Javascript Interview Questions Deep Dive(16-30 QA)

16. What is generator function in javascript ?

A **generator function** in JavaScript is a special type of function that can **pause and resume** its execution.

It allows you to **generate values on the fly** using the `yield` keyword, making it perfect for **lazy evaluation**, **custom iterators**, and **asynchronous control flow**.

✅ **Syntax of a Generator Function**

javascript

```
function* generatorFunction() {
  yield 1;
  yield 2;
  yield 3;
}
```

- `function*` (with a star) defines a generator.
- `yield` is used to **pause and return** a value.

🧪 **Example: Simple Generator**

javascript

```
function* numbers() {
  yield 10;
  yield 20;
  yield 30;
}

const gen = numbers();

console.log(gen.next()); // { value: 10, done: false }
console.log(gen.next()); // { value: 20, done: false }
console.log(gen.next()); // { value: 30, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Every time `.next()` is called, it resumes from the last `yield`.

🔁 Loop Through a Generator

javascript

```
for (const num of numbers()) {
  console.log(num);
}
// Output: 10, 20, 30
```

⚡ Why Use Generators?

✅ 1. Custom Iteration Logic

javascript

```
function* countdown(start) {
  while (start > 0) {
    yield start--;
  }
}

for (const value of countdown(3)) {
  console.log(value); // 3, 2, 1
}
```

✅ 2. Lazy Evaluation (on-demand values)

You don't create a full array in memory — you **generate values when needed**.

---

17. What is prototype in javascript ?

🧠 Simple Definition:

> Every JavaScript object has a hidden internal property called `[[Prototype]]` (accessible via `.prototype` or `__proto__`) that points to another object.
> This allows one object to **inherit properties and methods** from another.

✅ Example: Using Prototypes to Share Methods

javascript

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

```
}

// Add a method to the prototype
Person.prototype.greet = function () {
  console.log(`Hello, I'm ${this.name}`);
};

const p1 = new Person("Alice");
p1.greet(); //  🔸 Hello, I'm Alice


// p1 doesn't have greet on itself — it inherits it from Person.prototype.
```

🔄 Prototype Chain

When you access a property like `p1.greet()`, JavaScript:

1. Looks in `p1` directly.
2. If not found, looks in `p1.__proto__` (i.e., `Person.prototype`).
3. Keeps going up the **prototype chain** until it finds it or hits `null`.

🔍 Real Example of Prototype Chain

javascript

```
const obj = {};
console.log(obj.toString); // comes from Object.prototype


// Even an empty object inherits toString() from Object.prototype.
```

🧱 Built-in Prototypes

- `Array.prototype` – All arrays share methods like `push()`, `map()`
- `Function.prototype` – All functions share methods like `call()`, `bind()`
- `Object.prototype` – The root of all objects

📐 Custom Inheritance with Prototype

javascript

```
const animal = {
  speak() {
    console.log("Animal speaks");
  },
};

const dog = Object.create(animal);
dog.bark = function () {
  console.log("Woof!");
};

dog.speak(); //  🔸 Animal speaks (inherited)
dog.bark();  //  🔸 Woof!
```

---

18. What is IIFE ?

An **IIFE** (Immediately Invoked Function Expression) is a **function that runs as soon as it is defined**.

🧠 Full Form:

IIFE stands for **Immediately Invoked Function Expression**

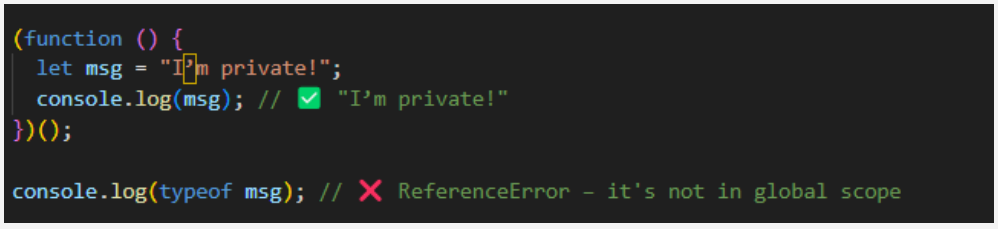
✅ Syntax

javascript

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

- The function is wrapped in parentheses `()` to turn it into an **expression**
- The second pair of parentheses `()` **immediately invokes** it

✅ Example with Variables



✅ IIFEs are often used to **create private scope** and **avoid polluting the global namespace**.

🧪 IIFE with Parameters

```
(function(name) {
  console.log(`Hello, ${name}`);
})("Alice"); // ✅ Hello, Alice
```

🗒️ Arrow Function IIFE

```
(() => {
  console.log("Arrow IIFE!");
})();
```

✅ Use Cases of IIFE

Use Case	Why It's Useful
Avoid global variables	Scope isolation
Create private variables	Can't be accessed from outside
Modular code before ES6 modules	Emulated module-like behavior
Execute code once (init logic)	Run config/init code on load

---

19. What is CORS ?

CORS stands for **Cross-Origin Resource Sharing**.

🌍 What is CORS?

> **CORS is a security feature** built into browsers that **controls how web pages can make requests to a different domain (origin)** than the one that served the page.

✅ Same-Origin Example:

```
Frontend: https://example.com
Backend:  https://example.com/api/data ✅ ALLOWED
```

❌ Cross-Origin Example (CORS kicks in):

```
Frontend: https://myapp.com
Backend:  https://api.example.com ❌ BLOCKED unless allowed by server
```

🛡️ Why Does CORS Exist?

To **prevent malicious websites** from reading sensitive data from another site **without permission** (like stealing your banking session).

🔧 How CORS Works

When your frontEnd makes a request to another origin:

1. The browser sends a **preflight request** (for methods like `POST`, `PUT`, etc.).
2. The server must respond with specific headers like:

http

```
Access-Control-Allow-Origin: https://myapp.com
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Content-Type
```

If these headers are missing or incorrect, the browser **blocks the response**.

🧪 Example Using `fetch` in JavaScript

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

If the API doesn't allow your origin, you'll get a CORS error like:

```
Access to fetch at '...' from origin '...' has been blocked by CORS
policy
```

⚙️ Fixing CORS

- **On server side (best practice):**
  - Add `Access-Control-Allow-Origin` header for allowed domains
  - Can be configured in:
    - **Express (Node.js)**: Use `cors` middleware
    - **Spring Boot, Django, Flask**: Configure CORS in settings
- **On client side (dev only):**
  - Use a **proxy** (e.g. Webpack dev server)
  - Bypass with browser extensions (not recommended for production)

🔁 Summary

Term	Description
CORS	A browser policy that **restricts cross-origin requests** unless the server explicitly allows them
Error Message	"Blocked by CORS policy"
Fix	Add CORS headers on the server

---

20. What are the different datatypes in javascript ?

🧱 1. Primitive Data Types (Immutable)

These are the most basic data types. They are **not objects** and are stored **by value**.

Type	Example	Description
`String`	`"hello"`	Sequence of characters
`Number`	`42`, `3.14`, `NaN`	Numeric values (integer or float)
`BigInt`	`1234567890123456789012345678901234567890n`	Large integers beyond Number limit

`Boolean`	`true`, `false`	Logical values
`undefined`	`let x;` → `x === undefined`	Declared but not assigned a value
`null`	`let x = null`	Intentional absence of any value
`Symbol`	`Symbol("id")`	Unique and immutable identifier

📦 2. Non-Primitive (Reference) Data Types

These are more complex types. They are **stored by reference**.

Type	Example	Description
`Object`	`{ name: "Alice", age: 25 }`	Key-value pairs
`Array`	`[1, 2, 3]`	Ordered list (special kind of object)
`Function`	`function() {}` or `() => {}`	Callable object
`Date`	`new Date()`	Date/time representation
`RegExp`	`/abc/`	Regular expressions
`Map`	`new Map()`	Key-value store (keys can be objects)
`Set`	`new Set()`	Collection of unique values

🔎 `typeof` Operator Behavior

```
typeof "hello";      // "string"
typeof 42;           // "number"
typeof true;         // "boolean"
typeof undefined;    // "undefined"
typeof null;         // ❗ "object" (this is a known bug)
typeof {};           // "object"
typeof [];           // "object" (arrays are also objects)
typeof function(){}; // "function"
```

🎯 Summary

✅ **Primitive:**

- `string`, `number`, `bigint`, `boolean`, `undefined`, `null`, `symbol`

✅ **Non-Primitive:**

- `object`, `array`, `function`, `map`, `set`, `date`, `regexp`, etc.

---

21. What are the difference between typescript and javascript ?

🧠 **1. Definition**

Aspect	JavaScript	TypeScript
Type	**Scripting language**	**Superset of JavaScript**
Created By	Netscape (Brendan Eich)	Microsoft
Extension	`.js`	`.ts`
Execution	Runs directly in browser or Node.js	Must be compiled to JavaScript

---

✅ **2. Type System**

Feature	JavaScript	TypeScript
Static Typing	❌ No	✅ Yes
Type Inference	❌ No	✅ Yes (e.g., `let name = "Alice";`)
Compile-time Error Check	❌ No	✅ Yes (helps catch bugs early)

Example:

ts

```
// TypeScript
let age: number = 25;
age = "twenty-five"; // ❌ Error: Type 'string' is not assignable to type 'number'
```

js

```
// JavaScript
let age = 25;
age = "twenty-five"; // ✅ No error, but might cause bugs at runtime
```

---

22. What is authentication vs authorization ?

🔐 1. Authentication (WHO are you?)

Authentication is the process of **verifying a user's identity**.

✅ **Example:**

- Login with username & password
- Fingerprint or Face ID
- Google/Facebook login (OAuth)

✅ **Real-world analogy:**

Showing your **ID card** at the gate

🛡️ 2. Authorization (WHAT can you do?)

Authorization is the process of **granting or denying access** to resources **based on permissions**.

✅ **Example:**

- Can this user **view admin dashboard**?
- Can this user **delete a post**?
- Role-based access: admin vs user vs guest

✅ **Real-world analogy:**

Being **allowed into VIP area** after your ID is checked

🧪 Example in Code:

js

```
// Authentication
if (user.email === 'admin@example.com' && user.password === '1234') {
  // Authenticated ✅
}
```

```
// Authorization
if (user.role === 'admin') {
  // Allow access to admin panel ✅
} else {
  // Deny access ❌
}
```

🔁 Summary Table

Feature	Authentication	Authorization
Purpose	Verifies **who you are**	Verifies **what you can access**
Occurs First?	✅ Yes	❗ Only after authentication
Example	Login, Biometrics	Roles, Permissions, Access Control
Returns	Identity (user ID/token)	Access level (can/can't do stuff)
Controlled by	**User credentials**	**Access rules / policies**

✅ Quick Tip

> You **must be authenticated before you can be authorized**.

---

23. Difference between null and undefined ?

🔍 Quick Comparison

Feature	`undefined`	`null`
Meaning	A variable has been declared but not assigned a value	A variable has been explicitly set to "no value"
Type	`undefined`	`object` (❗ this is a known quirk)
Set by	JavaScript (implicitly)	Developer (explicitly)
Common Usage	Default value for uninitialized variables or missing function params	Represents intentional absence or reset
Equality (==)	`null == undefined` → `true`	
Strict Equality (===)	`undefined === null` → `false`	

✅ **Examples:**

🔹 **`undefined` – Implicit absence**

javascript

```
let x;
console.log(x); // ❗ undefined (not assigned)

function greet(name) {
  console.log(name);
}
greet(); // ❗ undefined (no argument passed)
```

🔹 **`null` – Explicit absence**

javascript

```
let user = null;
console.log(user); // ❗ null (set intentionally)

if (user === null) {
  console.log("User is intentionally empty");
}
```

🧠 **`typeof` behavior**

javascript

```
typeof undefined; // 🔸 "undefined"
typeof null;      // 🔸 "object" (this is a JavaScript bug from early days)
```

🗒️ **Summary**

- ✅ Use **`undefined`** for **missing/uninitialized** values (default).
- ✅ Use **`null`** for **intentional clearing** or **"empty object"** semantics.

---

24. What is the output of 3+2+”7” ? 57

In JavaScript:

```
3 + 2 + "7"
```

Step-by-step breakdown:

1. **Left to right evaluation (JavaScript uses left-to-right associativity for `+`)**:
    - First, `3 + 2` is evaluated:
        - Both are numbers → `3 + 2 = 5`
2. Next, we have:

    `5 + "7"`
3. **Type coercion happens**:
    - One operand is a string, so JavaScript converts the number `5` into the string `"5"` and performs **string concatenation**:
        - `"5" + "7"` → `"57"`

Final Result:

`3 + 2 + "7" === "57"`

Key Concepts:

- `+` is both a numeric **addition** and **string concatenation** operator.
- If either operand is a string, it **coerces the other to a string** and **concatenates**.

To force numeric addition:

If you want the result to be numeric, do:

```
3 + 2 + Number("7") // => 12
```

---

25. Slice vs Splice in javascript ?

✅ **`slice(start, end)`**

- Extracts a **portion** of the array.
- Does **not** change the original array.
- end is **not included**.

```
const arr = [1, 2, 3, 4, 5];
const sliced = arr.slice(1, 4);  // [2, 3, 4]
console.log(arr);                // [1, 2, 3, 4, 5] - unchanged
```

✅ **splice(start, deleteCount, ...items)**

- **Changes** the original array.
- Can **remove**, **replace**, or **add** elements

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

// Remove 2 items starting from index 1
const removed = arr.splice(1, 2); // [2, 3]
console.log(arr);                 // [1, 4, 5]

// Insert items at index 1
arr.splice(1, 0, "a", "b");
console.log(arr); // [1, "a", "b", 4, 5]
```

---

26. What is destructuring ?

Destructuring is a JavaScript feature that allows you to **extract values from arrays or objects** and assign them to variables in a **clean, readable way**.

✅ **1. Array Destructuring**

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

const [a, b, c] = arr;
console.log(a); // 🔸 10
console.log(b); // 🔸 20
```

🔹 **Skip elements:**

```
const [x, , z] = [1, 2, 3];
console.log(z); // 🔸 3
```

🔹 **Default values:**

```
const [name = "Guest"] = [];
console.log(name); // 🔸 "Guest"
```

✅ **2. Object Destructuring**

```
const user = { name: "Alice", age: 25 };

const { name, age } = user;
console.log(name); // 🔸 "Alice"
```

🔹 **Rename variables:**

```
const { name: username } = user;
console.log(username); // 🔸 "Alice"
```

🔹 **Default values:**

```
const { city = "Unknown" } = user;
console.log(city); // 🔸 "Unknown"
```

✅ **3. Function Parameter Destructuring**

```
function greet({ name, age }) {
  console.log(`Hi ${name}, you are ${age}`);
}

greet({ name: "Bob", age: 30 });
```

✅ **4. Nested Destructuring**

```
const person = {
  name: "Eve",
  address: {
    city: "Paris",
    zip: 75000
  }
};

const {
  address: { city }
} = person;

console.log(city); // 🔸 "Paris"
```

---

27. What is setTimeOut in javascript ?

`setTimeout()` – Run Once After Delay

🔹 **What It Does:**

Executes a function **once** after a specified delay (in milliseconds).

✅ **Syntax:**

javascript

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

✅ **Example:**

javascript

```
setTimeout(() => {
  console.log("Hello after 2 seconds");
}, 2000);
```

🔁 Runs **once** after 2 seconds.

---

28. What is setInterval in javascript ?

🔁 `setInterval()` – Run Repeatedly After Delay

🔹 **What It Does:**

Executes a function **repeatedly** at specified time intervals.

✅ **Syntax:**

javascript

```
setInterval(callback, delay);
```

✅ **Example:**

javascript

```
setInterval(() => {
  console.log("Repeating every 1 second");
}, 1000);
```

❗ Runs **every 1 second** until manually stopped.

🔹 **To stop it:**

```
const intervalId = setInterval(...);
clearInterval(intervalId);
```

---

29. What are Promises in javascript ?

⏳ 3. Promises – Handle Asynchronous Results

🔹 **What It Does:**

Represents the **result of an asynchronous operation** (that may complete now, later, or never).

✅ **States of a Promise:**

- `pending`
- `fulfilled`
- `rejected`

✅ **Creating a Promise:**

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Success!");
  }, 1000);
});
```

✅ **Consuming a Promise:**

```
promise
  .then(result => console.log(result))    // 🔸 Success!
  .catch(error => console.error(error));  // If rejected
```

---

30. What is a callstack in javascript ?

The call stack is like a to-do list for JavaScript.
It keeps track of which function is currently running and where to go next.

JavaScript runs one thing at a time — it's single-threaded — so the call stack helps manage that.

🧪 Simple Example

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

function askName() {
  console.log("What is your name?");
}

greet();
```

🪜 Step-by-Step What Happens:

1. greet() is called → it's added to the call stack
2. Inside greet(), console.log("Hello") runs

3. Then askName() is called → added to the stack
4. Inside askName(), console.log("What is your name?") runs
5. askName() finishes → removed from the stack
6. greet() finishes → removed from the stack

📦 Call Stack Visual

```
Start:            []

Call greet():     [greet]
Call askName():   [askName, greet]
askName done:     [greet]
greet done:       []
```

💥 Bonus: Stack Overflow Example

```
function repeat() {
  repeat();
}

repeat(); // ❌ Error: Maximum call stack size exceeded
```

This happens because the function keeps calling itself forever → stack never clears → crash.

🧠 Summary

- The call stack runs functions in order, one at a time.
- New function → pushed onto the stack.
- Function finishes → popped off the stack.
- Too many calls → stack overflow.