JavaScript Interview Questions and Answers

1. What is JavaScript?

JavaScript is a lightweight, interpreted programming language used to create dynamic web pages. It enables interactivity, like form validations, animations, and real-time updates.

2. What are the differences between `var`, `let`, and `const`?

- `var`: Function-scoped, can be redeclared and updated.
- `let`: Block-scoped, cannot be redeclared but can be updated.
- `const`: Block-scoped, cannot be redeclared or updated.

3. Explain closures in JavaScript.

A closure is a function that has access to its outer function's variables, even after the outer function has returned.

Example:

javascript

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

4. What is the difference between `==` and `===`?

- `==` checks for equality with type coercion (e.g., `5 == '5'` is true).
- `===` checks for strict equality without type coercion (e.g., `5 === '5'` is false).

5. What are promises in JavaScript?

Promises are objects representing the eventual completion or failure of an asynchronous operation.

Example:

javascript

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  success ? resolve("Done") : reject("Error");
});
promise.then((res) => console.log(res)).catch((err) =>
console.error(err));
```

6. Explain the concept of hoisting.
Hoisting is JavaScript's default behavior of moving declarations to the top of the scope. Variables declared with `var` are hoisted, but only their declarations, not initializations.
Example:
javascript

```
console.log(a); // undefined
var a = 10;
```

7. What is the DOM?
The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page as a tree structure, allowing scripts to update content, structure, and styles dynamically.
8. How does the `this` keyword work in JavaScript?
The value of `this` depends on how the function is called:

- In a method: Refers to the object.
- Alone: Refers to the global object (`window` in browsers).
- In strict mode: `this` is `undefined` for standalone functions.
- Arrow functions: Inherit `this` from their lexical scope.

9. What is event delegation?
Event delegation is a technique where a parent element handles events for its child elements using a single event listener.
Example:
javascript

```
document.getElementById("parent").addEventListener("click", function
(e) {
  if (e.target && e.target.matches("button.classname")) {
    console.log("Button clicked");
  }
});
```

10. What are JavaScript data types?
Primitive: String, Number, Boolean, Null, Undefined, Symbol, BigInt.
Non-Primitive: Objects (e.g., Arrays, Functions).
11. What is the event loop in JavaScript?

The event loop is a mechanism that handles asynchronous operations. It ensures that functions in the call stack execute, while deferred functions wait in the callback queue until the stack is clear.

12. How do you use `fetch` for API calls?

Example:

javascript

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

13. What are ES6 features?

Some key ES6 features include:

- Arrow functions
- Template literals
- Default parameters
- Rest and spread operators
- Destructuring assignment
- Modules (`import` and `export`)

14. What are higher-order functions?

A higher-order function is a function that takes other functions as arguments or returns a function as output.

Example:

javascript

```
function higherOrder(fn) {
  return fn();
}
higherOrder(() => console.log("Hello"));
```

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

- `null`: Represents the intentional absence of any value.
- `undefined`: Represents an uninitialized variable.

16. What is an IIFE (Immediately Invoked Function Expression)?

An IIFE is a function that is defined and immediately executed. It is often used to create a private scope.

Example:

javascript

```
(function() {
  console.log("I am an IIFE");
})();
```

---

17. What are template literals in JavaScript?
Template literals are string literals that allow embedded expressions, multi-line strings, and string interpolation using backticks (`` ` ``).
Example:
javascript

```
let name = "John";
let greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, John!
```

---

18. Explain the difference between synchronous and asynchronous programming.

- **Synchronous**: Tasks are executed one after the other.
- **Asynchronous**: Tasks run independently, allowing other tasks to run concurrently, without blocking the main thread. Promises, callbacks, and `async/await` are used for async programming.

---

19. What is the difference between `apply()`, `call()`, and `bind()`?

- **`call()`**: Invokes a function immediately, with a specified `this` value and arguments.
- **`apply()`**: Similar to `call()`, but arguments are passed as an array.
- **`bind()`**: Returns a new function with a specified `this` value and arguments, but does not invoke it immediately.

Example:
javascript

```
function greet(name) {
  console.log(`Hello, ${name}`);
}
greet.call(null, "John"); // Hello, John
greet.apply(null, ["John"]); // Hello, John
const boundGreet = greet.bind(null, "John");
boundGreet(); // Hello, John
```

---

20. What is an arrow function and how does it differ from regular functions?

Arrow functions are a concise way to write functions using the `=>` syntax. They do not have their own `this`, but inherit it from the outer scope.
Example:
javascript

```
const sum = (a, b) => a + b;
console.log(sum(2, 3)); // 5
```

---

21. What are JavaScript's data structures?
JavaScript provides several data structures such as:

- Arrays
- Objects
- Maps
- Sets

Each of these serves different purposes in storing and manipulating data efficiently.

---

22. What is the difference between `Object.freeze()` and `Object.seal()`?

- **`Object.freeze()`**: Prevents modifications to the object, including adding, deleting, or changing properties.
- **`Object.seal()`**: Prevents adding or deleting properties, but existing properties can still be modified.

---

23. What is a callback function in JavaScript?
A callback function is a function passed into another function as an argument to be executed later. It is commonly used in asynchronous operations.
Example:
javascript

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}
fetchData((message) => console.log(message)); // Data received
```

---

24. What is `async/await` in JavaScript?
`async/await` is a syntax for handling asynchronous operations in a more readable manner.

- `async` makes a function return a promise.

- `await` is used inside `async` functions to pause execution until a promise resolves.
  Example:

javascript

```
async function fetchData() {
  let response = await fetch('https://api.example.com/data');
  let data = await response.json();
  console.log(data);
}
fetchData();
```

---

25. What are the various ways to create an object in JavaScript?

Using object literals:
javascript

```
let person = { name: "John", age: 30 };
```

-

Using `new Object()`:
javascript

```
let person = new Object();
person.name = "John";
person.age = 30;
```

-

Using a constructor function:
javascript

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
let person = new Person("John", 30);
```

-

---

26. What is the purpose of `setTimeout()` and `setInterval()`?

- **`setTimeout()`**: Executes a function after a specified delay (in milliseconds).
- **`setInterval()`**: Repeatedly executes a function at specified intervals (in milliseconds).
  Example:

javascript

```
setTimeout(() => console.log("This runs after 2 seconds"), 2000);
setInterval(() => console.log("This runs every 3 seconds"), 3000);
```

---

27. Explain event bubbling and event capturing.

- **Event Bubbling**: The event starts from the target element and propagates up to the root.
- **Event Capturing**: The event starts from the root element and propagates down to the target element.
  You can control this behavior using the `capture` parameter in `addEventListener()`.

---

28. What is the `window` object in JavaScript?

The `window` object represents the browser window and provides methods and properties for manipulating it, such as `window.alert()`, `window.location`, and `window.document`.

---

29. How do you implement inheritance in JavaScript?

JavaScript supports inheritance through prototypes. In ES6, you can also use the `class` syntax.
Example:
javascript

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}
class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}
const dog = new Dog("Buddy");
dog.speak(); // Buddy barks.
```

---

30. What is a `Map` in JavaScript?

A `Map` is a collection of key-value pairs where both the keys and values can be any data type. It maintains the insertion order of its elements.
Example:
javascript

```
let map = new Map();
map.set("name", "John");
map.set("age", 30);
console.log(map.get("name")); // John
```

31. What is the `reduce()` method in JavaScript, and how does it work?

The `reduce()` method executes a reducer function on each element of an array, resulting in a single output value.
Syntax:
javascript

```
array.reduce(callback(accumulator, currentValue[, index[, array]])[,
initialValue])
```

- **Parameters:**
  - `callback`: Function executed on each array element.
  - `accumulator`: The accumulated result of the reducer function.
  - `currentValue`: The current element being processed.
  - `initialValue` (optional): Value to use as the initial accumulator value.

Example:
javascript

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

---

32. What is the `Promise` object in JavaScript?

A `Promise` represents a value that may be available now, or in the future, or never. It helps manage asynchronous operations.
States of a Promise:

1. **Pending**: Initial state, neither fulfilled nor rejected.
2. **Fulfilled**: Operation completed successfully.
3. **Rejected**: Operation failed.

Example:
javascript

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) resolve("Operation succeeded!");
  else reject("Operation failed!");
});

promise
  .then((result) => console.log(result)) // Operation succeeded!
  .catch((error) => console.log(error));
```

---

33. What is the difference between `null` and `undefined` in JavaScript?

`null`: Represents the intentional absence of a value.
javascript

```
let value = null;
console.log(typeof value); // object
```

-

`undefined`: Represents a variable that has been declared but not yet assigned a value.
javascript

```
let value;
console.log(value); // undefined
```

-

---

34. What are JavaScript modules, and why are they useful?

Modules are reusable pieces of code that can be exported from one file and imported into another. They allow better organization, code reuse, and maintainability.

Example (ES6 modules):
File: math.js
javascript

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

File: main.js
javascript

```
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

---

35. Explain the `event loop` in JavaScript.

The event loop is a mechanism that manages the execution of JavaScript code, particularly handling asynchronous operations like promises and `setTimeout`.

1. **Call Stack**: Executes synchronous code.
2. **Task Queue**: Queues asynchronous callbacks for execution after the current stack clears.
3. **Microtask Queue**: Handles tasks like resolved promises before the task queue.

Example:

javascript

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

Output:

sql

```
Start
End
Promise
Timeout
```

---

36. What are closures in JavaScript?

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

Example:

javascript

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

```javascript
}
let increment = outer();
increment(); // 1
increment(); // 2
```

---

37. How does JavaScript handle memory management?

JavaScript uses **garbage collection** to manage memory automatically.

- **Heap**: Stores objects and functions.
- **Stack**: Stores primitive values and function calls.

The garbage collector removes objects that are no longer reachable. Cyclic references can sometimes cause issues.

---

38. What are `Prototypes` in JavaScript?

Prototypes are objects that other objects can inherit properties and methods from.
Every JavaScript object has a hidden property `[[Prototype]]` that refers to its prototype.
Example:
javascript

```javascript
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}`);
};
let john = new Person("John");
john.greet(); // Hello, my name is John
```

---

39. What is `debouncing` in JavaScript?

Debouncing is a technique to limit the frequency of a function execution, ensuring it only runs after a specified time has passed since the last invocation.
Example:
javascript

```javascript
function debounce(func, delay) {
  let timeout;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}
let log = debounce(() => console.log("Debounced!"), 300);
window.addEventListener("resize", log);
```

---

40. What is `currying` in JavaScript?
Currying is a technique of transforming a function with multiple arguments into a sequence of functions, each taking a single argument.
Example:
javascript

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return (...nextArgs) => curried(...args, ...nextArgs);
    }
  };
}
function sum(a, b, c) {
  return a + b + c;
}
let curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3)); // 6
```

41. What is the difference between `==` and `===` in JavaScript?

`==` (Equality): Compares two values for equality after type coercion.
javascript

```
console.log(5 == '5'); // true (type coercion happens)
```

-

`===` (Strict Equality): Compares two values for equality without type coercion.
javascript

```
console.log(5 === '5'); // false (no type coercion)
```

-

---

42. What are JavaScript `Promises`, and how are they different from callbacks?

Promises simplify asynchronous operations and avoid "callback hell". A `Promise` has three states: pending, fulfilled, and rejected.
Callbacks, on the other hand, are functions passed as arguments to handle asynchronous results.

Example with Promise:

javascript

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    let success = true;
    if (success) resolve("Data fetched");
    else reject("Error fetching data");
  });
};

fetchData()
  .then((res) => console.log(res))
  .catch((err) => console.log(err));
```

---

43. Explain `async` and `await` in JavaScript.

`async` and `await` make asynchronous code look synchronous.

- **`async`**: Declares a function as asynchronous.
- **`await`**: Pauses execution until the `Promise` resolves.

Example:

javascript

```
async function fetchData() {
  let result = await new Promise((resolve) => setTimeout(() =>
resolve("Fetched"), 1000));
  console.log(result);
}
fetchData(); // Fetched (after 1 second)
```

---

44. What is `event delegation` in JavaScript?

Event delegation is a technique where a parent element handles events for its child elements. This improves performance and simplifies code.

Example:

javascript

```
document.getElementById("parent").addEventListener("click", (e) => {
  if (e.target && e.target.matches("button.child")) {
    console.log("Child button clicked!");
  }
});
```

---

45. How do you clone an object in JavaScript?

Cloning creates a copy of an object.

Shallow Clone:

javascript

```
const obj = { a: 1, b: 2 };
const shallowClone = { ...obj };
console.log(shallowClone); // { a: 1, b: 2 }
```

-

Deep Clone (for nested objects):

javascript

```
const obj = { a: 1, b: { c: 2 } };
const deepClone = JSON.parse(JSON.stringify(obj));
console.log(deepClone); // { a: 1, b: { c: 2 } }
```

-

---

46. What are `default parameters` in JavaScript?

Default parameters allow functions to use default values if no argument is provided.

Example:

javascript

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

---

47. What are `setTimeout` and `setInterval`?

`setTimeout`: Executes a function after a specified delay.
javascript

```
setTimeout(() => console.log("Delayed Execution"), 1000);
```

-

`setInterval`: Repeatedly executes a function at fixed intervals.
javascript

```
setInterval(() => console.log("Repeated Execution"), 1000);
```

-

---

48. What is `JSON`, and how is it used in JavaScript?

JSON (JavaScript Object Notation) is a format for data exchange.
Convert Object to JSON (stringify):
javascript

```
const obj = { name: "Alice", age: 25 };
const jsonString = JSON.stringify(obj);
console.log(jsonString); // '{"name":"Alice","age":25}'
```

-

Convert JSON to Object (parse):
javascript

```
const json = '{"name":"Alice","age":25}';
const obj = JSON.parse(json);
console.log(obj); // { name: "Alice", age: 25 }
```

-

---

49. What are `Arrow Functions`, and how are they different from regular functions?

Arrow functions provide a concise syntax and do not have their own `this` or `arguments`.
Example:
javascript

```
const regular = function () {
  console.log("Regular Function");
};

const arrow = () => {
  console.log("Arrow Function");
};

arrow();
regular();
```

---

50. What is the `spread operator` in JavaScript?

The spread operator (`...`) expands iterable elements like arrays or objects.
Example:
javascript

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];
console.log(arr2); // [1, 2, 3, 4, 5]

const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 };
console.log(obj2); // { a: 1, b: 2, c: 3 }
```

51. What is the `rest operator` in JavaScript?

The rest operator (`...`) collects all remaining elements into an array or object.
Example:
javascript

```
function sum(...args) {
  return args.reduce((acc, val) => acc + val, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

In objects:
javascript

```
const { a, ...rest } = { a: 1, b: 2, c: 3 };
console.log(rest); // { b: 2, c: 3 }
```

---

52. What is the difference between `let`, `const`, and `var`?

`var`: Function-scoped, can be redeclared, and hoisted.
javascript

```
var x = 10;
var x = 20; // No error
```

-

`let`: Block-scoped, cannot be redeclared in the same scope, hoisted but uninitialized.
javascript

```
let y = 10;
// let y = 20; // Error
```

-

`const`: Block-scoped, cannot be redeclared or reassigned.
javascript

```
const z = 10;
// z = 20; // Error
```

-

---

53. What are `closures` in JavaScript?

A closure is a function that retains access to its outer scope, even after the outer function has returned.

Example:
javascript

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

const counter = outer();
counter(); // 1
counter(); // 2
```

---

54. Explain the `event loop` in JavaScript.

The event loop is responsible for executing code, collecting and processing events, and handling asynchronous operations.

1. Synchronous code is executed first in the **call stack**.
2. Asynchronous operations (e.g., `setTimeout`, Promises) are sent to **Web APIs** or **task queues**.
3. The event loop checks if the call stack is empty and pushes queued tasks back to it.

---

55. What are `modules` in JavaScript?

Modules allow code to be split into reusable pieces.

Export:
javascript

```
export const greet = () => console.log("Hello");
```

-

Import:
javascript

```
import { greet } from './module.js';
greet(); // Hello
```

-

---

56. What is `debouncing` in JavaScript?

Debouncing ensures that a function executes only after a specified delay. It's commonly used in event handling (e.g., input or scroll).

Example:
javascript

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

const handleResize = debounce(() => console.log("Resized"), 500);
window.addEventListener("resize", handleResize);
```

---

57. What is `hoisting` in JavaScript?

Hoisting allows variable and function declarations to be used before they are defined in the code.

Function Hoisting:
javascript

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

-

Variable Hoisting:
javascript

```
console.log(a); // undefined
var a = 10;
```

-

---

58. What is the purpose of `strict mode` in JavaScript?

`"use strict";` makes JavaScript behave more strictly, catching errors and unsafe actions.

Example:
javascript

```
"use strict";
x = 10; // Error: x is not declared
```

---

59. What is `prototype` in JavaScript?

Every object in JavaScript has a `prototype`, which is an object from which it can inherit properties and methods.

Example:
javascript

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  console.log(`Hello, ${this.name}`);
};

const alice = new Person("Alice");
```

```
alice.greet(); // Hello, Alice
```

---

60. Explain `currying` in JavaScript.

Currying transforms a function so it can be called with fewer arguments at a time.
Example:
javascript

```
function multiply(a) {
  return function (b) {
    return a * b;
  };
}

const double = multiply(2);
console.log(double(3)); // 6
```

---

61. How can you handle errors in JavaScript?

Use `try-catch` blocks for error handling.
Example:
javascript

```
try {
  throw new Error("Something went wrong");
} catch (error) {
  console.log(error.message); // Something went wrong
}
```

---

62. What are `Generators` in JavaScript?

Generators are functions that can be paused and resumed using `yield`.
Example:
javascript

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

const gen = generator();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

---

63. What is the difference between `call`, `apply`, and `bind`?

`call`: Invokes a function with a specific `this` value and arguments.
javascript

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}
const user = { name: "Alice" };
greet.call(user, "Hello"); // Hello, Alice
```

-

`apply`: Similar to `call`, but arguments are passed as an array.
javascript

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

-

`bind`: Returns a new function with `this` bound to the provided value.
javascript

```
const boundGreet = greet.bind(user, "Hey");
```

- `boundGreet(); // Hey, Alice`

64. What is `memoization` in JavaScript?

Memoization is an optimization technique to cache the results of expensive function calls for reuse.

Example:

javascript

```
function memoize(fn) {
  const cache = {};
  return function (...args) {
    const key = JSON.stringify(args);
    if (cache[key]) return cache[key];
    const result = fn(...args);
    cache[key] = result;
    return result;
  };
}

const factorial = memoize((n) => (n <= 1 ? 1 : n * factorial(n - 1)));
console.log(factorial(5)); // 120
console.log(factorial(5)); // Cached: 120
```

---

65. What is `event delegation` in JavaScript?

Event delegation is a technique where a parent element handles events for its child elements using event bubbling.

Example:

javascript

```
document.getElementById("parent").addEventListener("click", (e) => {
  if (e.target && e.target.matches("button")) {
    console.log(`Button ${e.target.textContent} clicked`);
  }
});
```

---

66. What are `promises` in JavaScript?

Promises represent the eventual completion (or failure) of an asynchronous operation.

Example:

javascript

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Data fetched"), 1000);
});

fetchData
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

---

67. What is the difference between `setTimeout` and `setInterval`?

`setTimeout`: Executes a function once after a specified delay.
javascript

```
setTimeout(() => console.log("Executed after 1 second"), 1000);
```

-

`setInterval`: Repeatedly executes a function at a fixed interval.
javascript

```
setInterval(() => console.log("Runs every second"), 1000);
```

-

---

68. What are `arrow functions`, and how do they differ from regular functions?

Arrow functions are concise alternatives to regular functions. They do not bind their own `this`, `arguments`, or `super`.

Example:
javascript

```
const greet = (name) => `Hello, ${name}`;
console.log(greet("Alice")); // Hello, Alice
```

Differences:

- Arrow functions cannot be used as constructors.
- They do not have their own `this`.

---

69. What is the purpose of `async/await`?

`async/await` simplifies working with Promises by providing a way to write asynchronous code that looks synchronous.

Example:
javascript

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

```
fetchData();
```

---

70. What is `strict equality` (`===`) in JavaScript?

The `===` operator checks for both value and type equality.
Example:
javascript

```
console.log(1 === "1"); // false
console.log(1 === 1); // true
```

---

71. What is the `spread operator` in JavaScript?

The spread operator (`...`) expands elements of an array or object into individual elements.
Example (arrays):
javascript

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2); // [1, 2, 3, 4]
```

Example (objects):
javascript

```
const obj1 = { a: 1 };
const obj2 = { ...obj1, b: 2 };
console.log(obj2); // { a: 1, b: 2 }
```

---

72. What are `immediately invoked function expressions` (IIFE)?

IIFEs are functions executed immediately after their definition.
Example:
javascript

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

---

73. Explain the `this` keyword in JavaScript.

The `this` keyword refers to the object that is executing the current function.

Examples:

Global context:
javascript

```
console.log(this); // Window (in browsers)
```

-

Object method:
javascript

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

-

---

74. What is the `Map` object in JavaScript?

`Map` is a collection of key-value pairs where keys can be of any type.

Example:

javascript

```
const map = new Map();
map.set("name", "Alice");
console.log(map.get("name")); // Alice
```

---

75. What is the difference between `for...of` and `for...in`?

`for...of`: Iterates over iterable objects like arrays, strings, etc.
javascript

```
for (const val of [1, 2, 3]) {
  console.log(val); // 1, 2, 3
}
```

-

`for...in`: Iterates over the keys of an object.
javascript

```
for (const key in { a: 1, b: 2 }) {
  console.log(key); // a, b
```
- }

Advanced JavaScript Concepts with Explanations and Examples

---

1. What is the JavaScript Event Loop?

The Event Loop is a mechanism that handles asynchronous operations in JavaScript, allowing non-blocking behavior despite being single-threaded.

Explanation:

- The Call Stack executes synchronous code.
- Async operations (like `setTimeout`, Promises) go to the Web APIs.
- Once completed, their callback is queued in the **Task Queue** or **Microtask Queue** for execution.
- The Event Loop ensures the Call Stack is empty before processing tasks from the queue.

Example:

javascript

```javascript
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise resolved");
});

console.log("End");

// Output: Start, End, Promise resolved, Timeout
```

---

2. What is `currying` in JavaScript?

Currying transforms a function with multiple arguments into a series of functions that each take a single argument.

Example:

javascript

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn(...args);
    } else {
      return (...nextArgs) => curried(...args, ...nextArgs);
    }
  };
}

const sum = (a, b, c) => a + b + c;
const curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3)); // 6
```

---

3. What are Prototypes in JavaScript?

Prototypes enable inheritance in JavaScript. Every object has an internal link to another object called its prototype.

Example:

javascript

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

Person.prototype.greet = function () {
  console.log(`Hello, ${this.name}`);
};

const alice = new Person("Alice");
alice.greet(); // Hello, Alice
```

---

4. What is Debouncing in JavaScript?

Debouncing ensures that a function is executed after a specified delay, preventing it from being called repeatedly.

Example:

javascript

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

const log = debounce((message) => console.log(message), 300);
log("Hello"); // Executes after 300ms
```

---

5. What is Throttling in JavaScript?

Throttling ensures a function is called at most once in a specified time period, regardless of how many times it's triggered.

Example:

javascript

```
function throttle(fn, interval) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= interval) {
      lastCall = now;
      fn(...args);
    }
  };
}

const log = throttle((message) => console.log(message), 1000);
log("Hello"); // Executes immediately
log("Hello Again"); // Ignored if called within 1 second
```

---

6. What is `Closure` in JavaScript?

A closure is a function that retains access to its parent scope, even after the parent function has executed.

Example:

javascript

```
function outer() {
  const message = "Hello, Closure!";
  return function inner() {
    console.log(message);
  };
}

const greet = outer();
greet(); // Hello, Closure!
```

---

7. What is `Hoisting` in JavaScript?

Hoisting moves variable and function declarations to the top of their scope during the compile phase.

Example:

javascript

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

greet(); // Works due to hoisting
function greet() {
  console.log("Hi");
}
```

---

8. Explain `call`, `apply`, and `bind` methods.

These methods set the value of `this` in a function.

`call`: Invokes a function with arguments passed individually.

javascript

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

```javascript
const user = { name: "Alice" };
greet.call(user, "Hello"); // Hello, Alice
```

-

`apply`: Invokes a function with arguments passed as an array.
javascript

```javascript
greet.apply(user, ["Hi"]); // Hi, Alice
```

-

`bind`: Returns a new function with `this` bound to the specified object.
javascript

```javascript
const boundGreet = greet.bind(user);
boundGreet("Hey"); // Hey, Alice
```

-

---

9. Explain the `module pattern` in JavaScript.

The module pattern is a design pattern to create encapsulated code with private variables and public methods.

Example:

javascript

```javascript
const Counter = (() => {
  let count = 0;
  return {
    increment() {
      count++;
      console.log(count);
    },
    reset() {
      count = 0;
    },
  };
})();

Counter.increment(); // 1
Counter.increment(); // 2
Counter.reset();
Counter.increment(); // 1
```

10. What are Generators in JavaScript?

Generators are functions that can pause execution (`yield`) and resume later (`next`).

Example:

javascript

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

const gen = generatorFunction();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

11. What is `WeakMap` and `WeakSet`?

- **WeakMap:** Stores key-value pairs where keys are weakly referenced objects. Keys cannot be primitives.
- **WeakSet:** Stores unique objects, also weakly referenced.

Example (WeakMap):

javascript

```
const weakMap = new WeakMap();
let obj = {};
weakMap.set(obj, "value");
obj = null; // Entry removed
```

Example (WeakSet):

javascript

```
const weakSet = new WeakSet();
let obj = {};
weakSet.add(obj);
```

```
obj = null; // Entry removed
```

12. What is the `Reflect` API in JavaScript?

The `Reflect` API provides methods to interact with objects in a way similar to built-in operations. It is often used with proxies or to simplify object manipulation.

Example:

javascript

```
const obj = { x: 10 };

// Reflect.set modifies properties
Reflect.set(obj, 'x', 20);
console.log(obj.x); // 20

// Reflect.has checks for property existence
console.log(Reflect.has(obj, 'x')); // true

// Reflect.deleteProperty deletes properties
Reflect.deleteProperty(obj, 'x');
console.log(obj); // {}
```

---

13. What is the `Proxy` object in JavaScript?

A `Proxy` allows custom behavior for fundamental operations (e.g., property access, assignment).

Example:

javascript

```
const handler = {
  get: (target, prop) => {
    return prop in target ? target[prop] : 'Property does not exist';
  },
};

const obj = { name: 'Alice' };
const proxy = new Proxy(obj, handler);

console.log(proxy.name); // Alice
console.log(proxy.age); // Property does not exist
```

---

14. What is `async/await` in JavaScript?

`async/await` simplifies asynchronous programming by making it look synchronous.

Example:

javascript

```
function fetchData() {
  return new Promise((resolve) => setTimeout(() => resolve('Data
loaded'), 1000));
}

async function loadData() {
  console.log('Loading...');
  const data = await fetchData();
  console.log(data);
}

loadData();
// Output: Loading... (1-second pause) Data loaded
```

---

15. Explain `nullish coalescing` and `optional chaining`.

Nullish Coalescing (`??`): Returns the right-hand value if the left-hand value is `null` or `undefined`.

javascript

```
const value = null ?? 'Default';
console.log(value); // Default
```

-

Optional Chaining (`?.`): Safely accesses deeply nested properties without throwing errors.

javascript

```
const obj = { user: { name: 'Alice' } };
console.log(obj.user?.name); // Alice
console.log(obj.user?.age); // undefined
```

-

---

16. What is `Promise.all`, `Promise.race`, `Promise.allSettled`, and `Promise.any`?

`Promise.all`: Resolves when all promises resolve or rejects if any promise fails.
javascript

```
Promise.all([Promise.resolve(1),
Promise.resolve(2)]).then(console.log); // [1, 2]
```

-

`Promise.race`: Resolves/rejects when the first promise resolves/rejects.
javascript

```
Promise.race([Promise.resolve(1),
Promise.reject('Error')]).then(console.log); // 1
```

-

`Promise.allSettled`: Resolves when all promises settle (resolve or reject).
javascript

```
Promise.allSettled([Promise.resolve(1),
Promise.reject('Error')]).then(console.log);
// [{status: 'fulfilled', value: 1}, {status: 'rejected', reason:
'Error'}]
```

-

`Promise.any`: Resolves with the first fulfilled promise or rejects if all fail.
javascript

```
Promise.any([Promise.reject('Error'),
Promise.resolve(1)]).then(console.log); // 1
```

-

---

17. What are Symbols in JavaScript?

Symbols are unique and immutable primitive values often used as object keys.
Example:
javascript

```
const sym = Symbol('id');
const obj = { [sym]: 123 };

console.log(obj[sym]); // 123
```

---

18. Explain `this` in Arrow Functions vs Regular Functions.

- In regular functions, `this` is dynamic and depends on how the function is called.
- In arrow functions, `this` is lexically bound to the context where it was defined.

Example:
javascript

```
const obj = {
  name: 'Alice',
  regular: function () {
    console.log(this.name);
  },
  arrow: () => {
    console.log(this.name);
  },
};

obj.regular(); // Alice
obj.arrow(); // undefined
```

---

19. What is the difference between `==` and `===`?

`==` (Equality): Compares values, performs type conversion if necessary.
javascript

```
console.log(1 == '1'); // true
```

-

`===` (Strict Equality): Compares values without type conversion.
javascript

```
console.log(1 === '1'); // false
```

-

---

20. What is `memoization` in JavaScript?

Memoization is a technique to optimize functions by storing results of expensive operations and reusing them for the same inputs.

Example:
javascript

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

const factorial = memoize((n) => (n <= 1 ? 1 : n * factorial(n - 1)));
console.log(factorial(5)); // 120
```

21. What are Generators in JavaScript?

Generators are special functions that can pause execution and later resume, allowing for lazy evaluation. They are defined using the `function*` syntax.

Example:

javascript

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

const gen = generator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

---

22. What is Event Loop in JavaScript?

The Event Loop is a mechanism that handles asynchronous operations. It ensures that the JavaScript runtime executes code, collects and processes events, and runs queued subtasks.

Key Concepts:

- **Call Stack:** Tracks active function calls.
- **Task Queue:** Holds callbacks from asynchronous operations (e.g., `setTimeout`).
- **Microtask Queue:** Holds higher-priority tasks like `Promise` callbacks.

Example:

javascript

```
console.log('Start');

setTimeout(() => console.log('Timeout'), 0);

Promise.resolve().then(() => console.log('Promise'));

console.log('End');
// Output: Start, End, Promise, Timeout
```

---

23. What are the types of Web Storage in JavaScript?

Web Storage provides mechanisms to store data in the browser:

Local Storage: Stores data persistently across sessions.
javascript

```
localStorage.setItem('key', 'value');
console.log(localStorage.getItem('key')); // value
```

1.

Session Storage: Stores data for the session only (cleared on tab close).
javascript

```
sessionStorage.setItem('key', 'value');
console.log(sessionStorage.getItem('key')); // value
```

2.

---

24. What is `debouncing` in JavaScript?

Debouncing limits the rate at which a function is executed. Useful in scenarios like search input or window resize events.

Example:
javascript

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

```javascript
const log = debounce(() => console.log('Event triggered'), 300);
window.addEventListener('resize', log);
```

---

25. What is `throttling` in JavaScript?

Throttling ensures a function is called at most once in a specified period, regardless of the number of events triggered.

Example:

javascript

```javascript
function throttle(func, limit) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      func.apply(this, args);
    }
  };
}

const log = throttle(() => console.log('Event triggered'), 1000);
window.addEventListener('scroll', log);
```

---

26. What is `Currying` in JavaScript?

Currying transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

Example:

javascript

```javascript
function curry(func) {
  return function curried(...args) {
    if (args.length >= func.length) {
      return func(...args);
    }
    return (...nextArgs) => curried(...args, ...nextArgs);
  };
}

const add = (a, b, c) => a + b + c;
const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3)); // 6
```

---

27. Explain `Object.freeze` and `Object.seal`.

`Object.freeze`: Prevents modifications to properties and prohibits adding or removing properties.
javascript

```
const obj = Object.freeze({ a: 1 });
obj.a = 2; // Ignored
console.log(obj.a); // 1
```

-

`Object.seal`: Allows modifying existing properties but prevents adding or removing them.
javascript

```
const obj = Object.seal({ a: 1 });
obj.a = 2; // Works
delete obj.a; // Ignored
console.log(obj.a); // 2
```

-

---

28. What is the difference between `setTimeout` and `setInterval`?

`setTimeout`: Executes a function after a specified delay.
javascript

```
setTimeout(() => console.log('Executed after 1s'), 1000);
```

-

`setInterval`: Repeatedly executes a function at specified intervals.
javascript

```
setInterval(() => console.log('Repeated every 1s'), 1000);
```

-

---

29. Explain `call`, `apply`, and `bind`.

These methods allow controlling the `this` context of a function:

`call`: Invokes a function with a specified `this` and arguments.
javascript

```
const obj = { x: 1 };
function printX() {
  console.log(this.x);
}
printX.call(obj); // 1
```

-

`apply`: Similar to `call`, but takes arguments as an array.
javascript

```
printX.apply(obj); // 1
```

-

`bind`: Returns a new function with a specified `this`.
javascript

```
const boundPrintX = printX.bind(obj);
boundPrintX(); // 1
```

-

---

30. What is `hoisting` in JavaScript?

Hoisting moves variable and function declarations to the top of their scope during the compile phase.

Example:
javascript

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

console.log(b); // ReferenceError (let/const are not initialized)
let b = 10;
```

31. What is the `this` keyword in JavaScript?

The `this` keyword refers to the context in which the function is called. It can change depending on how a function is invoked.

Global context (non-strict mode): `this` refers to the global object (e.g., `window` in browsers).
javascript

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

-

Method context: `this` refers to the object that the method belongs to.
javascript

```
const obj = {
  name: 'John',
  greet: function() {
    console.log(this.name);
  }
};
obj.greet(); // 'John'
```

-

Arrow functions: In arrow functions, `this` is lexically bound to the surrounding context, meaning it inherits `this` from the enclosing function.
javascript

```
const obj = {
  name: 'John',
  greet: () => {
    console.log(this.name); // undefined because `this` refers to the
global context
  }
};
obj.greet();
```

-

---

32. What is `Promises` in JavaScript?

A `Promise` is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

States of a Promise:

1. **Pending:** Initial state.
2. **Fulfilled:** Operation completed successfully.
3. **Rejected:** Operation failed.

Example:
javascript

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve('Operation succeeded!');
  } else {
    reject('Operation failed!');
  }
});

promise
  .then(result => console.log(result))  // "Operation succeeded!"
  .catch(error => console.log(error));  // "Operation failed!"
```

---

33. What is the difference between `==` and `===` in JavaScript?

`==` (Equality Operator): Compares values, performing type coercion if necessary.
javascript

```
console.log(1 == '1'); // true, because '1' is coerced to number
```

-

`===` (Strict Equality Operator): Compares both value and type without type coercion.
javascript

```
console.log(1 === '1'); // false, because they are different types
(number vs string)
```

-

---

34. What are `set` and `map` in JavaScript?

`Set`: A collection of unique values. Duplicate values are ignored.
javascript

```
let set = new Set();
set.add(1);
set.add(2);
set.add(1); // Duplicate, will be ignored
console.log(set); // Set { 1, 2 }
```

-

`Map`: A collection of key-value pairs where keys can be any data type.
javascript

```
let map = new Map();
map.set('a', 1);
map.set(2, 'b');
console.log(map); // Map { 'a' => 1, 2 => 'b' }
```

-

35. What are higher-order functions in JavaScript?

Higher-order functions are functions that take other functions as arguments or return a function as a result.

Example:

javascript

```
function greet(name) {
  return `Hello, ${name}`;
}

function runCallback(callback) {
  console.log(callback('John'));
}

runCallback(greet); // 'Hello, John'
```

36. What is a `closure` in JavaScript?

A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope.

Example:

javascript

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

const counter = outer();
counter(); // 1
counter(); // 2
```

37. What is a `Module` in JavaScript?

A module is a self-contained unit of code that can be reused and exported to other parts of an application. Modules help organize code, making it more maintainable.

Exporting a function or variable:

javascript

```
// In file module.js
export const greet = () => "Hello!";
```

-

Importing in another file:

javascript

```
// In main.js
import { greet } from './module.js';
console.log(greet()); // 'Hello!'
```

-

---

38. What is the `async`/`await` syntax in JavaScript?

The `async` and `await` keywords are used to simplify working with promises. An `async` function always returns a promise, and `await` makes JavaScript wait for a promise to resolve or reject.

Example:

javascript

```
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}

fetchData();
```

---

39. What is `Object Destructuring` in JavaScript?

Object destructuring allows extracting properties from an object and assigning them to variables in a concise way.

Example:

javascript

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
```

```
console.log(name); // 'John'
console.log(age);  // 30
```

---

40. What is `Array Destructuring` in JavaScript?

Array destructuring allows extracting elements from an array and assigning them to variables in a concise way.

Example:

javascript

```
const arr = [1, 2, 3];
const [a, b] = arr;

console.log(a); // 1
console.log(b); // 2
```

41. What is Event Delegation in JavaScript?

Event delegation is a technique in which a parent element listens for events on its child elements. Instead of adding an event listener to each child, you add it to the parent and use the event's `target` property to identify the element that triggered the event.

Example:

javascript

```
document.querySelector('#parent').addEventListener('click',
function(e) {
  if (e.target && e.target.matches('button')) {
    console.log('Button clicked');
  }
});
```

In this example, the event listener is attached to the parent element (`#parent`), and it listens for `click` events on any button inside it.

---

42. What is the `bind()` method in JavaScript?

The `bind()` method creates a new function that, when called, has its `this` value set to the provided value, and any given arguments are prepended to the arguments passed to the bound function.

Example:

javascript

```
const obj = {
  name: 'John',
  greet: function() {
    console.log('Hello, ' + this.name);
  }
};

const greetJohn = obj.greet.bind(obj);
greetJohn(); // 'Hello, John'
```

---

43. What are JavaScript data types?

JavaScript has primitive and non-primitive (reference) data types:

- **Primitive types:**
  - `String` – Represents textual data.
  - `Number` – Represents numerical values.
  - `Boolean` – Represents `true` or `false`.
  - `undefined` – Represents an uninitialized variable.
  - `null` – Represents an empty or non-existent value.
  - `Symbol` – Represents a unique identifier.
  - `BigInt` – Represents large integers beyond the safe range of `Number`.
- **Reference types:**
  - `Object` – Represents a collection of properties.
  - `Array` – A special type of object for storing ordered collections.
  - `Function` – A callable object.

---

44. What is the `fetch()` function in JavaScript?

The `fetch()` function is used to make network requests to retrieve resources from a server. It returns a promise that resolves to the response of the request.

Example:

javascript

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log('Error:', error));
```

---

45. What is an Immediately Invoked Function Expression (IIFE)?

An IIFE is a function that is defined and immediately invoked (executed) after its creation. It's commonly used to create a local scope to avoid polluting the global namespace.

Example:

javascript

```
(function() {
  console.log('I am an IIFE!');
})();
```

---

46. What is the `new` keyword in JavaScript?

The `new` keyword is used to create a new instance of an object from a constructor function or a class. It sets up the new object and binds `this` to it.

Example:

javascript

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

const person1 = new Person('Alice');
console.log(person1.name); // 'Alice'
```

---

47. What are `setTimeout()` and `setInterval()` in JavaScript?

setTimeout(): Executes a function after a specified delay in milliseconds.
javascript

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

-

setInterval(): Executes a function repeatedly at specified intervals.
javascript

```
setInterval(() => {
  console.log('This will run every 2 seconds');
}, 2000);
```

-

---

48. What is localStorage and sessionStorage in JavaScript?

localStorage: Stores data with no expiration time. The data persists even after the browser is closed.
javascript

```
localStorage.setItem('username', 'John');
console.log(localStorage.getItem('username')); // 'John'
```

-

sessionStorage: Stores data for the duration of the page session. The data is cleared when the page is closed.
javascript

```
sessionStorage.setItem('sessionKey', '12345');
console.log(sessionStorage.getItem('sessionKey')); // '12345'
```

-

---

49. What is the constructor function in JavaScript?

The `constructor` is a special function in a class or constructor function, responsible for creating and initializing objects created with `new`.

Example:

javascript

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

const person1 = new Person('Alice', 25);
console.log(person1.name); // 'Alice'
console.log(person1.age);  // 25
```

---

50. What is `window` in JavaScript?

The `window` object represents the global scope in browsers. It refers to the browser's window and is the root object of JavaScript in the browser context.

- Provides methods like `alert()`, `prompt()`, and `confirm()`.
- Holds properties like `document`, `location`, and `history`.

javascript

```
window.alert("Hello, World!");
console.log(window.document);
```