JavaScript Interview Answers

Basics

Q1: What is JavaScript?

A: JavaScript is a high-level, interpreted programming language primarily used to make web pages interactive. It runs in browsers and on servers (Node.js).

Q2: Key features of JavaScript:

- **Dynamic typing**: variables can hold any type
- **First-class functions**: functions are objects
- **Event-driven**: responds to user events
- **Prototype-based**: supports prototypal inheritance
- **Lightweight & versatile**: works in client & server environments

Q3: Differences between JavaScript and Java:

Feature	JavaScript	Java
Type	Dynamically typed	Statically typed
Execution	Browser/Node.js	JVM
Inheritance	Prototype-based	Class-based
Syntax	Flexible	Strict

Q4: How to include JavaScript in HTML?

```
<!-- Inline -->
<script>
  console.log('Hello World');
</script>

<!-- External -->
<script src="script.js"></script>
```

Q5: Data types in JavaScript

- **Primitive types:** string, number, boolean, null, undefined, symbol, bigint
- **Non-primitive types:** object (arrays, functions, objects)

Q6: Difference between = and ==

- = → Assignment operator
- == → Equality (type coercion)

Q7: Difference between null and undefined

- null → intentional absence of value
- undefined → variable declared but not initialized

Q8: typeof operator
Returns the data type of a variable:

```
typeof 42; // "number"

typeof 'hi'; // "string"
```

Q9: Variables in JavaScript
Declared using var, let, or const:

```
var a = 1;   // function scoped

let b = 2;   // block scoped

const c = 3; // block scoped, cannot reassign
```

Q10: Hoisting

- JS moves variable & function declarations to the top during compilation.
- var → hoisted with undefined
- let & const → hoisted but in **temporal dead zone**

Q11: Scope

- **Global scope** → accessible anywhere
- **Local scope** → accessible inside a function/block

Q12: Arrow functions

- Shorter syntax, lexical this binding:

```
const sum = (a, b) => a + b;
```

Q13: Closures

- Functions that remember the scope in which they were created:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}
const counter = outer();
counter(); // 1
```

Q14: IIFE

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

Q15: this keyword

- Refers to the object context where function is called.
- In arrow functions, this is lexically bound.

Q16: JSON

- **Parse:** string → object
- **Stringify:** object → string

```
JSON.parse('{"name":"Chetan"}');
JSON.stringify({name: 'Chetan'});
```

Q17: Promises
Used for async operations:

```
let p = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Done'), 1000);
});
p.then(console.log);
```

Q18: call, apply, bind

```
function greet(greeting) { console.log(greeting + ' ' + this.name); }

const person = { name: 'Chetan' };

greet.call(person, 'Hi');  // Hi Chetan

greet.apply(person, ['Hello']); // Hello Chetan

const bound = greet.bind(person);

bound('Hey'); // Hey Chetan
```

Basics & Variables (continued)

Q19: Difference between let, const, and var

Feature	**var**	**let**	**const**
Scope	Function	Block	Block
Re-declaration	Allowed	Not allowed	Not allowed
Re-assignment	Allowed	Allowed	Not allowed
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)

Q20: Hoisting differences for functions

- **Function declaration** → fully hoisted

```
foo(); // Works

function foo() { console.log('Hello'); }
```

- **Function expression** → hoisted as variable undefined

```
bar(); // Error

const bar = function() { console.log('Hi'); };
```

Q21: Scope Chain

- JS looks for variables **lexically** in local → outer → global scope.

Q22: Closure use cases

- Encapsulation of private data
- Maintaining state in asynchronous code
- Function factories

Q23: IIFE purpose

- Avoid polluting global scope
- Execute code immediately

Q24: this keyword explained

- **Global context:** this → window (browser)
- **Function context:** depends on call site
- **Object method:** refers to the object
- **Arrow function:** inherits this from enclosing scope

Q25: JSON parsing & stringifying

```
const obj = JSON.parse('{"name":"Chetan"}');

const str = JSON.stringify({ age: 25 });
```

Q26: Promises

- States: pending → fulfilled / rejected
- Methods: .then(), .catch(), .finally()

Q27: call, apply, bind difference

- **call** → function invoked immediately with arguments separated by commas
- **apply** → arguments as array
- **bind** → returns a new function

Q28: exec() vs test() (Regex)

```
const regex = /a/;

regex.test('abc'); // true (checks match)

regex.exec('abc'); // ['a'] (returns matched value)
```

Q29: Currying in JS

- Converting a function with multiple arguments into a sequence of functions each taking one argument.

```
const sum = a => b => a + b;

sum(2)(3); // 5
```

Q30: Rest parameter & spread operator

```
function sum(...nums) { return nums.reduce((a,b)=>a+b,0); } // rest

const arr = [1,2,3];

const arr2 = [...arr,4]; // spread
```

Q31: Implicit type coercion

- JS automatically converts types in operations:

```
'5' - 2; // 3

'5' + 2; // "52"
```

Q32: JS is dynamically typed ✅

Q33: NaN property

- Represents “Not a Number”

```
typeof NaN; // "number"
```

Q34: Pass by value vs reference

- Primitives → pass by value
- Objects/arrays → pass by reference

Q35: Strict mode

- Enables safer JS, disallows undeclared variables

```
'use strict';
```

Q36: Advantages of external JS

- Cacheable
- Cleaner HTML
- Reusable across pages

Q37: Constructor functions

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

const p = new Person('Chetan');
```

Q38: Recursion

- Function calling itself:

```
function factorial(n) { return n <= 1 ? 1 : n * factorial(n-1); }
```

Q39: Memoization

- Caching function results for faster future calls.

Q40: Temporal Dead Zone (TDZ)

- let & const variables cannot be accessed before declaration (causes ReferenceError).

DOM Manipulation Basics

Q41: What is the DOM?

- Document Object Model; represents HTML as a tree structure.

Q42: Selecting DOM elements

```
document.getElementById('id');

document.querySelector('.class');

document.querySelectorAll('div');
```

Q43: innerHTML vs innerText

- innerHTML → HTML content
- innerText → text content only

Q44: Event bubbling vs capturing

- **Bubbling:** events propagate **from target → parent**
- **Capturing:** events propagate **from parent → target**

Q45: addEventListener vs onclick

- addEventListener allows multiple listeners and event phases
- onclick overwrites previous handler

Q46: Retrieving character at index

```
'hello'.charAt(1); // 'e'
```

Q47: BOM (Browser Object Model)

- Provides access to browser-specific objects like window, navigator, location, history.

Q48: Client-side vs server-side JS

Feature	Client-side	Server-side

Feature	Client-side	Server-side
Runs	Browser	Node.js
Access	DOM	File system, DB
Use	UI interactivity	APIs, DB operations

Q49: Cookie vs sessionStorage vs localStorage

Feature	Cookie	sessionStorage	localStorage
Lifetime	Set expiry	Session	Permanent
Storage	4KB	5–10MB	5–10MB
Sent to server	Yes	No	No

Q50: Event delegation

- Add event listener to parent; handle child events dynamically.

DOM & Event Handling

Q51: mouseenter vs mouseover

- mouseenter → doesn’t bubble, triggers only when mouse enters the element
- mouseover → bubbles, triggers when mouse enters child elements too

Q52: event.preventDefault() vs event.stopPropagation()

- preventDefault() → stops default action of the element
- stopPropagation() → stops event from bubbling/capturing

Q53: Add/remove/modify HTML elements

```
const div = document.createElement('div');

div.textContent = 'Hello';

document.body.appendChild(div);

document.body.removeChild(div);

div.setAttribute('id','myDiv');
```

Q54: Event listeners usage

```
element.addEventListener('click', () => console.log('Clicked'));
```

Q55: Event phases

- **Capturing** → target → child
- **Target** → at target element
- **Bubbling** → child → parent

Q56: innerHTML vs textContent

- innerHTML → parses HTML tags
- textContent → only text

Q57: Manipulating CSS styles

element.style.color = 'red';

element.style.backgroundColor = 'yellow';

Q58: document.querySelector() vs getElementById()

- querySelector() → CSS selector, first match
- getElementById() → only by ID

Q59: Event delegation

- Assign listener to parent, handle child events via event.target.

Functions

Q60: First-class functions

- Functions can be assigned, passed, or returned like variables.

Q61: Higher-order functions

- Accepts a function as argument or returns a function.

const numbers = [1,2,3];

const doubled = numbers.map(n => n*2);

Q62: Callback functions

- Functions passed as arguments to execute later.

Q63: Pure function

- Output depends only on input; no side effects.

Q64: Currying function

- Transform multi-argument function into single-argument sequence.

Q65: Thunk function

- Returns a function wrapping an expression to delay computation.

Q66: Asynchronous thunks

- Thunks used in async code to delay execution.

Q67: Decorator in JS

- Enhances or modifies functions/classes.

```
function readonly(target, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

Q68: Proper tail call

- Last function call optimization in ES6.

Q69: Anonymous function use cases

- Callbacks, IIFE, event handlers.

Q70: Recursion example

```
function factorial(n) { return n <= 1 ? 1 : n*factorial(n-1); }
```

Q71: Default parameters

```
function greet(name='Guest'){ console.log('Hello ' + name); }
```

Q72: IIFE correction

- function foo(){}(); → needs parentheses around function:

```
(function foo() {})();
```

Q73: Ways to create objects

- Object literal, constructor function, Object.create, ES6 class.

Q74: Dot vs bracket notation

```
obj.name; // dot
```

obj['name']; // bracket

Q75: Array iteration methods

- for, forEach, map, filter, reduce, for...of.

Q76: Add/remove/update array elements

arr.push(4); arr.pop(); arr[0]=10;

Q77: Copy object/array

- Shallow copy: spread ..., Object.assign()
- Deep copy: JSON.parse(JSON.stringify(obj)) or libraries

Q78: Shallow vs deep copy

- Shallow → only top-level properties
- Deep → all nested objects

Q79: Spread operator advantages

- Merge arrays/objects, shallow copy, add elements easily.

Q80: Check object property

obj.hasOwnProperty('key');

'key' in obj;

Q81: Destructuring assignment

const {name, age} = obj;

const [a,b] = [1,2];

Q82: Object.freeze()

- Makes object immutable (cannot add/delete/update properties).

Q83: Object.seal()

- Prevents adding/removing properties; allows modifying existing.

Q84: Object.preventExtensions()

- Prevents adding new properties.

Q85: Array.prototype.slice()

- Returns shallow copy of portion of array.

Q86: Array.prototype.splice()

- Adds/removes elements in-place.

Q87: slice vs splice difference

Method	**Modifies original?**	**Returns?**
slice	No	New array
splice	Yes	Removed elements

Q88: Object vs Map

Feature	**Object**	**Map**
Key type	String/Symbol	Any type
Size	No built-in	map.size
Iteration	for..in	map.forEach

Q89: Object.keys vs Object.getOwnPropertyNames

- keys → enumerable keys
- getOwnPropertyNames → all own keys

Q90: Object prototypes

- Each object has prototype for inheritance.

Q91: Prototype design pattern

- Create object by cloning a prototype instead of new instance.

Q92: Object destructuring

const {name, age} = obj;

Q93: Sets & Maps

- Set → unique values
- Map → key-value pairs

Q94: Map/Set vs WeakMap/WeakSet

- WeakMap/WeakSet → keys are weakly referenced (garbage-collected)

Q95: Set to array

```
const arr = [...mySet];
```

Q96: Map vs plain object

- Map allows any key type, preserves insertion order, has size property.

Q97: Sets & Maps equality check

- Objects are checked by reference, not value.

Q98: Prototype chain

- Object → prototype → prototype … → null

Q99: Prototypal vs classical inheritance

- Prototypal → objects inherit from other objects
- Classical → classes define blueprint, instantiate objects

Q100: Object.create vs new keyword

- Object.create(proto) → new object with prototype proto
- new → calls constructor function, sets prototype automatically

Q101: Prototype chain

- Every object has a prototype; when accessing a property, JS looks up the prototype chain until found or null.

Q102: Prototypal vs Classical inheritance

- Prototypal → objects inherit from other objects directly
- Classical → classes and instances, introduced in ES6

Q103: Object.create vs new keyword

- Object.create(proto) → new object inherits directly from proto
- new → creates instance from constructor function

Q104: Object.freeze vs Object.seal

- freeze → immutable (no add, remove, modify)
- seal → cannot add/remove, but existing properties can be modified

Q105: Deep copy vs shallow copy

- Shallow copy → top-level properties only
- Deep copy → nested objects copied completely

Q106: Deferred scripts

- defer attribute → scripts execute after HTML parsing, maintains order.

Q107: Lexical scoping

- Variables are resolved in the scope where the function was defined, not where called.

Q108: Closures for private variables

```
function counter() {
  let count = 0;
  return () => ++count;
}
const c = counter();
c(); // 1
c(); // 2
```

Q109: Global, function, block scope

- **Global** → anywhere
- **Function** → inside function
- **Block** → inside {} using let/const

Q110: this keyword behavior

- Global → window
- Function → depends on call site
- Method → object
- Arrow → lexically inherited

Q111: this binding ways

- Default binding, implicit binding, explicit binding (call, apply, bind), new binding

Q112: Event handler this

- In DOM events, this refers to the element that received the event

Q113: Attribute vs property in DOM

- Attribute → HTML markup

- Property → JS object representation

Q114: Add/remove/modify elements

```
let p = document.createElement('p');

p.textContent = 'Hello';

document.body.appendChild(p);

document.body.removeChild(p);
```

Q115: Event listeners usage

```
element.addEventListener('click', () => console.log('Clicked'));
```

Asynchronous JavaScript

Q116: Event loop

- JS is single-threaded; event loop handles async callbacks after current execution stack clears.

Q117: Synchronous vs asynchronous functions

- Synchronous → blocking
- Asynchronous → non-blocking, executed later

Q118: Promises

- Object representing eventual completion/failure of async operation.

Q119: Promise states

- pending, fulfilled, rejected

Q120: Promises vs callbacks

- Promises avoid "callback hell", provide .then/.catch chaining.

Q121: Promise.all()

- Resolves when all promises resolve, rejects if any fail

Q122: Promise.allSettled()

- Resolves after all promises finish, regardless of success/failure

Q123: async/await

```
async function fetchData() {
  const res = await fetch('url');

  const data = await res.json();

}
```

Q124: Error handling in async

- Try/catch block:

```
try {

  await asyncFunc();

} catch(e) { console.log(e); }
```

Q125: Microtask queue

- Promises are processed before next macrotask (setTimeout, etc.)

Q126: setTimeout vs setImmediate vs process.nextTick

- setTimeout → scheduled after delay
- setImmediate → next iteration of event loop
- process.nextTick → immediately after current operation (Node.js)

Q127: Prototypal inheritance

- Objects inherit from prototype objects

```
const parent = {name:'Parent'};

const child = Object.create(parent);
```

Q128: Prototype chain

- Accessing property → object → prototype → prototype … → null

Q129: Classical inheritance in ES6 classes

```
class Parent { constructor(name){ this.name=name; } }

class Child extends Parent { }
```

Q130: new keyword

- Creates instance, sets prototype, binds this, returns object

Q131: Constructor function creation

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

```
const p = new Person('Chetan');
```

Q132: ES5 vs ES2015 class

- ES5 → function constructors + prototype
- ES6 → class syntax, extends, static methods

Q133: Arrow functions in constructor

- Lexically bind this, avoid re-binding inside callbacks

Q134: Static class members

- Shared across all instances:

```
class MyClass {
  static greet() { console.log('Hi'); }
}
MyClass.greet();
```

ES6+ Features

Q135: Template literals

```
const name = 'Chetan';
console.log(`Hello ${name}`);
```

Q136: Destructuring assignment

- Object and array destructuring:

```
const {a,b} = {a:1,b:2};
const [x,y] = [1,2];
```

Q137: Default parameters

```
function greet(name='Guest'){ console.log(name); }
```

Q138: JavaScript modules

- export and import to split code into files.

Q139: Generators

```
function* gen() { yield 1; yield 2; }
```

```
const g = gen();

g.next(); // {value:1, done:false}
```

Q140: Classes

- ES6 syntax for constructor, methods, inheritance.

Q141: Symbols

- Unique identifiers for object properties.

Q142: Proxies

- Intercept object operations:

```
const p = new Proxy({}, { get: (target,key)=> key });
```

Q143: Iterators & generators

- Iterators: {next: function}
- Generators: function* yields multiple values

Q144: Mutable vs immutable objects

- Mutable → can change (arrays, objects)
- Immutable → cannot change (primitive types, frozen objects)

Q145: Map vs plain object

- Map → any type keys, preserves order, size property
- Object → string/symbol keys only

Q146: Map/Set vs WeakMap/WeakSet

- Weak → garbage-collected keys, no iteration

Q147: Static class members

- Shared across all instances (repeat of Q134, sometimes asked twice)

Q148: Object getters & setters

```
const obj = {

  a: 10,

  get value(){ return this.a; },

  set value(v){ this.a = v; }
};
```

Q149: Property flags & descriptors

- writable, enumerable, configurable using Object.getOwnPropertyDescriptor().

Q150: Check if object is empty

```
Object.keys(obj).length === 0;
```

Q151: Tree shaking

- Removes unused code during bundling to reduce file size.

Q152: Need for tree shaking

- Optimizes performance, smaller bundle size, faster loading.

Q153: Optimize JS performance

- Minimize DOM access
- Debounce/throttle events
- Lazy load resources
- Use Web Workers for heavy tasks

Q154: Debounce vs Throttle

- **Debounce:** executes after X ms of inactivity
- **Throttle:** executes at most once per X ms

Q155: requestAnimationFrame

- Optimizes animations, syncs with browser repaint.

Q156: Performance bottlenecks

- Heavy DOM manipulation
- Large data processing on main thread
- Excessive reflows/repaints

Q157: Debouncing & throttling example

```
// Debounce

function debounce(fn, delay) {

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


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

Q158: Optimize DOM manipulation

- Batch updates, use DocumentFragment, avoid repeated queries.

Q159: Reduce reflows/repaints

- Minimize layout thrashing, use classList to update styles, cache measurements.

Q160: Lazy loading

- Load resources only when needed, improves page load.

Q161: Web Workers

- Run scripts in background threads, avoid blocking UI.

Q162: Caching

- Store data locally to reduce network calls.

Q163: Tools to measure performance

- Chrome DevTools, Lighthouse, WebPageTest.

Q164: Optimize network requests

- Minify assets, use HTTP/2, cache API responses, debounce/throttle API calls.

Testing & Code Quality

Q165: Types of testing

- Unit, Integration, End-to-End

Q166: Unit vs Integration vs E2E

Type	Scope	Example
Unit	Single function	Test a sum() function
Integration	Module interaction	API + DB calls
E2E	Full flow	User login process

Q167: Popular JS testing frameworks

- Jest, Mocha, Jasmine, Cypress

Q168: Unit test example

```
test('sum adds numbers', ()=>{ expect(sum(2,3)).toBe(5); });
```

Q169: Test-driven development (TDD)

- Write tests before implementation; code passes tests.

Q170: Mocks & stubs

- Mock → simulate object behavior
- Stub → replace function with controlled output

Q171: Testing async code

- Use async/await or return Promises in tests

Q172: Test best practices

- Small, isolated tests
- Clear naming
- Avoid side effects

Q173: Code coverage

- Measures % of code executed by tests, ensures quality

Q174: Tools for JS testing

- Jest, Mocha, Cypress, Karma

Design Patterns

Q175: Design patterns purpose

- Solve common programming problems, improve code maintainability.

Q176: Singleton pattern

- Single instance shared across application:

```
const Singleton = (function() {
  let instance;
  return { getInstance: () => instance || (instance = {}) };
})();
```

Q177: Factory pattern

- Creates objects without exposing instantiation logic:

```
function createShape(type){
  if(type==='circle') return {type:'circle'};
  if(type==='square') return {type:'square'};
}
```

Q178: Observer pattern

- Publish/subscribe model for events.

Q179: Module pattern

- Encapsulates code using closures, exposes API.

Q180: Prototype pattern

- Clone existing objects instead of creating new instances.

Q181: Decorator pattern

- Add behavior dynamically to objects/functions.

Q182: Strategy pattern

- Select algorithm at runtime.

Q183: Command pattern

- Encapsulate request as object; queue/undo operations.

Q184: Extending built-in JS objects

- Not recommended; can break future code.

Security

Q185: Cross-Site Scripting (XSS)

- Injecting malicious scripts; prevent via input sanitization.

Q186: Cross-Site Request Forgery (CSRF)

- Attack that performs actions without user consent; prevent with tokens.

Q187: Prevent SQL injection

- Use parameterized queries; never interpolate inputs.

Q188: Handle sensitive data

- Hash passwords, use HTTPS, avoid localStorage for secrets.

Q189: Content Security Policy (CSP)

- Restrict sources of scripts to prevent XSS.

Q190: Common security headers

- X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security

Q191: Prevent clickjacking

- X-Frame-Options: DENY

Q192: Input validation importance

- Prevent security risks, ensure proper data handling.

Q193: Tools to identify vulnerabilities

- ESLint security plugins, OWASP ZAP, Snyk.

Q194: Secure authentication & authorization

- JWT, OAuth2, role-based access control.

Q195: Same-origin policy

- Restricts JS to access resources from same domain.

Q196: use strict

- Enforces safer JS; prevents undeclared variables, duplicate parameters.

Q197: Debugging tools & techniques

- Chrome DevTools, breakpoints, console.log, debugger statement.

Q198: Garbage collection

- JS automatically frees memory for unreachable objects.

Q199: Single-page app (SPA) & SEO

- Render HTML dynamically; use SSR or prerendering for SEO.

Q200: Share code between JS files

- Use modules: export / import

Q201: Organizing code

- Modular structure, classes, functional decomposition, folder structure

---

Coding Challenges (Examples)

Beginner:

- Reverse string
- Palindrome check
- Factorial
- Fibonacci
- Prime number check

Intermediate:

- Flatten nested arrays
- Debounce/throttle functions
- Deep clone objects
- Memoization

Advanced:

- Polyfills: map, filter, reduce, Promise.all, Promise.race

Behavioral & Conceptual Questions

Debugging JS:

- Use console.log, breakpoints, debugger, DevTools performance tab

Why “debugger”?

- Pauses execution at a line for inspection

Best practices:

- Use const/let, avoid globals, modular code, comments

Cross-browser compatibility:

- Test on multiple browsers, use polyfills/transpilers (Babel)

Security best practices:

- Sanitize input, HTTPS, CSP, use secure cookies, avoid eval()