1) **What are the different data types in JavaScript?**

   Answer => String, Number, Boolean, Undefined, Null, Symbol, BigInt (Primitives) and Object (Reference type, includes Arrays, Functions, etc.).

2) **How is var different from let and const?**

   Answer => var is function-scoped and hoisted initialized with undefined. let and const are block-scoped and hoisted into a "temporal dead zone" (not accessible until declared). const also prevents re-assignment.

3) **What is hoisting in JavaScript?**

   Answer => JavaScript's mechanism where variable and function declarations are moved to the top of their scope during compilation. var is initialized; let/const are not accessible before declaration.

4) **What are primitive types vs reference types?**

   Answer => Primitive types (e.g., Number, String) store values directly and are copied by value. **Reference types** (e.g., Object, Array) store a reference (memory address) to the value and are copied by reference.

5) **What does the typeof operator return for different values?**

   Answer => Returns a string: "string", "number", "boolean", "undefined", "symbol", "bigint", "function", and "object" (for objects, arrays, and null).

6) **What is the difference between == and ===?**

   Answer => == performs type coercion before comparing values. === compares values and types strictly, without coercion.

7) **How do you convert a string to a number?**

   Answer => Number(), unary plus operator (+), parseInt(), or parseFloat().

8) **What is the use of parseInt() and parseFloat()?**

   Answer => parseInt() parses a string to an integer, stopping at the first non-numeric character (can specify radix). parseFloat() parses a string to a floating-point number.

9) **What will be the output of console.log(1 + '1')?**

   Answer => '11'. The number 1 is coerced to a string, resulting in string concatenation.

10) **What are falsy values in JavaScript?**

    Answer => false, 0, -0, 0n, "", null, undefined, and NaN. These evaluate to false in a boolean context.

11) **How do you check if a variable is undefined?**

    Answer => typeof myVariable === 'undefined' (recommended for declared/undeclared variables) or myVariable === undefined (for declared variables).

12) **What is the difference between null and undefined?**

    Answer => undefined means a variable has been declared but not assigned a value. null is an intentional assignment indicating the absence of any object value.

13) **How do you create a function in JavaScript?**

    Answer => Using function declarations (function name() {}), function expressions (const name = function() {};), or arrow functions (const name = () => {};).

14) **What is the difference between function declaration and expression?**

    Answer => Declarations are hoisted (callable before defined). **Expressions** are not hoisted (only the variable is hoisted, not the function definition).

15) **What is scope in JavaScript?**

Answer => Determines the accessibility of variables and functions. JavaScript has Global, Function, and Block (for let/const) scopes.

16) **What is block scope and function scope?**

Answer => Function scope means variables (using var) are accessible throughout the function. **Block scope** (for let/const) means variables are only accessible within the {} block where they're defined.

17) **What are arrow functions and how are they different?**

Answer => A concise syntax for function expressions. Key differences: no this binding (lexically inherits this), no arguments object, cannot be used as constructors.

18) **How does JavaScript handle automatic type conversion?**

Answer => Implicitly converts values from one data type to another during operations (e.g., 1 + '2' results in string concatenation; '10' - 5 converts string to number for subtraction).

19) **What is a callback function?**

Answer => A callback function is a function that is passed as an argument to another function, to be executed later. The "calling" function will "call back" or execute the provided function at a specific point in its own execution, typically after some operation (like an asynchronous task, an event, or an iteration) has completed. This pattern allows for flexible and asynchronous control flow, ensuring that certain code runs only after a preceding task is finished.

Example: Consider a function `fetchUserData` that simulates fetching data from a server. It might take some time (asynchronous operation). We want to `displayUser` data *only after* it's fetched.

JavaScript

```
function fetchUserData(userId, callback) {
console.log(`Fetching data for user ID: ${userId}...`);
// Simulate network request delay (asynchronous operation)
setTimeout(() => {
const userData = { id: userId, name: "Alice", email: "alice@example.com" };
console.log("Data fetched successfully!");
callback(userData); // This is where the callback function is executed
}, 2000); // 2-second delay
}
function displayUser(user) {
console.log(`Displaying user: Name - ${user.name}, Email - ${user.email}`);
}
// Call fetchUserData, passing displayUser as the callback
fetchUserData(123, displayUser);
console.log("Request initiated. Will display user data when available.");
```

21. **What is the difference between for, for...in, and for...of?**

Answer) **for**: A traditional loop for iterating a specific number of times, typically over array indices.

for...in: Iterates over *enumerable property names (keys)* of an object, including inherited ones. Not recommended for arrays.

for...of: Iterates over *iterable values* (e.g., elements of an array, characters of a string, items in a Map/Set).

22. **What are template literals?**

Answer) Template literals (backticks `) are ES6 string literals allowing embedded expressions (${expression}), multi-line strings, and simplified string interpolation without explicit concatenation.

23. How do you use default parameters **in a function?**

Answer) You assign default values directly in the function's parameter list. If an argument is omitted or undefined for that parameter, the default value is used. function greet(name = 'Guest') { console.log(Hello, ${name}); }

24. How do you clone an object?

Answer) **Shallow clone:** Spread syntax ({...obj}), Object.assign({}, obj), or JSON.parse(JSON.stringify(obj)) (for simple objects, but handles deep clone for them, fails on functions/Dates).

Deep clone: structuredClone() (browser/Node.js v17+), or custom recursive functions/libraries (e.g., Lodash's _.cloneDeep()).

25. What is destructuring in JavaScript?

Answer) A convenient way to extract values from arrays or properties from objects into distinct variables using syntax that mirrors array and object literals. const { name, age } = person; or const [first, second] = arr;

26. What is a spread operator (...)?

Answer) The spread operator expands an iterable (like an array or string) into individual elements or an object into key-value pairs. [...arr1, ...arr2] (merge arrays), {...obj1, ...obj2} (merge objects), Math.max(...numbers) (pass array elements as arguments).

27. How can you merge two arrays?

Answer) Spread syntax: const newArray = [...array1, ...array2]; (common and concise)

concat() method: const newArray = array1.concat(array2);

28. What is a pure function?

Answer) A function that always returns the same output for the same input and produces no side effects (e.g., doesn't modify external state, doesn't perform I/O).

29. What are higher-order functions?

Answer) Functions that either take one or more functions as arguments, or return a function as their result (or both). Examples: map, filter, setTimeout, forEach.

30. Explain Array.prototype.map(), filter(), and reduce().

Answer)

map(): Creates a *new array* by transforming each element of the original array according to a provided callback function.

filter(): Creates a *new array* containing only elements for which the provided callback function returns true.

reduce(): Executes a reducer callback function on each element of the array, resulting in a *single output value* (e.g., sum, average, a transformed object).

31. What is closure in JavaScript?

Answer) A closure is the combination of a function and the lexical environment within which that function was declared. It allows an inner function to retain access to its outer function's scope, even after the outer function has finished executing.

32. **What are IIFEs and why are they used?**

Answer) IIFE stands for Immediately Invoked Function Expression. It's a function defined and executed immediately after creation ((function() { /* ... */ })();). Used to create a private scope, avoiding polluting the global namespace, and for module patterns.

33. **Explain event bubbling and event capturing.**

Answer) These are two phases of event propagation in the DOM:

- **Capturing (Trickle Down):** The event starts from the window object and "trickles down" to the target element.
- **Bubbling (Bubble Up):** After reaching the target, the event "bubbles up" from the target element back to the window object. Events typically default to the bubbling phase.

34. **What are the different ways to create objects in JavaScript?**

Answer)

- Object literal: { name: "value" }
- Constructor function: new MyConstructor()
- Object.create(): Creates a new object, using an existing object as the prototype.
- Classes (ES6): new MyClass() (syntactic sugar over constructor functions/prototypes).

35. **What is the prototype chain?**

Answer) A mechanism in JavaScript for inheritance. When trying to access a property or method on an object, if it's not found directly on the object, JavaScript looks for it on the object's prototype, then on that prototype's prototype, and so on, until it reaches null (the end of the chain).

36. **How does prototypal inheritance work?**

Answer) Instead of classes, JavaScript uses prototypes. Objects inherit properties and methods from other objects (their prototypes). When an object is created, it gets a link to a prototype object, from which it can inherit behavior.

37. **What are getters and setters?**

Answer) Special methods in objects that allow you to define custom behavior when a property is accessed (getter) or assigned (setter). They act like regular properties but run a function under the hood. get propertyName() { ... }, set propertyName(value) { ... }

38. **What is the new keyword in JavaScript?**

Answer) The new keyword is used to create an instance of a user-defined object type or a built-in object type that has a constructor function. It:

1. Creates a new, empty object.
2. Sets the new object's internal [[Prototype]] to the constructor function's prototype property.
3. Executes the constructor function with this bound to the new object.
4. Returns the new object (unless the constructor explicitly returns another object).

39. **What are constructors?**

Answer) In JavaScript, a constructor is a regular function used with the new keyword to create and initialize new objects. By convention, constructor function names start with an uppercase letter. They define the initial state and properties of objects created from them.

40. **What is the difference between .call(), .apply(), and .bind()?**

    Answer) All three manipulate the this context of a function:

    - **.call(thisArg, arg1, arg2, ...)**: Invokes the function immediately with a specified this value and arguments passed individually.
    - **.apply(thisArg, [argsArray])**: Invokes the function immediately with a specified this value and arguments passed as an array.
    - **.bind(thisArg, arg1, arg2, ...)**: Returns a *new function* with this permanently bound to thisArg and optionally pre-set arguments. The new function is *not* executed immediately.

41. **What is a factory function?**

    Answer) A factory function is a function that returns a new object every time it's called. It's an alternative to using classes or constructors, especially useful when you want to create multiple similar objects without using the new keyword.

42. What is currying in JavaScript?

    Answer)Currying is a technique where a function is transformed into a sequence of functions, each taking a single argument. It allows partial function application and improves code modularity and reusability.

43. **What are modules and how do you import/export in ES6?**

    Answer) Modules allow code to be organized and reused across files. In ES6, export is used to expose variables or functions, and import is used to bring them into another file. For example:

    ```
    JS
    // module.js
    export const x = 10;
    // main.js
    import { x } from './module.js';
    ```

44. **What is a symbol in JavaScript?**

    Answer) A symbol is a primitive data type introduced in ES6. It represents a unique and immutable identifier, often used as object property keys to avoid name collisions.

45. **What is Object.freeze() and Object.seal()?**

    Answer) Object.freeze() makes an object completely immutable — you can't add, remove, or modify properties.

    Object.seal() allows modifying existing properties but prevents adding or removing them.

46. **How do you check if an object has a property?**

    Answer)You can use the hasOwnProperty() method or the in operator:

    ```
    JS
    obj.hasOwnProperty('prop') // true
    'prop' in obj
    ```

47. **What is the in operator?**
    Answer) The in operator checks whether a property exists in an object, including inherited properties from the prototype chain.
    Example: 'length' in [] // true
48. **What is JSON and how do you parse and stringify it?**
    Answer) JSON (JavaScript Object Notation) is a lightweight data format.
    Use JSON.parse() to convert a JSON string into a JavaScript object.
    Use JSON.stringify() to convert a JavaScript object into a JSON string.
49. **What is the difference between Array.isArray() and instanceof Array?**
    Answer) Array.isArray() is reliable across different frames or contexts.
    instanceof Array can fail if the array comes from a different window or iframe.
    So, Array.isArray() is the recommended way to check for arrays
50. **What is event delegation?**
    Answer) Event delegation is a technique where a parent element handles events for its child elements using event bubbling. It's efficient for dynamically generated content and reduces the number of event listeners.
51. How do you debounce or throttle a function?
    Answer) Debouncing delays function execution until after a specified time has passed since the last call. Throttling limits function execution to once per specified time interval.

```
// Debouncing
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}
// Throttling
function throttle(func, delay) {
  let lastCall = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      func.apply(this, args);
    }
  };
}
```

52. What are Promises?
    Answer) Promises represent the eventual completion or failure of an asynchronous operation. They provide a cleaner alternative to callbacks for handling async code and avoid callback hell.
    Javascript

```
const promise = new Promise((resolve, reject) => {
// async operation
if (success) resolve(data);
else reject(error);
});
promise.then(data => console.log(data))
.catch(error => console.error(error));
```

53. What are the states of a Promise?

    Answer) A Promise has three states:

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

    Once a promise is settled (fulfilled or rejected), it cannot change states.

54. **How does the event loop work?**

    Answer) The event loop manages JavaScript's single-threaded execution by coordinating the call stack, callback queue, and microtask queue.

    Process:

    - ➢ Execute code in call stack
    - ➢ When call stack is empty, process all microtasks
    - ➢ Then process one microtask
    - ➢ Repeat the cycle This ensures non-blocking execution of asynchronous operations.

55. **What is the difference between microtasks and macrotasks?**

    Answer) Microtasks (higher priority):

    - ➢ Promise callbacks (.then, .catch, .finally)
    - ➢ queueMicrotask()
    - ➢ MutationObserver

    Macrotasks (lower priority):

    - ➢ setTimeout/setInterval
    - ➢ DOM events
    - ➢ HTTP requests

56. **What is setTimeout() and setInterval()?**

    Answer) setTimeout(): Executes a function once after a specified delay

    setInterval(): Executes a function repeatedly at specified intervals

    javascript

    ```
    setTimeout(callback, delay, ...args);
    setInterval(callback, interval, ...args);
    ```

    Both return unique IDs that can be used to cancel the timer.

57. **What is memory leak in JavaScript?**

    Answer) Memory leaks occur when objects are no longer needed but aren't garbage collected due to lingering references, causing memory usage to grow continuously.**Common causes:**

    - ➢ Forgotten event listeners

- ➢ Closures holding references to large objects
- ➢ Global variables not cleaned up
- ➢ Detached DOM nodes
- ➢ Timers not cleared

58. **What are WeakMap and WeakSet?**

A: WeakMap: Stores key-value pairs where keys must be objects and are weakly referenced

- ➢ Keys can be garbage collected when no other references exist
- ➢ Not enumerable
- ➢ Use case: Private object data

WeakSet: Collection of objects that are weakly referenced

- ➢ Objects can be garbage collected
- ➢ Not enumerable
- ➢ Use case: Tracking object instances

59. **What is async/await in JavaScript?**

Answer) async/await is syntactic sugar built on top of Promises that allows us to write asynchronous code in a synchronous-looking and more readable way.

Technical Explanation:

- ➢ async keyword is added before a function to make it return a Promise.
- ➢ await is used inside an async function to pause execution until a Promise is resolved or rejected.
- ➢ This helps avoid callback hell and then-chaining in Promises.

JS

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

What really happens:

- ➢ When the engine sees await, it pauses the function's execution.
- ➢ It resumes only after the awaited Promise settles (either success or error).
- ➢ The rest of the function executes only when the awaited line is done.

60. **What are rest parameters?**

Answer) Rest parameters allow a function to accept an indefinite number of arguments as an array. They're denoted by three dots (...) followed by a parameter name.

Key points:

- Must be the last parameter
- Creates a real array (not array-like object)
- Different from arguments object

61. **What is optional chaining (?.)?**

Answer) Optional chaining allows safe access to nested object properties without throwing errors if intermediate properties are null or undefined.

Benefits:

- ➢ Prevents runtime errors

- ➢ Cleaner code than manual null checks
- ➢ Works with properties, methods, and array indices

62. **What is nullish coalescing (??)?**

    Answer) Nullish coalescing operator returns the right-hand operand when the left-hand operand is null or undefined (but not other falsy values like 0, '', false).

    const name2 = " " ?? "Default"; *// "" (empty string is preserved)*

    const count2 = 0 ?? 10; *// 0 (zero is preserved)*

    Difference from ||:

    - ➢ || triggers on all falsy values (0, '', false, null, undefined, NaN)
    - ➢ ?? triggers only on null and undefined

1. What is the use of Object.entries() and Object.fromEntries()?

    Object.entries(obj) converts an object into an array of [key, value] pairs.

    Object.fromEntries(array) does the reverse — it converts an array of key-value pairs back into an object.

    Example:

    Js

    const user = { name: 'Prashant', age: 25 };

    const entries = Object.entries(user); // [['name', 'Prashant'], ['age', 25]]

    const obj = Object.fromEntries(entries); // { name: 'Prashant', age: 25 }

2. How do you deep clone an object?

    Deep cloning creates a complete copy of an object and all of its nested objects.

    Ways to deep clone:

    Using structuredClone() (best in modern browsers)

    Using JSON.parse(JSON.stringify(obj)) (works but loses functions, undefined, etc.)

    Using libraries like Lodash: _.cloneDeep(obj)

    Manually using recursion

3. Explain the difference between shallow copy and deep copy.

    Shallow copy copies only the top-level properties; nested objects still reference the original.

    Deep copy duplicates everything, including nested objects, making them independent.

    Example:

    Js

    const original = { name: 'John', details: { age: 30 } };

    const shallow = { ...original };

    shallow.details.age = 40;

    // original.details.age is also 40 — that's shallow copy

4. What is tail call optimization?

    Tail Call Optimization (TCO) is a JavaScript engine feature where recursive function calls in tail position (i.e., the last action in a function) don't add a new stack frame.

    This helps prevent stack overflow errors and improves performance in recursion-heavy code.

    Note: TCO is part of the ES6 spec, but it's not widely implemented in major JavaScript engines like V8 (used in Chrome/Node.js).

5. What is a generator function?

A generator function is a special function that can pause and resume its execution using the yield keyword. It's defined with function*.
Used for lazy iteration, state machines, async control flows, etc.Example:
Js
```
function* count() {
yield 1;
 yield 2;
  yield 3;
}
const gen = count();
gen.next(); // { value: 1, done: false }
```
gen.next(); // { value: 2, done: false }

1. What is eval() and why is it dangerous?
   eval() executes a string of JavaScript code at runtime.
   - It's dangerous because:
   - It can run malicious code (security risk)
   - It's slow (forces the JS engine to re-parse and re-optimize code)
   - It can access and manipulate local variables in unsafe ways
   - Avoid it and use safer alternatives like JSON.parse, Function, or logic-based parsing.
2. What are service workers?
   Service workers are JavaScript scripts that run in the background and allow features like:
   - Offline support
   - Sync
   - Notifications
   - Intercepting and caching network requests
   - They act as a proxy between the browser and the network.
   - Example use-case: In a PWA (Progressive Web App), service workers cache assets and let the app work offline.
3. How do you access an element by ID, class, or tag name?
   - By ID: document.getElementById('id')
   - By class: document.getElementsByClassName('className') (returns HTMLCollection)
   - By tag name: document.getElementsByTagName('tagName') (returns HTMLCollection)
4. What is querySelector vs getElementById?
   - querySelector() allows CSS-style selectors (e.g., #id, .class, div > p) and returns the first match.
   - getElementById() is faster but only works for IDs.
5. How do you change the content of an element?
   Use:
   - element.textContent = "new text";
   - element.innerHTML = "<b>new HTML</b>";
   - element.innerText = "visible text"; (includes CSS visibility rules)
6. How do you create and append a new DOM element?
   Js

```
const div = document.createElement("div");
div.textContent = "Hello!";
document.body.appendChild(div);
```

7. What is the difference between innerHTML, textContent, and innerText?
   - innerHTML: Sets/gets HTML content.
   - textContent: Gets/sets raw text (ignores HTML).
   - innerText: Similar to textContent, but respects CSS styles (like display: none).
8. How do you remove an element from the DOM?

   Js
   ```
   element.remove(); // Modern// or
   element.parentNode.removeChild(element); // Legacy
   ```
9. What are attributes and how can you modify them?
   - Attributes are HTML properties like href, src, id, etc.
   - Get: element.getAttribute("href")
   - Set: element.setAttribute("href", "https://example.com")
   - Remove: element.removeAttribute("href")
10. How do you add or remove a class from an element?

    Js
    ```
    element.classList.add("active");
    element.classList.remove("active");
    element.classList.toggle("active");
    element.classList.contains("active"); // check
    ```
11. How do you handle click events in JavaScript?

    Js
    ```
    element.addEventListener("click", function (event) {  // handle click});
    ```
12. What are the different types of DOM events?
    - Mouse Events: click, dblclick, mouseenter, mouseleave
    - Keyboard Events: keydown, keyup, keypress
    - Form Events: submit, change, input, focus, blur
    - Window Events: resize, scroll, load, DOMContentLoaded
13. How do you stop event propagation?

    Js
    ```
    event.stopPropagation(); // stops bubbling
    event.stopImmediatePropagation(); // stops all further listeners
    ```
14. What is event.target vs event.currentTarget?

    event.target: The actual element that triggered the event.

    event.currentTarget: The element that the event listener is attached to.
15. How do you detect keypress or keydown events?

    Js
    ```
    document.addEventListener("keydown", (e) => {  console.log(e.key); // logs the pressed key});
    ```
16. How do you handle form submissions using JavaScript?

    Js

form.addEventListener("submit", function (e) { e.preventDefault(); // stop actual submit // Handle form data});

17. What are the new features introduced in ES6?

    Some major ES6 features include:

    - let and const
    - Arrow functions (=>)
    - Template literals
    - Default parameters
    - Destructuring
    - Spread & rest operators
    - Classes and inheritance
    - Modules (import/export)
    - Promises
    - Symbols
    - Enhanced object literals
    - Iterators and generators

18. What are arrow functions and lexical this?

    - Arrow functions are a concise syntax for writing functions:

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

    - They don't have their own this — instead, they lexically bind this from their surrounding scope. This avoids issues in callbacks where this changes.

19. How do let and const work with block scope?

    - let and const are block-scoped (limited to { } blocks), unlike var which is function-scoped.
    - const prevents reassignment (but not mutation of objects).

    ```
    Js
    if (true) {
     let x = 10;
    const y = 20;
    }
    // x and y are not accessible here
    ```

20. What is destructuring assignment?

    - Destructuring lets you extract values from arrays or objects easily:

    ```
    Js
    const [a, b] = [1, 2];
    const { name, age } = { name: "John", age: 25 };
    ```

21. What is the spread syntax and how is it different from rest?

    - Spread (...) expands elements:

    ```
    Js
    const arr = [1, 2];
    const newArr = [...arr, 3]; // [1, 2, 3]
    ```

    Rest (...) collects multiple elements into one:

```
Js
function sum(...nums) {
  return nums.reduce((a, b) => a + b);
}
```

22. What are modules in ES6?
    - ➢ Modules allow code separation and reuse using import and export:

```
Js
// math.js
export const add = (a, b) => a + b;
// main.js
import { add } from "./math.js";
```

23. What are classes in JavaScript?
    - ➢ Classes are syntactic sugar over constructor functions:

```
Js
class Person {
constructor(name) {
this.name = name;
}
 greet() {
 return `Hello, I'm ${this.name}`;
 }
}
```

24. What is inheritance using extends?
    - ➢ The extends keyword allows a class to inherit from another:

```
Js
class Animal {
  speak() { return "sound"; }
}
class Dog extends Animal {
 speak() { return "bark"; }
}
```

25. What is a static method?
    - ➢ A static method is called on the class itself, not instances:

```
Js
class MathUtil {
static square(x) {
return x * x;
}
}
MathUtil.square(5); // 25
```

26. What is the use of super()?
    - ▪ super() calls the parent class constructor.

- Used inside a subclass constructor to access this.

```
Js
class Animal {
 constructor(name) {
 this.name = name;
}
}
class Dog extends Animal {
  constructor(name, breed) {
 super(name); // Call parent constructor
this.breed = breed;
  }
}
```

27. **What is synchronous vs asynchronous code?**
    - **Synchronous:** Executes line by line; each operation blocks the next.
    - **Asynchronous:** Allows tasks to run in the background (e.g., network requests), freeing up the main thread to continue executing other code.
    - *Real-world example:* Waiting in a queue vs placing an online order and continuing your work.
28. **What is a callback hell?**
    - It's a situation where multiple nested callbacks make code **difficult to read and maintain**, often forming a "pyramid of doom":

```
Js
getUser(id, (user) => {
getPosts(user.id, (posts) => {
getComments(posts[0], (comments) => {
// hard to manage});
 });});
```

29. **What is .then() and .catch() in Promises?**
    - .then(): Used to handle resolved promises.
    - .catch(): Used to catch and handle errors.

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

30. **How do you handle errors in async/await?**
    - Use try...catch blocks:

```
Js
async function getData() {
  try {
 const res = await fetch(url);
const data = await res.json();
 } catch (error) {console.error("Error:", error);
}
}
```

31. **What is fetch() API and how is it used?**
    - fetch() is used to make network requests. It returns a **Promise**.

```
Js
fetch('https://api.example.com')
.then(res => res.json())
 .then(data => console.log(data));
```

32. How does a for...in loop differ from a for...of loop?
    - for...in: Iterates over keys (property names) in an object or array.
    - for...of: Iterates over values of an iterable (like arrays, strings).

```
    Js
let arr = ['a', 'b'];
for (let i in arr) console.log(i); // 0, 1
for (let i of arr) console.log(i); // 'a', 'b'
```

33. What is the purpose of the finally block in a try-catch statement?
    - The finally block always executes, regardless of an error. Useful for cleanup tasks.

```
Js
try {
} catch (e) {
} finally {// always runs}
```

34. What are the common methods to add elements to an array?
    - push() – adds to the end
    - unshift() – adds to the beginning
    - splice() – adds at a specific index
35. What are the common methods to remove elements from an array?
    - pop() – removes from the end
    - shift() – removes from the beginning
    - splice() – removes by index
36. Difference between slice() and splice()?
    - slice(start, end): Returns a shallow copy, doesn’t modify original.
    - splice(start, count): Modifies original array by adding/removing.
37. Difference between Object.create() and constructor pattern?
    - Object.create(proto) creates a new object with the specified prototype.
    - Constructor pattern uses functions with new.

```
Js
function Person(name) {
 this.name = name;
    }
const p1 = new Person("John");
const p2 = Object.create({ name: "Jane" });
```

1. What is Node.js and how does it work under the hood?
   Answer:
   Node.js is a runtime environment that allows you to run JavaScript code outside the browser, built on Chrome's V8 engine. Under the hood, it uses a single-threaded, non-blocking event loop architecture, powered by libuv, to handle asynchronous operations efficiently.
2. What is the event loop in Node.js?
   Answer:
   The event loop is the core mechanism that handles asynchronous operations in Node.js. It continuously checks the call stack and task queue (like timers, I/O, promises) to execute callbacks when the call stack is clear, allowing Node.js to perform non-blocking I/O operations.
3. How is Node.js single-threaded but still handles concurrency?
   Answer:
   Node.js uses a single-threaded event loop for JavaScript execution but delegates I/O tasks (like file system, network requests) to background threads using libuv. Once completed, callbacks are queued and processed by the event loop, enabling concurrent operations without multi-threading in JS.
4. What is the difference between process.nextTick(), setImmediate(), and setTimeout()?
   Answer:
   process.nextTick(): Executes after the current operation, before the event loop continues (microtask).
   setImmediate(): Executes on the next iteration of the event loop (macrotask).
   setTimeout(fn, 0): Also executes on the next tick but after setImmediate, depending on the context.
5. Explain the concept of streams in Node.js.
   Answer:
   Streams are a way to handle large amounts of data efficiently by processing it in chunks rather than all at once. Node.js provides four types: Readable, Writable, Duplex, and Transform. They're ideal for handling files, network data, etc., in a memory-efficient manner.
6. What are buffers in Node.js?
   Answer:
   Buffers are used to handle binary data streams, especially when dealing with files or network packets. They're instances of the Buffer class and allow reading and writing of raw memory outside the V8 heap.
7. What is the difference between require and import?
   Answer:
   require: CommonJS module system, used in Node.js by default. It's synchronous.
   import: ES6 module syntax, asynchronous and must be used with type="module" in Node.js or with bundlers like Webpack in the browser.
8. How does module caching work in Node.js?
   Answer:
   When a module is first required, Node.js loads and caches it. Subsequent require() calls return the cached version, which improves performance. Changes to the module after initial load won't reflect unless you manually clear the cache.
9. What are the global objects in Node.js?
   Answer:
   Some important global objects include:
   global: Similar to window in browsers.
   process: Provides info and control over the current Node.js process.
   __dirname and __filename: Give directory and file path.
   Buffer, console, setTimeout, setInterval, etc.
10. How do you handle uncaught exceptions and unhandled promise rejections?

Answer:
Use:
Js
```
process.on('uncaughtException', (err) => {
 console.error('Uncaught Exception:', err);
});
process.on('unhandledRejection', (reason, promise) => {
 console.error('Unhandled Rejection:', reason);
});
```
However, it's better to avoid relying on these by writing safer, error-handling code.

11. What is clustering in Node.js?

    Answer:
    Clustering allows Node.js to utilize multiple CPU cores by running multiple instances of the app (workers). The cluster module enables this, improving scalability for high-traffic applications.

12. How do you manage environment variables in Node.js?

    Answer:
    You can use the process.env object to access environment variables. For example:
    Js
    ```
    const port = process.env.PORT || 3000;
    ```
    For easier management, use the dotenv package to load variables from a .env file:
    Js
    ```
    require('dotenv').config();
    ```

13. What is Express.js and why is it used?

    Answer:
    Express.js is a fast, minimal, and flexible Node.js web application framework that simplifies building APIs and web servers. It provides robust features like routing, middleware support, and HTTP utilities, making backend development efficient.

14. How do you create middleware in Express?

    Answer:
    Middleware is a function with (req, res, next) signature. It can be created like this:
    Js
    ```
    app.use((req, res, next) => {
    console.log('Request received');
    next(); // Pass control to the next middleware
    });
    ```

15. What is the difference between middleware and routes?

    Answer:
    - Middleware: Functions that execute before the route handler. They can modify request/response or end the cycle.
    - Routes: Define specific endpoints and handle business logic, e.g., GET /users.

16. How does the request-response lifecycle work in Express?

    Answer:
    - Request hits the server.
    - Global/app-level middleware runs.
    - Route-specific middleware executes (if any).
    - Route handler processes the request.
    - Response is sent back to the client.
    - If no match, 404 or error middleware runs.

17. How do you handle errors in Express?

    Answer:

    Define error-handling middleware with 4 parameters:

    Js
    ```
    app.use((err, req, res, next) => {
    console.error(err.stack);
     res.status(500).send('Something broke!');
    });
    ```
    Use next(err) in routes/middleware to trigger it.

18. What is next() in Express and when do you use it?

    Answer:

    next() passes control to the next middleware in the stack. Use it:

    In custom middleware to move on.

    With an argument (next(err)) to pass errors to error-handling middleware.

19. How do you implement route-level and application-level middleware?

    Answer:

    App-level (runs for every request):

    Js
    ```
    app.use(middlewareFn);
    ```
    Route-level:

    Js
    ```
    app.get('/user', middlewareFn, (req, res) => { res.send('User'); });
    ```

20. What are the HTTP status codes and their meanings?

    Answer:

    - 200 OK – Success
    - 201 Created – Resource created
    - 400 Bad Request – Invalid input
    - 401 Unauthorized – No valid auth
    - 404 Not Found – Route not found
    - 500 Internal Server Error – Server failure

1. How do you handle CORS in Express?

    Answer:

    Use the cors package:

    Js
    ```
    const cors = require('cors');
    app.use(cors()); // Allow all
    ```
    For custom config:

    Js
    ```
    app.use(cors({ origin: 'https://yourdomain.com' }));
    ```

2. How do you serve static files in Express?

    Answer:

    Use Express's built-in express.static():

    Js
    ```
    app.use(express.static('public'));
    ```
    Now files in the public/ folder are accessible directly via browser.

3. How do you validate request data in Express apps?

    Answer:

    Use libraries like express-validator or Joi.

With express-validator:
Js
```
const { body, validationResult } = require('express-validator');
app.post('/user',
 body('email').isEmail(),
(req, res) => {
 const errors = validationResult(req);
if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });
res.send('Valid input!');
 }
);
```
4. What is MongoDB and how is it different from relational databases?
   Answer:
   MongoDB is a NoSQL, document-based database that stores data in JSON-like BSON format.
   Key differences from relational databases:
   - No tables, rows → uses collections and documents
   - Schema-less: Flexible structure
   - Horizontal scaling is easier
   - No JOINs by default; data is often embedded or linked
5. What is a document and a collection in MongoDB?
   Answer:
   A document is a single data record in BSON format (like a row in SQL).
   Json
   ```
   { "name": "John", "age": 25 }
   ```
   A collection is a group of documents (like a table in SQL).
6. How does indexing work in MongoDB?
   Answer:
   Indexes improve read performance by allowing MongoDB to locate data faster.
   Types:
   - Single field index
   - Compound index
   - Text index
   - Geospatial index

   Create an index:
   Js
   ```
   db.users.createIndex({ name: 1 });
   ```
7. What is aggregation in MongoDB? When do you use it?
   Answer:
   Aggregation is used to transform, filter, and compute over documents using a pipeline of stages like $match, $group, $project.
   Used for:
   - Reports
   - Analytics
   - Complex queries

   Example:
   Js
   ```
   db.orders.aggregate([
    { $match: { status: "delivered" } },
   ```

```
{ $group: { _id: "$customerId", total: { $sum: "$amount" } } }
]);
```

8. How do you define schemas and models using Mongoose?

   Answer:

   Js

   ```
   const mongoose = require("mongoose");
   const userSchema = new mongoose.Schema({
   name: String,
   age: Number
   });
   const User = mongoose.model("User", userSchema);
   ```

   Schema defines structure.

   Model is the constructor for querying and manipulating.

9. What is the difference between populate() and aggregate()?

   Answer:

   - populate() is used for referencing documents and auto-fetching related data (like SQL joins).

   Js

   ```
   Post.find().populate('author');
   ```

   - aggregate() is used for complex data transformations and groupings.

10. How do you handle relationships in MongoDB (1:1, 1:N, N:N)?

    Answer:

    - 1:1: Embed or reference.
    - 1:N: Embed array of sub-documents or use references.
    - N:N: Use reference IDs in both collections.

    Example:

    Js

    ```
    // 1:N reference
    const postSchema = new Schema({ user: { type: Schema.Types.ObjectId, ref: 'User' } });
    ```

11. What is the use of lean() in Mongoose queries?

    Answer:

    .lean() returns plain JavaScript objects instead of full Mongoose documents.

    Benefits:

    - Faster query
    - Less memory usage
    - Use when you don't need Mongoose document methods

    Js

    ```
    User.find().lean();
    ```

12. How do you handle transactions in MongoDB?

    Answer:

    MongoDB supports ACID transactions for multi-document operations using sessions.

    Example with Mongoose:

    Js

    ```
    const session = await mongoose.startSession();
    session.startTransaction();
    try {
      await User.create([{ name: "John" }], { session });
    await Order.create([{ userId: someId }], { session });
     await session.commitTransaction();
    ```

```
} catch (err) {
await session.abortTransaction();
}
session.endSession();
```

13. What are the different update operators in MongoDB?

    Answer:
    - $set: Update fields
    - $unset: Remove fields
    - $inc: Increment values
    - $push: Add to array
    - $pull: Remove from array
    - $addToSet: Add unique value to array

    Example:

    Js
    ```
    db.users.updateOne({ _id: id }, { $set: { name: "Jane" }, $inc: { age: 1 } });
    ```

14. How do you implement authentication in Node.js?

    Answer:

    Authentication can be implemented using:
    - Session-based auth (with express-session)
    - Token-based auth (with JWT)
    - OAuth (e.g., with Google, GitHub using Passport.js)
    - Common steps for JWT:
    - User logs in → server verifies credentials.
    - Server signs a JWT and sends it back.
    - Client stores it (usually in localStorage or cookie).
    - Client includes token in Authorization header in future requests.
    - Server verifies the token on each request.

15. What is the difference between session-based and token-based authentication?

Feature	Session-based	Token-based (JWT)
Storage	Server-side (in memory/Redis/DB)	Client-side (localStorage/cookie)
Scalability	Harder (requires sticky sessions or shared session store)	Easier (stateless)
Mechanism	Uses Set-Cookie and server memory	Uses Authorization: Bearer token
CSRF vulnerability	Yes (needs CSRF protection)	Less prone

16. How do you implement JWT (JSON Web Token) authentication?

    Answer:

    Install dependencies:

    ```
    npm install jsonwebtoken bcryptjs
    ```

    Login example:

```
Js
const jwt = require("jsonwebtoken");
app.post("/login", async (req, res) => {
const user = await User.findOne({ email: req.body.email });
const isMatch = await bcrypt.compare(req.body.password, user.password);
if (!isMatch) return res.status(401).send("Invalid credentials");
const token = jwt.sign({ userId: user._id }, "SECRET_KEY", { expiresIn: "1h" });
res.json({ token });
});
```

Verify middleware:

```
Js
const auth = (req, res, next) => {
 const token = req.headers.authorization?.split(" ")[1];
try {
  const payload = jwt.verify(token, "SECRET_KEY");
req.user = payload;
 next();
} catch (e) {
res.status(401).send("Unauthorized");
 }
};
```

17. How do you hash passwords and why is it important?

    Answer:

    Hashing protects passwords from being stored as plain text, so even if the database is leaked, passwords remain secure.

    - ➢ Use bcrypt:

    ```
    Js
    const bcrypt = require("bcryptjs");
    const hashed = await bcrypt.hash("password123", 10);
    ```

    During login:

    ```
    Js
    const isMatch = await bcrypt.compare(plainPassword, user.password);
    ```

    - ➢ Bcrypt automatically salts and hashes the password, making it secure.

18. How do you scale a Node.js app?

    Answer:

    - Use clustering to run on multiple CPU cores.
    - Deploy multiple instances behind a load balancer.
    - Use Redis or shared DB for session storage and caching.
    - Use PM2 to manage multiple Node.js processes.
    - pm2 start app.js -i max
    - You can also deploy horizontally on cloud platforms like AWS, Heroku, GCP using Docker and Kubernetes.

19. What is load balancing and how does it apply to Node.js?

    Answer:

    Load balancing distributes incoming traffic across multiple servers or processes to:

    - Improve performance
    - Increase availability
    - Ensure fault tolerance
    - Node.js apps can be load-balanced using:
    - Nginx, HAProxy (reverse proxies)
    - Cloud Load Balancers
    - Cluster module in Node.js (for multi-core use)

20. How do you optimize MongoDB queries?

    Answer:

    - Use Indexes on frequently queried fields.
    - Use .lean() in Mongoose for faster reads.
    - Avoid $where and large $in queries.
    - Project only required fields using .select().
    - Use explain() to analyze query performance.
    - Avoid unbounded queries (limit() and pagination help).
    - Avoid large $lookup in aggregation (use data modeling instead).

21. What are common vulnerabilities in Node.js apps and how do you prevent them?

Vulnerability	Solution
SQL/NoSQL Injection	Use parameterized queries or Mongoose schemas
XSS (Cross-site Scripting)	Sanitize user input; use libraries like DOMPurify
CSRF	Use CSRF tokens (csurf middleware) if using cookies
Insecure HTTP	Use HTTPS and helmet middleware
Directory traversal	Use path.join() and avoid using user input in file paths
Rate Limiting/Brute Force	Use express-rate-limit, captcha, or user lockouts
JWT attacks (Replay, Forgery)	Use short expiry + httpOnly cookies + rotate secrets