
const heading = document.getElementById("title");
console.log(heading);
```

JS now gets the <h1> element.

Modifying DOM
Change text

```
heading.textContent = "Welcome Nikhil";
```

Q. How to select elements in DOM?
Selecting elements in the DOM means: how JavaScript finds HTML elements so you can read or change them.
JavaScript uses the document object to access the page.
Ways to Select Elements in DOM

getElementById() (Most common)
Selects element using id.
HTML

```
<h1 id="title">Hello</h1>
```

JS

```
const el = document.getElementById("title");
console.log(el);
```

Returns: single element

getElementsByClassName()
Selects elements using class name.

```
<p class="text">One</p>
<p class="text">Two</p>
const items = document.getElementsByClassName("text");
console.log(items);
```

Returns: HTMLCollection (array-like, not real array)
Access:

```
items[0]
```

getElementsByTagName()
Selects elements by tag.

```
const paragraphs = document.getElementsByTagName("p");
```

Returns multiple elements.

querySelector() ⍰ (VERY IMPORTANT)
Uses CSS selector.
Returns first matching element.

```
document.querySelector("#title");   // id
document.querySelector(".text");    // class
document.querySelector("p");        // tag
```

querySelectorAll() ⍰
Returns all matching elements.

```
const all = document.querySelectorAll(".text");
console.log(all);
```

Returns: NodeList

You can loop:

```
all.forEach(el => console.log(el.textContent));
```

Q. Event bubbling vs capturing
When you click an element inside another element, the event doesn’t stay there.
It travels through the DOM tree.
That travel is called event propagation, and it has two phases:
Capturing phase
Bubbling phase
Event Bubbling vs Event Capturing
Example HTML

```
<div id="parent">
  <button id="child">Click Me</button>
```

</div>

You click the button.
Now question:
Should the event run on button first or div first?
This is exactly bubbling vs capturing.

Event Bubbling (Default) ↑
Event starts from the target element and goes upward to ancestors.
child → parent → document
So:
button runs
div runs

Code
```
document.getElementById("child")
  .addEventListener("click", () => console.log("Button clicked"));

document.getElementById("parent")
  .addEventListener("click", () => console.log("Div clicked"));
```
Output
```
Button clicked
Div clicked
```

Because event bubbles up.

Event Capturing ↓
Event starts from the top (document) and goes downward to the target.
document → parent → child
To enable capturing, pass true as third parameter.

```
document.getElementById("parent")
  .addEventListener("click", () => console.log("Div clicked"), true);

document.getElementById("child")
  .addEventListener("click", () => console.log("Button clicked"), true);
```
Output
```
Div clicked
Button clicked
```

Now parent executes first.

stopPropagation() (VERY IMPORTANT ⍰)
Stops the event from moving further.

```
document.getElementById("child")
  .addEventListener("click", (e)=>{
    e.stopPropagation();
    console.log("Button only");
});
```

Now parent won’t run.

> Lazy loading is a performance optimization technique where resources are loaded only when they are needed. It is not specific to JavaScript, but JavaScript is commonly used to implement it, especially in modern frameworks like React.

What is localStorage?
What is sessionStorage?
Cookies vs localStorage

localStorage
localStorage stores data permanently in the browser.
Data remains even after refreshing, closing tab, or restarting browser.

Store data
```
localStorage.setItem("username", "Nikhil");
```

Get data
```
const user = localStorage.getItem("username");
console.log(user);
```

Remove
```
localStorage.removeItem("username");
```

Clear all
```
localStorage.clear();
```

Important
Stores only strings

```
localStorage.setItem("user", JSON.stringify({name:"Nikhil"}));
```

sessionStorage

sessionStorage stores data only for one tab session.
Data is deleted when the tab/browser is closed.

Refresh → keeps data
Close tab → data gone
Example

```
sessionStorage.setItem("token","12345");
```

Works same as localStorage, just lifetime is different.

Cookies
Cookies are small pieces of data stored in browser and sent to server with every HTTP request.

Used for:
authentication
session id
tracking

Create cookie

```
document.cookie = "user=Nikhil; expires=Fri, 31 Dec 2026 12:00:00 UTC";
```