1. What is JavaScript?
JavaScript is a **high-level, interpreted programming language** used mainly for web development. It is **single-threaded, dynamic, and prototype-based**.

---

2. Difference between var, let, and const?

- var → Function-scoped, hoisted, allows redeclaration.
- let → Block-scoped, hoisted but in temporal dead zone, no redeclaration.
- const → Same as let but cannot be reassigned.

```
var a = 10;

let b = 20;

const c = 30;
```

---

3. What is Hoisting?
Hoisting means moving variable & function declarations **to the top** of their scope before execution.

```
console.log(a); // undefined

var a = 10;
```

---

4. Difference between == and ===?

- == → Checks only value, does type coercion.
- === → Checks value + type, strict equality.

```
'5' == 5   // true

'5' === 5  // false
```

---

5. What are Template Literals?
Template literals allow embedding variables & expressions inside strings using backticks.

```
let name = "Darshan";

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

---

6. Explain null vs undefined.

- null → Intentional empty value.
- undefined → Declared but not assigned.

```
let a; // undefined
```

let b = null; // null

---

7. What are Arrow Functions?
Shorter syntax & do not have their own this.

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

---

8. What is the difference between function declaration and function expression?

- **Declaration** → Hoisted.
- **Expression** → Not hoisted.

// Declaration

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

// Expression

const sub = function(a, b) { return a - b; }

---

9. What is Scope in JavaScript?

- **Global Scope** → Available everywhere.
- **Function Scope** → Inside a function.
- **Block Scope** → Inside {} using let & const.

---

10. What is Closures in JavaScript?
Closure = Function + its lexical scope.
Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  }
}
let inc = outer();
console.log(inc()); // 1
```

```
console.log(inc()); // 2
```

---

11. What is the difference between synchronous and asynchronous JS?

- **Synchronous** → Executes line by line.
- **Asynchronous** → Non-blocking, executes later (via event loop).

---

12. What are Higher Order Functions?
Functions that take other functions as arguments or return a function.
Example: map, filter, reduce.

---

13. Explain call, apply, bind.

- call → Calls function with arguments separated.
- apply → Calls function with array of arguments.
- bind → Returns a new function with this fixed.

```
function greet(msg) { console.log(`${msg}, ${this.name}`); }

const user = { name: "Darshan" };

greet.call(user, "Hello");

greet.apply(user, ["Hi"]);

let newFn = greet.bind(user, "Hey");

newFn();
```

---

14. What are JavaScript Data Types?

- Primitive → string, number, boolean, null, undefined, bigint, symbol.
- Non-primitive → object, array, function.

---

15. Explain Event Bubbling and Capturing.

- **Bubbling** → Event goes from child → parent.
- **Capturing** → Event goes parent → child.

```
element.addEventListener("click", fn, true); // capturing

element.addEventListener("click", fn, false); // bubbling
```

🔷 **Intermediate (16–35)**

16. What is the Event Loop?
It allows JS to handle async code. Executes stack first, then queue (callbacks, promises).

---

17. What are Promises?
Object that represents eventual completion or failure.
States: **pending → fulfilled → rejected**.

```
new Promise((res, rej) => res("done"));
```

---

18. Difference between Callback and Promise?

- Callback → Pass function into another function.
- Promise → Cleaner async handling, chaining possible.

---

19. What is async/await?
Syntactic sugar over promises. Makes async code look synchronous.

```
async function getData() {
  let res = await fetch("url");
  return await res.json();
}
```

---

20. What are Generators in JS?
Functions that can pause & resume using yield.

```
function* gen() {
  yield 1;
  yield 2;
}
const g = gen();
console.log(g.next().value);
```

---

21. What is Debouncing & Throttling?

- **Debounce** → Delay execution until user stops.
- **Throttle** → Limit execution to once per time interval.

---

22. Explain Event Delegation.
Attaching one event listener to parent instead of multiple child nodes.

---

23. Difference between Shallow Copy & Deep Copy?

- **Shallow** → Only top-level copied. Nested objects still reference.
- **Deep** → Full independent copy.

```
let obj = {a:1, b:{c:2}};

let shallow = {...obj}; // shallow

let deep = JSON.parse(JSON.stringify(obj)); // deep
```

---

24. What is Prototypal Inheritance?
Objects inherit properties from prototype chain.

---

25. Difference between for...in and for...of?

- for...in → Iterates over object keys.
- for...of → Iterates over iterable values.

---

26. What is a Module in JS?
A file with export/import.

```
// math.js

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

// app.js

import { add } from './math.js';
```

---

27. Difference between ES5 & ES6 features?
ES6 introduced let, const, arrow functions, promises, classes, modules.

---

28. What is Currying?
Transform function of multiple args into series of single-arg functions.

```
const curry = (a)=>(b)=>(c)=>a+b+c;

console.log(curry(1)(2)(3));
```

---

29. Explain Memory Management in JS.
Automatic garbage collection using reference counting & reachability.

---

30. What are WeakMap and WeakSet?
Collections where keys are weakly referenced → helps in garbage collection.

---

31. Explain IIFE (Immediately Invoked Function Expression).
Executes immediately after creation.

```
(function() {
  console.log("IIFE runs immediately!");
})();
```

---

32. What are Service Workers?
Scripts that run in background, used for caching & offline apps (PWA).

---

33. Difference between map and forEach?

- map → Returns new array.
- forEach → Iterates but returns undefined.

---

34. Explain this keyword.

- In global → window object.
- Inside function → depends on how function is called.
- Arrow function → takes lexical this.

---

35. What is Destructuring in JS?

```
const {name, age} = {name:"Darshan", age:22};
```

🔷 **Advanced (36–50)**

36. What is the difference between localStorage, sessionStorage, and cookies?

- localStorage → Permanent (5–10MB).
- sessionStorage → Cleared on tab close.
- cookies → Small data, sent with every request.

---

37. What is CORS in JavaScript?
Cross-Origin Resource Sharing → Controls access to resources from different domains.

---

38. What is Polyfill in JS?
Code that provides modern feature in old browsers. Example: Array.prototype.includes.

---

39. What is the difference between deepFreeze and Object.freeze?

- Object.freeze → Shallow freeze.
- deepFreeze → Recursively freezes all nested objects.

---

40. Explain Microtask Queue vs Macrotask Queue.

- Microtask → Promises, MutationObservers.
- Macrotask → setTimeout, setInterval.

---

41. What is the difference between WeakMap and Map?

- WeakMap keys must be objects.
- WeakMap allows garbage collection.

---

42. What are JavaScript Engines?
Programs that run JS code → V8 (Chrome), SpiderMonkey (Firefox).

---

43. Difference between Functional & Object-Oriented Programming in JS?

- FP → Pure functions, immutability.
- OOP → Classes, objects, inheritance.

---

44. What is Shadow DOM?
Encapsulation method for web components. Prevents style leakage.

---

45. Explain typeof null returning object.
Legacy bug in JS. null is not an object but typeof shows "object".

---

46. What is Tree Shaking in JS?
Removing unused code during bundling (e.g., webpack).

---

47. What is the difference between async/await and .then()?

- .then() → Promise chaining.
- async/await → Cleaner, synchronous-like syntax.

---

48. Explain JavaScript Event Loop with example.

```
console.log("1");

setTimeout(()=>console.log("2"),0);

Promise.resolve().then(()=>console.log("3"));

console.log("4");

// Output: 1,4,3,2
```

---

49. What is a Pure Function?
Function that has no side effects and always returns same output for same input.

---

50. Difference between Immutable and Mutable objects in JS?

- Mutable → Can be changed (objects, arrays).
- Immutable → Cannot be changed once created (primitives, frozen objects).

51. What is Event Loop starvation?

If the microtask queue (Promises, process.nextTick) grows too large, it can block macrotasks (setTimeout, setInterval) → starving them from execution.

---

52. Difference between undefined, NaN, and isNaN()?

- undefined → Variable declared but not assigned.
- NaN → Result of invalid numeric operation ("abc" / 2).
- isNaN(value) → Checks if value is NaN, but Number.isNaN() is more accurate.

```
isNaN("abc") // true

Number.isNaN("abc") // false
```

---

53. What are Tagged Template Literals?

Special kind of template literals that let you parse template strings with a function.

```
function tag(strings, ...values) {
  return strings[0] + values[0].toUpperCase();
}
console.log(tag`hello ${"darshan"}`); // "hello DARSHAN"
```

---

54. What is Optional Chaining (?.) and Nullish Coalescing (??)?

- **Optional chaining** prevents errors when accessing nested properties.
- **Nullish coalescing** returns a fallback if value is null or undefined.

```
let user = { profile: null };
console.log(user?.profile?.name); // undefined
console.log(user?.profile?.name ?? "Guest"); // Guest
```

---

55. Explain Temporal Dead Zone (TDZ).

Variables declared with let & const are hoisted but not initialized until execution reaches them. Accessing before initialization → ReferenceError.

```
console.log(a); // ReferenceError
let a = 5;
```

---

56. Difference between Object.create() and class in JS?

- Object.create(proto) → Creates object directly with given prototype.
- class → Syntactic sugar for constructor + prototype methods.

---

57. What is Dynamic Import in JS?

Loads modules **lazily** at runtime → improves performance.

```
import("./math.js").then(module => {
  console.log(module.add(2,3));
});
```

---

58. What are Observables (RxJS) vs Promises?

- **Promise** → Handles one async value.

- **Observable** → Handles multiple values over time (streams, like events, WebSockets).

---

59. Explain Garbage Collection in JS (Mark-and-Sweep Algorithm).

- JS uses **reachability** → If an object is not reachable from the root (global scope), it gets garbage collected.

---

60. What are Symbols in JS? Why use them?

- Unique, immutable identifiers, used as object keys to avoid naming collisions.

```
const id = Symbol("id");
let user = { [id]: 123 };
```

---

61. Difference between Object.seal(), Object.freeze(), and Object.preventExtensions()?

- preventExtensions → Cannot add new properties.
- seal → Cannot add/remove, but can modify existing values.
- freeze → Fully immutable.

---

62. What are Proxy objects in JS?

Used to intercept and redefine operations on objects.

```
let obj = {name: "Darshan"};
let proxy = new Proxy(obj, {
  get(target, prop) {
    return prop in target ? target[prop] : "Not Found";
  }
});
console.log(proxy.age); // Not Found
```

---

63. What is the difference between async function and function* generator?

- **async** → Always returns a Promise.
- **generator (function*)** → Returns an iterator, can pause execution with yield.

---

64. What is Top-Level Await?

In ES2022+, you can use await outside async in modules.

```
const data = await fetch("url").then(r=>r.json());
```

---

65. What is Memoization in JavaScript?

Caching function results to improve performance.

```
function memoize(fn) {
  let cache = {};
  return function(x) {
    if (x in cache) return cache[x];
    return (cache[x] = fn(x));
  };
}
```

1. Primitives vs. Objects

Your list touches on data types, but it's important to understand the core distinction between primitive values and objects.

- **Primitives** are simple, immutable data types. When you assign a primitive to a new variable, you are creating a new copy of the value. Changing the new variable's value does not affect the original. The primitive data types are string, number, boolean, null, undefined, bigint, and symbol.
- **Objects** are complex, mutable data types. When you assign an object to a new variable, you're copying a **reference** to the object in memory, not the object itself. Changes made through the new variable will affect the original object because they both point to the same location in memory. Examples include object, array, and function.

---

2. The typeof operator

While your list mentions the typeof null issue, it doesn't explain what the typeof operator is used for in the first place.

The **typeof operator** is a simple way to determine the data type of a variable or expression. It returns a string indicating the type.

JavaScript

```
console.log(typeof "hello"); // "string"
console.log(typeof 10);     // "number"
console.log(typeof true);   // "boolean"
console.log(typeof {});     // "object"
```

```
console.log(typeof []);     // "object"

console.log(typeof null);   // "object" (This is a well-known bug)

console.log(typeof function() {}); // "function"

console.log(typeof undefined); // "undefined"
```

---

3. Type Coercion

Your list defines the difference between == and ===, which is a great start, but it doesn't explicitly define **type coercion** itself.

Type coercion is JavaScript's automatic conversion of a value from one data type to another. It happens implicitly with operators like == and explicitly with functions like Number() or String(). Understanding this is crucial for debugging why certain comparisons or operations behave unexpectedly.

JavaScript

```
// Implicit Coercion with ==

console.log(5 == '5'); // true (The string '5' is coerced to the number 5)


// Explicit Coercion

let str = "100";

let num = Number(str); // Explicitly converts "100" to the number 100

console.log(typeof num); // "number"
```