Introduction to JavaScript and MERN Stack

Overview of JavaScript and the MERN Stack

JavaScript is a **high-level, interpreted scripting language** used to add interactivity, control, and logic to web pages. It is one of the **core technologies of the web**, alongside HTML and CSS. Originally designed for the browser, JavaScript now powers both frontend and backend development.

The MERN Stack: Building Full Web Applications in JavaScript

MERN is a set of technologies that allows developers to build **end-to-end web applications** entirely in JavaScript.

Component	Role
MongoDB	Document-based NoSQL database
Express.js	Lightweight web framework for Node.js
React.js	Frontend library for building user interfaces
Node.js	Backend runtime for running JavaScript on servers

Benefits of MERN:

- Full-stack development with **one language** (JavaScript)
- React for **interactive UIs**, Node & Express for **scalable backend**

- MongoDB for **flexible, JSON-like data models**
- Popular in startups, scalable for enterprise apps

Real-World Analogy:

Imagine building an online shopping app:

- **React** shows the product listing to the user.
- **Express + Node** handle adding to cart, checkout, login.
- **MongoDB** stores product, user, and order data.

Setting Up the Environment

Setup installations Install Node, npm, and Visual Studio Code

Writing Your First JavaScript Program

- Use `console.log()` to print values
- JavaScript is case-sensitive
- Semicolons are optional but recommended

```
let name = "Alice";
console.log("Welcome, " + name);
```

Question: What happens if you forget to declare a variable before using it?

JavaScript Essentials

Variables, Data Types, and Operators (JavaScript Essentials)

Variable Declarations

In modern JavaScript, there are **three ways to declare variables**:

Keyword	Scope	Reassignment	Use Case
`let`	Block scope	Yes	When the value may change
`const`	Block scope	No	When the value should stay same
`var`	Function scope	Yes	**Avoid** in modern code

Example:

```
const name = "Laxman";

let age = 20;

var city = "Mumbai";  // discouraged
```

JavaScript Data Types

1. **Primitive Types**:
    - `string`: `"Hello"`
    - `number`: `42`, `3.14`
    - `boolean`: `true`, `false`
    - `undefined`: declared but not assigned
    - `null`: intentional absence of value
2. **Reference Type**:
    - `object`: `{ key: "value" }`, arrays, functions

Example:

```
let score = 95;                // number
let fullName = "Laxman Bafna";  // string
let isEnrolled = true;         // boolean
let middleName;                // undefined
let spouse = null;             // null
let user = { name: "Laxman", age: 20 }; // object
```

Type	Operators
Arithmetic	`+`, `-`, `*`, `/`, `%`
Assignment	`=`, `+=`, `-=`, etc.

Comparison	`==`, `===`, `!=`, `!==`, `<`, `>`
Logical	`&&`, `

Example:

```
let a = 5;

let b = "5";


console.log(a == b);   // true (loose equality, only value checked)

console.log(a === b);  // false (strict equality, value + type checked)
```

Control Structures (if-else, loops)

If-Else Statements in JavaScript

Overview

- Used to perform **decision-making** in code.
- Syntax allows the program to choose between different paths based on a condition.
- Conditions use **comparison operators** like `===`, `>`, `<`, `!==`, `<=`, `>=`.
- Can be used as:
    - Simple `if`
    - `if-else`

- `if-else if-else`

Syntax

```
if (condition) {
  // code block
} else if (anotherCondition) {
  // another block
} else {
  // default block
}
```

Examples

Example 1: Grading System

```
let marks = 85;

if (marks >= 90) {
  console.log("Grade A");
} else if (marks >= 75) {
  console.log("Grade B");
} else {
  console.log("Grade C");
}
```

Example 2: Voting Eligibility

```
let age = 17;

if (age >= 18) {
  console.log("Eligible to vote");
} else {
  console.log("Not eligible to vote");
}
```

Example 3: Number Sign Checker

```
let num = -3;
```

```
if (num > 0) {
  console.log("Positive");
} else if (num < 0) {
  console.log("Negative");
} else {
  console.log("Zero");
}
```

Example 4: Password Strength Checker

```
let password = "abcd1234";

if (password.length >= 12) {
  console.log("Strong Password");
} else if (password.length >= 8) {
  console.log("Moderate Password");
} else {
  console.log("Weak Password");
}
```

Programming Questions

Question 1: Easy Level

Question:

You are given a variable `num`. Write a conditional block that:

- Prints `"Even"` if the number is even and **not zero**
- Prints `"Zero"` if the number is exactly 0
- Prints `"Odd"` otherwise

Example Input:

```
let num = 0
```

Expected Output:

```
"Zero"
```

Question 2: Medium Level

Write a function getGrade(score, isBonus) that returns the **grade as a string** based on the following:

- If isBonus is true, **add 5** to the score (but max cap is 100).
- Then, assign grade:
  - "A" if score ≥ 90
  - "B" if score ≥ 80 and < 90
  - "C" if score ≥ 70 and < 80
  - "D" if score ≥ 60 and < 70
  - "F" otherwise

1. Functions

Components

- Function declaration/definition
- Parameters and arguments
- Return values
- Function expressions
- Arrow functions
- Higher-order functions
- IIFE (Immediately Invoked Function Expressions)

Properties

- First-class citizens
- Closures
- Can return other functions

Variants with Examples

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

// Function Expression
const greetExpression = function(name) {
  return `Hi, ${name}`;
};

// Arrow Function
const greetArrow = (name) => `Hey, ${name}`;

// IIFE
(function() {
  console.log('This runs immediately');
})();

// Higher-order function
function withLogger(fn) {
  return function(...args) {
    console.log('Arguments:', args);
    return fn(...args);
  };
}
```

Use Cases

- API logic modularization
- Utility functions in large apps
- Event handlers in frameworks

Questions

Medium 1: Create a function that returns the nth Fibonacci number using memoization.

```
function fibonacci(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
}
```

Medium 2: Write a function that returns another function to calculate discount based on user type.

```
function getDiscount(userType) {
  return function(price) {
    const rates = { regular: 0.1, premium: 0.2 };
    return price * (1 - (rates[userType] || 0));
  };
}
```

Difficult: You are designing a throttling mechanism for a search input component that only allows API calls every 300ms. Implement a throttle function and demonstrate its use.

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func(...args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}
```

---

2. Arrays

Components

- Indexed collections
- Zero-based index
- Mutable

Properties

- Dynamic typing
- Iterable
- Length property

Use Cases

- UI data (lists, feeds, tables)
- Normalizing or grouping data
- Search/filter operations

Commonly Used Array Methods

1. forEach()

- Executes a callback for each element.

```
[1, 2, 3].forEach(num => console.log(num));
```

- **Use Case**: Loop through data without returning anything (e.g., render UI, log data).

2. map()

- Transforms each element and returns a new array.

```
const doubled = [1, 2, 3].map(num => num * 2); // [2, 4, 6]
```

- **Use Case**: Modify array elements for rendering (e.g., mapping tasks to UI cards).

3. filter()

- Returns a new array with elements that pass the test.

```
const evens = [1, 2, 3, 4].filter(num => num % 2 === 0); // [2, 4]
```

- **Use Case**: Filter users by role, tasks by status, etc.

4. reduce()

- Applies a function against an accumulator and each element to reduce it to a single value.

```
const sum = [1, 2, 3].reduce((acc, val) => acc + val, 0); // 6
```

- **Use Case**: Calculating totals, aggregating scores, flattening arrays.

5. find()

- Returns the first element that matches a condition.

```
const found = [4, 5, 6].find(num => num > 4); // 5
```

- **Use Case**: Find a user by ID, find a product by name.

8. includes()

- Checks if an array includes a certain element.

```
["a", "b", "c"].includes("b"); // true
```

9. sort()

- Sorts the array in-place. Be careful with numbers.

[3, 1, 2].sort(); // [1, 2, 3] (with comparator)

[3, 1, 2].sort((a, b) => a - b); // correct numeric sort

10. splice() / slice()

- `splice(index, count)`: Modifies array by removing/replacing elements
- `slice(start, end)`: Returns shallow copy of part of array

let arr = [1, 2, 3, 4];
arr.splice(1, 2); // [1, 4]
arr.slice(1, 3); // [2, 3]

Questions

Medium:

1. Implement your own version of `map()`.

```
function myMap(arr, callback) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(callback(arr[i], i, arr));
  }
  return result;
}
```

2. Remove all falsy values from an array using `filter()`.

```
const cleaned = [0, 1, false, 2, '', 3].filter(Boolean); // [1, 2, 3]
```

Medium 3: Write a function that finds the intersection of two arrays.

```
function intersect(arr1, arr2) {
  return arr1.filter(val => arr2.includes(val));
}
```

Difficult:

Case Study Problem:
You are building a dashboard for eco-tasks. Each task has an `impactScore` and `completed` status. Write a function to:

- Filter completed tasks
- Sort by impactScore (descending)
- Return only top 3 impactful tasks

```
function topImpactfulTasks(tasks) {
  return tasks
    .filter(task => task.completed)
    .sort((a, b) => b.impactScore - a.impactScore)
    .slice(0, 3);
}
```

3. Objects

Components

- Key-value pairs
- Nested structures
- Methods and computed keys

Properties

- Reference types
- Extensible
- Dynamic keys

Variants with Examples

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

```
// Access
console.log(user.name);

// Dynamic Key
const key = 'email';
user[key] = 'alice@example.com';

// Destructuring
const { name, age } = user;

// Method
user.greet = function() { return `Hi, I'm ${this.name}`; };

// Object.assign
const extended = Object.assign({}, user, { active: true });
```

Use Cases

- API response mapping
- Component props
- Data models (user, product, task)

Questions

Medium 1: Create a function to deeply clone a nested object.

function deepClone(obj) {
  return JSON.parse(JSON.stringify(obj));
}

Medium 2: Write a function to compare two objects shallowly.

```
function shallowEqual(obj1, obj2) {
  const keys1 = Object.keys(obj1);
  const keys2 = Object.keys(obj2);
  if (keys1.length !== keys2.length) return false;
  return keys1.every(key => obj1[key] === obj2[key]);
}
```

Difficult: In an e-commerce app, you need to merge two cart states (original and updated) ensuring quantities are summed for repeated product IDs.

```
function mergeCarts(cart1, cart2) {
  const merged = {};
  [...cart1, ...cart2].forEach(item => {
    merged[item.id] = merged[item.id] || { ...item, quantity: 0 };
    merged[item.id].quantity += item.quantity;
  });
  return Object.values(merged);
}
```

---

4. Loops

Components

- Initialization
- Condition
- Iteration expression

Properties

- Repeated execution
- Exit condition

Variants with Examples

```
// For loop
for (let i = 0; i < 5; i++) console.log(i);

// While loop
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}

// For...of
for (const value of [10, 20, 30]) {
  console.log(value);
}

// For...in
const obj = { a: 1, b: 2 };
for (let key in obj) {
  console.log(key, obj[key]);
}
```

Use Cases

- Iterating UI lists
- Searching and modifying data
- Grouping records

Questions

Medium 1: Write a loop to print factorial of a number.

```
function factorial(n) {
  let result = 1;
  for (let i = 2; i <= n; i++) result *= i;
  return result;
}
```

Medium 2: Create a loop that filters out users younger than 18 from an array of user objects.

```
function filterMinors(users) {
  const result = [];
  for (let user of users) {
    if (user.age >= 18) result.push(user);
  }
  return result;
}
```

Difficult: You are given a matrix representing terrain elevations. Write a function to flatten it into a 1D array while preserving row-wise order.

```
function flattenMatrix(matrix) {
  const result = [];
  for (let row of matrix) {
    for (let val of row) {
      result.push(val);
    }
  }
  return result;
}
```

---

5. ES6+ Features

Components

- let/const
- Arrow functions
- Template literals
- Destructuring
- Spread/rest
- Default parameters
- Enhanced object literals

Properties

- Reduces verbosity
- Improves readability
- Encourages immutability

Variants with Examples

```
// let/const
let a = 5;
const PI = 3.14;

// Template literals
const greeting = `Hello, ${name}`;

// Arrow functions
const sum = (a, b) => a + b;

// Destructuring
const [x, y] = [1, 2];
const { name, age } = { name: 'Sam', age: 22 };

// Spread
const arr2 = [...arr1];

// Rest
function logAll(...args) { console.log(args); }

// Default params
function power(base, exp = 2) { return base ** exp; }
```

Use Cases

- Cleaner React components
- API response processing
- Function configurations

Questions

Medium 1: Use rest/spread to write a function that removes a property from an object.

```
function omit(obj, key) {
  const { [key]: _, ...rest } = obj;
  return rest;
}
```

Medium 2: Create a function that returns the sum of all arguments using rest.

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

Difficult: In a configuration system, write a mergeConfig function that combines default and user config using destructuring and spread for deep nested structures.

```
function mergeConfig(defaults, userConfig) {
  return {
    ...defaults,
    ...userConfig,
    nested: {
      ...defaults.nested,
      ...userConfig.nested
    }
  };
}
```

JavaScript Core Concepts: Functions, Arrays, Objects, Loops, ES6+

[Previous sections remain unchanged above]

---

6. Promises

Components

- Promise constructor
- `.then()`, `.catch()`, `.finally()`
- States: pending, fulfilled, rejected

Properties

- Used for async operations
- Chainable with then/catch

Variants with Examples

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Done'), 1000);
});

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

Use Cases

- API calls
- File I/O (Node.js)
- Sequencing async operations

Difficult Problem

You are building a UI that fetches multiple endpoints: `/user`, `/posts`, and `/notifications`. Implement a `fetchAllData` function that executes all three API calls in parallel and returns the combined result with error handling.

```
function fetchAllData() {
  const endpoints = [
    fetch('/user'),
    fetch('/posts'),
```

```
    fetch('/notifications')
  ];

  return Promise.allSettled(endpoints)
    .then(results => {
      const data = {};
      results.forEach((result, index) => {
        const name = ['user', 'posts', 'notifications'][index];
        data[name] = result.status === 'fulfilled' ? result.value : null;
      });
      return data;
    })
    .catch(error => {
      console.error('Unexpected error:', error);
      return null;
    });
}
```

---

7. Async/Await

Components

- `async` keyword
- `await` for promises
- Error handling with try/catch

Properties

- Syntactic sugar over promises
- Allows linear-style async code

Variants with Examples

```
async function getData() {
  try {
    const res = await fetch('/api');
    const json = await res.json();
```

```
    return json;
  } catch (error) {
    console.error(error);
  }
}
```

Use Cases

- API data fetching
- Sequential async flows

Difficult Problem

Write an async function `fetchWithRetry` that retries an API call up to 3 times before throwing an error.

```
async function fetchWithRetry(url, attempts = 3) {
  let lastError;
  for (let i = 0; i < attempts; i++) {
    try {
      const response = await fetch(url);
      if (!response.ok) throw new Error('Failed');
      return await response.json();
    } catch (error) {
      lastError = error;
      console.warn(`Attempt ${i + 1} failed`);
    }
  }
  throw new Error(`All ${attempts} attempts failed: ${lastError}`);
}
```

---

8. JSON

Components

- `JSON.stringify()`

- `JSON.parse()`
- Standard JSON format (keys as strings)

Properties

- Interchange format between server and client
- Language-agnostic

Variants with Examples

```
const obj = { name: 'Alice', age: 25 };
const str = JSON.stringify(obj);
const parsed = JSON.parse(str);
```

Use Cases

- AJAX/REST communication
- Local storage

Difficult Problem

Write a function `safeParse` that safely parses a JSON string, detects circular references, and returns a custom error message.

```
function safeParse(str) {
  try {
    return { valid: true, data: JSON.parse(str) };
  } catch (err) {
    return { valid: false, error: 'Invalid JSON format' };
  }
}

// Optional: Custom stringifier that avoids circular
function safeStringify(obj) {
  const seen = new WeakSet();
  return JSON.stringify(obj, (key, value) => {
    if (typeof value === 'object' && value !== null) {
      if (seen.has(value)) return '[Circular]';
```

```
    seen.add(value);
  }
  return value;
});
}
```

---

9. Closures

Components

- Inner function retains access to outer scope
- Used in data privacy, currying

Properties

- Lexical scoping
- Persistent state across calls

Variants with Examples

```
function counter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}
const increment = counter();
```

Use Cases

- Private variables
- Function factories

Difficult Problem

Create a timer utility that provides methods like `start`, `pause`, `resume`, and `reset` using closures to maintain internal state.

```
function createTimer() {
  let startTime = null;
  let elapsed = 0;
  let timerId = null;

  return {
    start() {
      if (timerId) return;
      startTime = Date.now();
      timerId = setInterval(() => {
        elapsed = Date.now() - startTime;
      }, 100);
    },
    pause() {
      if (!timerId) return;
      clearInterval(timerId);
      timerId = null;
      elapsed = Date.now() - startTime;
    },
    resume() {
      if (timerId || startTime === null) return;
      startTime = Date.now() - elapsed;
      timerId = setInterval(() => {
        elapsed = Date.now() - startTime;
      }, 100);
    },
    reset() {
      clearInterval(timerId);
      startTime = null;
      elapsed = 0;
      timerId = null;
    },
    getTime() {
      return elapsed;
    }
  };
}
```

---

10. Hoisting

Components

- Declarations move to top
- Applies to `var`, function declarations

Properties

- Only declarations are hoisted, not initializations
- `let` and `const` are hoisted but in TDZ

Variants with Examples

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

hoisted();
function hoisted() {
  console.log('Function hoisted');
}
```

Use Cases

- Interview clarification
- Debugging unexpected reference errors

Difficult Problem

Simulate a transpiler that scans code and highlights hoisted variables and functions.

```
function analyzeHoisting(code) {
  const varMatches = [...code.matchAll(/var\s+(\w+)/g)].map(m => m[1]);
  const funcMatches = [...code.matchAll(/function\s+(\w+)/g)].map(m => m[1]);
  return {
    hoistedVariables: varMatches,
    hoistedFunctions: funcMatches
```

```
  };
}

const codeSample = `
var x = 10;
function greet() {}
var y = 5;
function sum() {}
`;

console.log(analyzeHoisting(codeSample));
```

---

11. Event Handling

Components

- `addEventListener`
- Event object
- Bubbling/Capturing

Properties

- Event delegation
- Can be attached dynamically

Variants with Examples

```
const button = document.getElementById('submit');
button.addEventListener('click', event => {
  console.log('Clicked', event.target);
});
```

Use Cases

- UI interactions

- Form validation
- SPA routing

Difficult Problem

Implement a delegated event system that attaches one listener to a parent and handles events for dynamically added child elements.

```
function delegate(parent, selector, type, handler) {
  parent.addEventListener(type, event => {
    if (event.target.matches(selector)) {
      handler(event);
    }
  });
}

const list = document.getElementById('todo-list');

delegate(list, 'li', 'click', event => {
  console.log('Clicked on item:', event.target.textContent);
});

// Works for dynamic additions
const newItem = document.createElement('li');
newItem.textContent = 'New Task';
list.appendChild(newItem);
```

---

12. Spread / Rest

Components

- `...` operator
- Expands (spread) or collects (rest)

Properties

- Used in function args, arrays, objects

Variants with Examples

```
// Spread in arrays
const arr1 = [1, 2];
const arr2 = [...arr1, 3];

// Rest in functions
function log(...args) {
  console.log(args);
}

// Spread in objects
const obj1 = { a: 1 };
const obj2 = { ...obj1, b: 2 };
```

Use Cases

- Cloning data
- Merging objects/arrays
- Flexible function arguments

Difficult Problem

Create a logger utility that supports variable number of labeled parameters using rest and prints them in formatted string.

```
function formatLogger(...entries) {
  entries.forEach(entry => {
    const [label, value] = entry;
    console.log(`[${label.toUpperCase()}]: ${JSON.stringify(value)}`);
  });
}

const user = { id: 1, name: 'John' };
const error = { code: 500, msg: 'Internal Error' };
const metadata = { ts: Date.now() };

formatLogger(
  ['user', user],
  ['error', error],
  ['meta', metadata]
```

);