

JAVASCRIPT INTERVIEW QUESTIONS



1. What is JavaScript?

JavaScript is a **high-level**, interpreted programming language used to create interactive and dynamic web content. It is a versatile, lightweight, and event-driven language that can run on both the client and server sides.



```
console.log("Hello, JavaScript!");
```

2. What are the **data types** supported by JavaScript?

JavaScript supports the following **data types**:

- **Primitive Types:** String, Number, BigInt, Boolean, Undefined, Null, Symbol.
- **Non-Primitive Types:** Object (including arrays, functions).



```
let num = 42; // Number  
let name = "Siva"; // String  
let isStudent = true; // Boolean  
let undef; // Undefined  
let obj = { key: "value" }; // Object
```

3. What is the difference between **let**, **const**, and **var**?

- **var**: Function-scoped, allows redeclaration, hoisted with undefined.
- **let**: Block-scoped, no redeclaration, hoisted but not initialized.
- **const**: Block-scoped, no redeclaration, and must be initialized during declaration.

```
var a = 10;  
let b = 20;  
const c = 30;
```

4. Explain how **==** and **===** differ.

- **==** checks for value equality with type coercion.
- **===** checks for strict equality without type coercion

```
console.log(5 == "5"); // true  
console.log(5 === "5"); // false
```

5. What is a closure?

A **closure** is a function that retains access to its outer scope variables even after the outer function has executed.

```
● ● ●  
function outer() {  
  let count = 0;  
  return function inner() {  
    count++;  
    return count;  
  };  
}  
const counter = outer();  
console.log(counter()); // 1  
console.log(counter()); // 2
```

6. What is hoisting?

Hoisting is JavaScript's behavior of moving variable and function declarations to the top of their scope during compilation.

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

7. Explain the concept of **this** in JavaScript.

this refers to the context in which a function is executed. Its value depends on how the function is called.

```
● ● ●  
const obj = {  
  name: "Siva",  
  greet() {  
    console.log(this.name);  
  },  
};  
obj.greet(); // Siva
```

8. What are JavaScript prototypes?

Prototypes allow objects to inherit properties and methods from other objects.

```
● ● ●  
  
function Person(name) {  
  this.name = name;  
}  
Person.prototype.greet = function () {  
  return `Hello, ${this.name}`;  
};  
const person = new Person("Siva");  
console.log(person.greet()); // Hello, Siva
```

9. What is the difference between `null` and `undefined`?

- `null`: A deliberate non-value.
- `undefined`: A variable declared but not assigned a value.



```
let x = null;
let y;
console.log(x); // null
console.log(y); // undefined
```

10. How does JavaScript handle asynchronous operations?

JavaScript uses the event loop with callbacks, promises, and `async/await` for asynchronous operations.

Example: Using a Callback



```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data fetched");
  }, 2000);
}

fetchData((data) => {
  console.log(data); // Logs "Data fetched" after 2 seconds
});
```

Example: Using a Promise

```
● ● ●

function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched");
    }, 2000);
  });
}

fetchData().then((data) => {
  console.log(data); // Logs "Data fetched" after 2 seconds
});
```

Example: Using async/await

```
● ● ●

function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched");
    }, 2000);
  });
}

async function getData() {
  const data = await fetchData();
  console.log(data); // Logs "Data fetched" after 2 seconds
}

getData();
```

11. What is a promise?

A **promise** represents the eventual completion or failure of an asynchronous operation.

```
● ● ●  
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => resolve("Success"), 1000);  
});  
promise.then(console.log); // Success
```

12. What are **async/await** functions?

async/await allows writing asynchronous code that looks synchronous.

```
● ● ●  
async function fetchData() {  
  const data = await fetch("https://api.example.com");  
  return data.json();  
}
```

13. Explain event delegation in JavaScript.

Event delegation allows you to handle events for multiple child elements at the parent level.



```
document.getElementById("parent").addEventListener("click", (e) => {
  if (e.target.tagName === "BUTTON") {
    console.log("Button clicked!");
  }
});
```

14. What are JavaScript modules?

Modules allow you to organize code into reusable files using import and export



```
//Example: module.js
export const greet = () => "Hello, Module!";

//main.js
import { greet } from "./module.js";
console.log(greet());
```

15. How can you prevent a function from being called multiple times?

You can use a debounce function.

```
function debounce(func, delay) {  
  let timeout;  
  return (...args) => {  
    clearTimeout(timeout);  
    timeout = setTimeout(() => func(...args), delay);  
  };  
}
```

16. What is the event loop?

The event loop processes tasks from the queue and stack for asynchronous operations.

```
console.log("Start");  
setTimeout(() => console.log("Timeout"), 0);  
console.log("End");
```

17. What is the difference between **apply()** and **call()** methods?

- **call()**: Invokes a function with a specific this value and arguments passed individually.
- **apply()**: Similar to call(), but arguments are passed as an array

```
● ● ●

function greet(greeting, punctuation) {
  return `${greeting}, ${this.name}${punctuation}`;
}
const person = { name: "Siva" };

console.log(greet.call(person, "Hello", "!")); // Hello, Siva!
console.log(greet.apply(person, ["Hi", "."])); // Hi, Siva.
```

18. What is **bind()** method used for?

The bind() method creates a new function with a specific this value and optional arguments.

```
● ● ●

const obj = { name: "Siva" };
function greet(greeting) {
  return `${greeting}, ${this.name}`;
}
const boundGreet = greet.bind(obj);
console.log(boundGreet("Hello")); // Hello, Siva
```

19. What is a JavaScript event loop?

The event loop continuously checks the call stack and the task queue, executing tasks from the queue when the stack is empty.



```
console.log("Start"); // Executed first, added to the call stack

setTimeout(() => {
  console.log("Timeout callback"); // Added to the task queue, executed later
}, 1000);

console.log("End"); // Executed immediately after "Start"
```

20. Explain the concept of "event bubbling" and "event capturing".

- **Event Bubbling:** Events propagate from the target element to the parent elements.
- **Event Capturing:** Events propagate from the parent elements to the target element.



```
document.getElementById("child").addEventListener("click", () =>
  console.log("Child"), true); // Capturing
document.getElementById("parent").addEventListener("click", () =>
  console.log("Parent")); // Bubbling
```

21. What is the difference between **deep copy** and **shallow copy**?

- **Shallow Copy:** Copies only the first layer of an object.
- **Deep Copy:** Copies all layers of an object.



```
let obj = { a: 1, b: { c: 2 } };
let shallow = { ...obj };
let deep = JSON.parse(JSON.stringify(obj));
```

22. What are **generator functions**?

Generators are special functions that can pause execution and resume later.



```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = generator();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

23. What is the **new** keyword used for?

The **new** keyword creates an instance of an object from a constructor function.

```
● ● ●  
function Person(name) {  
  this.name = name;  
}  
const person = new Person("Siva");  
console.log(person.name); // Siva
```

24. How do JavaScript's **setTimeout** and **setInterval** work?

- **setTimeout**: Executes a function after a specified delay.
- **setInterval**: Repeats execution at specified intervals.

```
● ● ●  
setTimeout(() => console.log("Timeout"), 1000);  
setInterval(() => console.log("Interval"), 2000);
```

25. What is a **WeakMap** and how is it different from a **Map**?

- **WeakMap**: Keys are only objects and are garbage collected.
- **Map**: Keys can be any type



```
let obj = { key: "value" };
let weakMap = new WeakMap();
weakMap.set(obj, "data");
```

26. What is a **Set** in JavaScript?

A **Set** is a collection of unique values.



```
const set = new Set([1, 2, 3, 2]);
console.log(set); // Set { 1, 2, 3 }
```

27. What is Object.create() used for?

It creates a new object with a specified prototype.



```
const proto = { greet: () => "Hello" };
const obj = Object.create(proto);
console.log(obj.greet()); // Hello
```

28. How does JavaScript's garbage collection work?

JavaScript uses a mark-and-sweep algorithm to identify and remove unused objects.



```
//Example: Garbage Collection in Action
function createObject() {
  const obj = { name: "Siva" }; // Memory is allocated for `obj`
  console.log(obj.name); // "Siva" is accessible here
}

createObject(); // After execution, `obj` is no longer accessible, so it is
// garbage-collected.
```



```
//Example: Removing References to Enable Garbage Collection
let obj1 = { name: "Siva" };
let obj2 = obj1; // `obj2` references the same object as `obj1`

console.log(obj1); // Accessible
console.log(obj2); // Accessible

obj1 = null; // The object is still not garbage-collected because `obj2` references it

obj2 = null; // Now the object is unreachable and will be garbage-collected
```

29. What are "decorators" in JavaScript?

Decorators are functions that modify classes or methods.
They are experimental features.



```
function decorator(target) {
  target.isDecorated = true;
}

@decorator
class Example {}

console.log(Example.isDecorated); // true
```

30. Explain the difference between **prototype** and **proto**.

- **prototype**: An object associated with functions for inheritance.
- **proto**: A reference to the object's prototype.



```
function Person() {}  
const person = new Person();  
console.log(person.__proto__ === Person.prototype); // true
```