JavaScript 100+ Theory Interview Q&A, 60+ Output Based Q&A, 50+ Tasks

This page is your complete guide to preparing for **JavaScript** interviews. We have included **100+ theory questions**, **60+ output-based questions**, and **50+ coding tasks with solutions** to help you learn and practice effectively. Each question is explained in simple words so that you can easily understand even the most advanced topics.

What's Included:

- **Theory Questions**: Cover the basics and advanced JavaScript concepts in a simple way.
- **Output-Based Questions**: Real examples to test how well you understand JavaScript behaviour.
- **Coding Tasks**: Practical problems with step-by-step solutions to help you build confidence.

Topics Covered:

- Introduction to JavaScript
- Type Coercion
- Data Types
- Operators
- Conditional Statements
- Switch Statements and Loops
- Arrays
- Functions
- Objects
- DOM (Document Object Model)
- Event Handlers
- Event Propagation and Delegation
- Event Loop
- Spread Operator, Rest Parameter, and Destructuring
- Prototype
- Classes
- Arrow Functions, Closures, and Callbacks
- Higher-Order Functions (HOF), map, filter, and reduce
- Debouncing and Throttling
- Modules, Cookies, Local Storage, and Session Storage

JavaScript 100+ Theory Interview Questions

Introduction to JavaScript:

1. What are JavaScript engines, and why are they important?

Answer:

1. JavaScript engines are programs that execute JavaScript code.
2. Examples of JavaScript engines:
   - **V8 Engine**: Used in Chrome and Node.js.
   - **SpiderMonkey**: Used in Firefox.
3. They optimize JavaScript code for faster execution.
4. Enable modern web capabilities by enhancing performance and speed.
5. Critical for running JavaScript applications in both browsers and server environments.

2. How is JavaScript executed in the browser?

Answer:

1. The browser's JavaScript engine (e.g., V8 for Chrome) reads and interprets JavaScript code.
2. Converts JavaScript into machine code using techniques like **Just-In-Time (JIT) compilation**.
3. Executes the machine code to produce dynamic and interactive results on the webpage.
4. Handles updates to the DOM and browser APIs efficiently.
5. Enables smooth interactivity and responsiveness in modern web appli

3. What are the limitations of JavaScript?

Answer:

1. **Security**: Can be exploited for malicious purposes if not handled correctly.
2. **Browser Dependency**: JavaScript behavior may vary slightly across different browsers.
3. **No Multithreading**: Runs on a single thread, which can limit performance for heavy computations.
4. **Sandboxed Environment**: Cannot access system-level resources directly.
5. **Dynamic Typing**: While flexible, it can lead to runtime errors if types are misused.

4. How do you include JavaScript in an HTML file?

Answer:

1. **Inline**: Define JavaScript code directly within an HTML element.

   Example:

```
<button onclick="alert('Hello!')">Click Me</button>;
```

2. **Internal**: Use a <script> tag within the same HTML file.

Example:

```
<script>console.log('Hello');</script>;
```

3. **External**: Link an external JavaScript file using the <script> tag with the src attribute.

   Example:

   ```
   <script src="script.js"></script>;
   ```

4. Inline scripts are ideal for small tasks, while external scripts improve maintainability for larger applications.
5. Internal scripts allow defining custom logic within the same file without external dependencies.

5. Why is JavaScript considered single-threaded?

Answer:

1. JavaScript uses a **single-threaded event loop** to execute code sequentially.
2. This design avoids the complexities and potential bugs of multithreading.
3. Handles **asynchronous operations** like API calls using callbacks, promises, and async/await.
4. Efficiently processes tasks via an event loop and a callback queue.
5. Ensures simplicity and predictability in application behavior.

6. Is JavaScript a Dynamically Typed and Why?

Answer:

1. Yes, JavaScript is dynamically typed, meaning variable types are determined at runtime.
2. Variables do not require explicit type declarations during initialization.
3. The type of a variable can change dynamically during execution.
4. Allows flexibility and rapid prototyping but increases the risk of runtime errors if types are misused.

   Example:

Example of Dynamic Typing in JavaScript:

```
// Declaring a variable without specifying a type
let data;
// Assigning a number to the variable
data = 42;
console.log(typeof data); // Output: "number"
// Reassigning a string to the same variable
data = "Hello, world!";
console.log(typeof data); // Output: "string"
// Reassigning a boolean to the same variable
```

```
data = true;
console.log(typeof data); // Output: "boolean"
```

Why is JavaScript Dynamically Typed?
Answer:

1. Variables do not have a fixed type.
2. The type of a variable is determined by the value assigned to it.
3. This flexibility allows for rapid prototyping but can lead to runtime errors if types are misused.

Comparison with Statically Typed Languages:

In statically typed languages like **Java**, you must define the variable's type at declaration, and it cannot change later.

Example in Java (Statically Typed Language):

```
int number = 42; // Variable type is explicitly declared
number = "Hello"; // Error: Incompatible types
```

Data Types :

7. What is the difference between primitive and non-primitive data types?

Answer:

Feature	**Primitive Types**	**Non-Primitive Types**
Mutability	Immutable (values cannot be changed)	Mutable (can change values)
Storage	Stored directly in the variable	Stored as a reference in memory
Examples	Number, String, Boolean, Undefined, Null, Symbol, BigInt	Object, Array, Function

8. What is the difference between undefined and null?

Answer:

Undefined:

1. A variable is declared but not assigned a value.
2. Indicates a variable is not initialized.

```
let x;
console.log(x); // Output: undefined
```

Null:

1. Represents the intentional absence of any value.
2. It is explicitly assigned.

```
let y = null;
console.log(y); // Output: null
```

9. How does JavaScript handle type coercion?

Answer:

- Automatically converts one data type to another when required.
- Coercion depends on the operation being performed.

```
console.log('5' - 2);  // Output: 3 (string '5' is coerced to a number)
console.log('5' + 2);  // Output: '52' (number 2 is coerced to a string)
```

- Coercion can happen with operators like +, -, ==, etc.

10. What are Symbol and BigInt in JavaScript?

Answer:

Symbol:

- Introduced in ES6.
- Represents a unique, immutable identifier.
- Commonly used as object keys to avoid property name conflicts.

```
const sym1 = Symbol('unique');
const sym2 = Symbol('unique');
console.log(sym1 === sym2); // Output: false
```

BigInt:

- Introduced in ES11 (ES2020).
- Used for numbers larger than 2^53 - 1, which cannot be safely represented using Number.

Example:

```
const bigNum = 1234567890123456789012345678901234567890n;
console.log(bigNum); // Output: 1234567890123456789012345678901234567890n
```

13. What is NaN in JavaScript?

Answer:

- **NaN (Not-a-Number):** A special numeric value indicating a computation result that isn't a valid number.
- It is of type Number.

```
console.log(typeof NaN); // Output: number
console.log(0 / 0);      // Output: NaN
```

14. How are objects and arrays different in JavaScript?

Answer:

Object:

- Stores key-value pairs.
- Keys can be strings or symbols.

```
const obj = { name: "John", age: 30 };
```

Array:

- Stores ordered data (indexed by numbers).

```
const arr = [1, 2, 3, 4];
```

Operators:

15. How does the ternary operator work?

Answer: The ternary operator is a shorthand for an if-else statement. The syntax is:

```
condition ? expression1 : expression2;
```

Example:

```
const age = 18;
const canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // Output: Yes
```

16. What is the difference between null and undefined with the == and === operators?

Answer:

- **== (Abstract Equality):** Compares values after type coercion.

```
console.log('5' == 5); // Output: true
```

=== (Strict Equality): Compares values **and types** without type coercion.

```
console.log('5' === 5); // Output: false
```

null and undefined with the == and === operators

```
console.log(null == undefined); // Output: true
```
The values are treated as equivalent

```
console.log(null === undefined); // Output: false
```
Strict comparison considers their types different.

17. Explain the typeof operator with examples.

Answer: The typeof operator returns the type of a variable.

Examples:

```
console.log(typeof 42); // Output: number
console.log(typeof "Hello"); // Output: string
console.log(typeof true); // Output: boolean
console.log(typeof undefined); // Output: undefined
console.log(typeof null); // Output: object (a legacy bug in JavaScript)
console.log(typeof []); // Output: object
console.log(typeof function(){}); // Output: function
```

18. What is the difference between ?? and || operator in JavaScript?

Answer:

??:
Specifically designed for scenarios where you want to handle **null or undefined values only** and leave other falsy values untouched.

Why use ??:

- Prevents overriding valid falsy values like 0, false, or "".
- Makes code more explicit when dealing with nullish values.

```
let result = value1 ?? value2;
```
If value1 is **not null or undefined**, the result is value1. If value1 is **null or undefined**, the result is value2.

Example:

```
let username = null;
let defaultUsername = "Guest";

let finalUsername = username ?? defaultUsername;

console.log(finalUsername); // Output: "Guest"
```

username is null, so defaultUsername is used.

||:
Designed for cases where you want to handle **all falsy values**, not just null or undefined. This is useful when any falsy value represents an invalid input and needs a fallback.

Why use ||:

- Provides a general-purpose fallback mechanism for any falsy value.
- Common in scenarios like defaulting an empty string, 0, or false to another value.

Example:

```
let age = 0; // User input, 0 is falsy
console.log(age || 18); // Output: 18 (0 is treated as invalid)
```

19. What is the difference between ++x and x++?

Answer:

- **++x (Pre-increment):** Increments the value of x and returns the updated value.

```
let x = 5;
console.log(++x); // Output: 6
```

- **x++ (Post-increment):** Returns the current value of x and then increments it.

```
let x = 5;
console.log(x++); // Output: 5
console.log(x);   // Output: 6
```

control flow:

20. What are the types of control flow statements in JavaScript?

Answer:

1. **Conditional Statements:**
   - if, else if, else
   - switch
2. **Loops:**
   - for, while, do...while, for...in, for...of
3. **Jump Statements:**
   - break, continue
4. **Exception Handling:**
   - try...catch, finally, throw

21. How do break and continue statements work in loops?

Answer:

- **break:** Terminates the loop entirely.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) break;
  console.log(i);
}
// Output: 0, 1, 2
```

- **continue:** Skips the current iteration and moves to the next one.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) continue;
  console.log(i);
}
// Output: 0, 1, 2, 4
```

22. How does JavaScript's event loop affect control flow?

Answer: The **event loop** ensures that JavaScript executes code in a non-blocking way. Tasks are executed in the following order:

1. **Call Stack:** Executes synchronous code.
2. **Task Queue:** Executes asynchronous callbacks after the stack is cleared.
3. **Microtasks:** Promises and async/await are executed before the task queue.

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

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

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

console.log("End");

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

Array:

23. What are some common array methods in JavaScript?

Answer:

Method	Description	Example
push()	Adds elements to the end of the array	arr.push(4)

Method	Description	Example
pop()	Removes the last element	arr.pop()
shift()	Removes the first element	arr.shift()
unshift()	Adds elements to the beginning	arr.unshift(0)
splice()	Adds/removes elements at a specific index	arr.splice(1, 2)
slice()	Returns a subarray without modifying the original	arr.slice(1, 3)
map()	Creates a new array by applying a function	arr.map(x => x * 2)
filter()	Filters elements based on a condition	arr.filter(x => x > 2)
reduce()	Reduces the array to a single value	arr.reduce((a, b) => a + b, 0)
forEach()	Iterates through the array	arr.forEach(x => console.log(x))
includes()	Checks if the array contains a value	arr.includes(3)
find()	Returns the first element that matches a condition	arr.find(x => x > 2)

24. What is the difference between splice() and slice()?

Answer:

Feature	splice()	slice()
Modification	Modifies the original array	Does not modify the original array
Purpose	Adds/removes elements at a specific index	Extracts a portion of the array
Example	arr.splice(1, 2)	arr.slice(1, 3)

25. What is the difference between for...of and for...in with arrays?

Answer:

Feature	for...of	for...in
Iterates Over	Values of the array	Indices of the array
Example	for (let value of arr)	for (let index in arr)

Example:

```
const arr = [10, 20, 30];

for (let value of arr) {
  console.log(value); // Output: 10, 20, 30
}

for (let index in arr) {
  console.log(index); // Output: 0, 1, 2
}
```

26. What is the purpose of the map() method in JavaScript?

Answer:

1. The map() method creates a new array by applying a function to every element of the original array.
2. It does not modify the original array but returns a transformed version of it.
3. Useful for performing operations like calculations, formatting, or applying transformations on arrays.
4. Works on all elements of the array, and the original array remains unchanged.

```
const numbers = [1, 2, 3];
const squares = numbers.map(num => num * num);
console.log(squares); // Output: [1, 4, 9]
```

27. How does the find() method work, and how is it different from filter()?

Answer:

1. **find() method:**
   - Returns the first element that satisfies a specified condition.
   - Stops searching as soon as it finds a match.
   - Returns the value directly, not in an array.
2. **filter() method:**
   - Returns all elements that satisfy a condition as a new array.
   - Does not stop after finding one match.
3. Use find() when you only need the first match.
4. Use filter() when you need all matches.

```
const numbers = [1, 2, 3, 4];
const firstEven = numbers.find(num => num % 2 === 0); // Finds the first even number
console.log(firstEven); // Output: 2

const allEvens = numbers.filter(num => num % 2 === 0); // Finds all even numbers
console.log(allEvens); // Output: [2, 4]
```

28. What is the purpose of the flat() method?

Answer:

1. The flat() method is used to flatten nested arrays into a single-level array.
2. By default, it flattens one level of nesting.
3. You can specify the depth of flattening as an argument.
4. To fully flatten a deeply nested array, use flat(Infinity).

Example:

```
const arr = [1, [2, [3, [4]]]];
```

```
console.log(arr.flat(2)); // Output: [1, 2, 3, [4]]
```

29. How does some() differ from every()?

Answer:

1. **some() method:**
   - Checks if **at least one** element satisfies the condition.
   - Stops checking once a match is found.
   - Returns true or false.
2. **every() method:**
   - Checks if **all elements** satisfy the condition.
   - Stops checking once a mismatch is found.
   - Returns true or false.
3. Both methods are short-circuiting, meaning they stop as soon as the result is determined.
4. some() is useful for partial matches, while every() is for all matches.

Example:

```
const numbers = [1, 2, 3, 4];
console.log(numbers.some(num => num > 3)); // Output: true (at least one element > 3)
console.log(numbers.every(num => num > 3)); // Output: false (not all elements > 3)
```

30. How can you check if a value exists in an array?

Answer:

1. Use the includes() method to check if an array contains a specific value.
2. It returns true if the value is found and false otherwise.
3. Works for both primitive values and exact matches.
4. Does not work for object references unless the reference itself matches.

Example:

```
const numbers = [1, 2, 3];
console.log(numbers.includes(2)); // Output: true
console.log(numbers.includes(4)); // Output: false
```

31. What is the difference between forEach() and map()?

Answer:
The forEach() and map() methods are both used to iterate over arrays, but they serve different purposes and have distinct characteristics:

1. **Purpose:**
   - forEach() is used for **executing a function** on each element of the array. It is commonly used for side effects, such as logging or modifying external variables, but it does not return a new array.

    - map() is used for **transforming data**. It applies a function to each element of the array and returns a new array with the transformed values.
2. **Return Value:**
    - forEach() does not return anything (its return value is undefined). It is designed for actions, not for creating a new array.
    - map() always returns a new array with the same length as the original, containing the results of the callback function.
3. **Chaining:**
    - forEach() cannot be chained because it does not return a new array.
    - map() can be chained with other array methods like filter() or reduce() because it returns a new array.

```
const numbers = [1, 2, 3];
numbers.forEach(num => console.log(num * 2)); // Logs: 2, 4, 6
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```

4. Performance:

- In most cases, there is no significant performance difference, but map() is better suited for creating transformed arrays since it is more declarative.

Function:

32. What is the difference between function declarations and function expressions?

Answer:

Function Declaration:

1. A named function is defined using the function keyword.
2. It is hoisted, meaning it can be used before its declaration in the code.

Example:

```
console.log(square(4)); // Output: 16
function square(num) {
  return num * num;
}
```

Function Expression:

1. A function is assigned to a variable, often anonymous.
2. It is not hoisted, meaning it cannot be used before its definition in the code.

Example:

```
const square = function(num) {
  return num * num;
};
```

```
console.log(square(4)); // Output: 16
```

Key Differences:

4. Function declarations are defined at parse time (hoisting), while function expressions are defined at runtime.

5. Function declarations must have a name, whereas function expressions can be anonymous.

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

Answer:
Arrow Functions:

1. Introduced in ES6 as a concise syntax for defining functions.
2. They use the => (arrow) syntax instead of the function keyword.
3. Do not have their own this context; they inherit this from their surrounding scope.
4. Do not have an arguments object, unlike regular functions.

```

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

Differences from Regular Functions:

1. Arrow functions cannot be used as constructors (do not work with the new keyword).
2. They do not bind their own this context, making them suitable for use in callbacks.
3. They provide a more concise syntax, especially for single-expression functions.

Example:

```
function regularFunction() {
  console.log(this); // Refers to the function's context
}
const arrowFunction = () => {
  console.log(this); // Inherits from the surrounding scope
};
regularFunction.call({ key: "value" });
arrowFunction.call({ key: "value" });
```

34. What is the difference between call(), apply(), and bind() in JavaScript?

Answer:
These methods are used to set the this context explicitly in JavaScript:

1. call(): Invokes a function immediately with the provided this and arguments passed individually.
Example:

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

2. apply(): Similar to call(), but arguments are passed as an array.
Example:

```
greet.apply(person, ["Hi"]); // Output: Hi, Mahesh
```

3. bind(): Does not invoke the function immediately. Instead, it creates a new function with the specified this.

```
const boundGreet = greet.bind(person, "Hey");
boundGreet(); // Output: Hey, Mahesh
```

35. What is the difference between return and console.log() in a function?

Answer:

return:

1. Ends the function's execution and sends the result back to the caller.
2. The returned value can be stored in a variable for later use.
3. Stops further code execution within the function.
4. Used to pass data back to the point where the function was called.
5. Only one value can be returned per function call.

```
function add(a, b) {
  console.log(a + b); // Logs the result
  return a + b;       // Returns the result
}
```

console.log():

1. Outputs a message or value to the console for debugging purposes.
2. Does not stop the function’s execution.
3. Does not return any value.
4. Primarily used to track or debug code behavior.
5. Cannot be used for further calculations or assignments.

```
const result = add(2, 3); // Logs 5 and returns 5
console.log(result); // Output: 5
```

36. How do you handle default parameters in JavaScript functions?

Answer:

1. Default parameters allow a function to use predefined values when no argument or undefined is provided for the parameter.
2. Helps avoid errors caused by missing arguments.
3. Default values are assigned in the function definition using the = operator.
4. Introduced in ES6 for better handling of optional parameters.
5. Makes code cleaner and more readable by eliminating the need for conditional checks for undefined parameters.

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}

console.log(greet()); // Output: Hello, Guest!
console.log(greet("Mahesh")); // Output: Hello, Mahesh!
```

Object:

37. What is the difference between dot notation and bracket notation in accessing object properties?

Answer:
Dot notation (.) and bracket notation ([]) are two ways to access properties in an object:

1. Dot Notation: Used when the property name is a valid identifier (e.g., no spaces or special characters).

```
const person = { name: "Mahesh" };
console.log(person.name); // Output: Mahesh
```

2. Bracket Notation: Used when the property name is dynamic, stored in a variable, or contains special characters.

```
const person = { "full name": "Mahesh" };
console.log(person["full name"]); // Output: Mahesh
```

Bracket notation provides more flexibility but is slightly less readable compared to dot notation.

38. What is the difference between Object.keys(), Object.values(), and Object.entries()?

Answer:
These methods provide different ways to access an object's data:

Object.keys(): Returns an array of the object's property names (keys).

```
const obj = { a: 1, b: 2 };
console.log(Object.keys(obj)); // Output: ['a', 'b']
```

Object.values(): Returns an array of the object's property values.

```
console.log(Object.values(obj)); // Output: [1, 2]
```

Object.entries(): Returns an array of key-value pairs as subarrays.

```
console.log(Object.entries(obj)); // Output: [['a', 1], ['b', 2]]
```

These methods are helpful for iterating over objects or transforming them into other structures.

39. What is the purpose of Object.freeze() and Object.seal()?

Answer:

- **

Object.freeze():** Prevents modifications to an object. You cannot add, delete, or change its properties. The object becomes immutable.
Example:

```
const obj = { a: 1 };
Object.freeze(obj);
obj.a = 2; // This has no effect
obj.b = 3; // This also has no effect
console.log(obj); // Output: { a: 1 }
```

Object.seal(): Prevents adding or deleting properties but allows modifying existing ones.
Example:

```
const obj = { a: 1 };
Object.seal(obj);
obj.a = 2; // Allowed
obj.b = 3; // Not allowed
console.log(obj); // Output: { a: 2 }
```

Both methods are used to control the mutability of objects for maintaining strict data integrity.

40. What is the difference between Object.create() and the new keyword?

Answer:

Object.create(): Creates a new object with a specified prototype. It gives you fine-grained control over inheritance.
Example:

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

new Keyword: Creates an object instance from a constructor function. It initializes the object using the constructor logic.
Example:

```

function Person(name) {
  this.name = name;
}
const person = new Person("Mahesh");
console.log(person.name); // Output: Mahesh
```

DOM:

41. What is the difference between innerHTML, innerText, and textContent?

Answer:

1. innerHTML: Gets or sets the HTML content inside an element, including HTML tags. It is useful for rendering complex HTML structures but can expose your code to security risks like XSS if used with untrusted content.

```
element.innerHTML = "<strong>Bold Text</strong>";
```

2. innerText: Gets or sets the text inside an element, excluding hidden elements. It reflects what is rendered visually on the page.

```
element.innerText = "Visible Text";
```

3. textContent: Gets or sets the text inside an element, including hidden elements, but ignores any HTML tags.

```
element.textContent = "Plain Text";
```

Choose the appropriate property based on whether you need plain text, formatted text, or raw HTML.

42. How do you add, remove, or modify DOM elements dynamically?

Answer:
You can manipulate DOM elements dynamically using various methods:

Create a new element:

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

Add the element to the DOM:

```
document.body.appendChild(newElement);
```

Remove an element:

```
const element = document.getElementById("removeMe");
element.remove();
```

Modify attributes or styles:

```
newElement.setAttribute("id", "greeting");
newElement.style.color = "blue";
```

These methods allow you to create interactive and dynamic web pages.

43. How do you manipulate classes on DOM elements?

Answer:

1. Use the classList property to manage CSS classes on an element.
2. The classList API provides methods to add, remove, toggle, or check for a class.
3. It simplifies class manipulations compared to working directly with the className property.
4. Allows easy addition of multiple classes in one statement.
5. Provides a cleaner and more efficient way to manage classes dynamically.

```
const element = document.getElementById("myElement");

// Add a class
element.classList.add("active");

// Remove a class
element.classList.remove("inactive");

// Toggle a class
element.classList.toggle("hidden");

// Check if a class exists
console.log(element.classList.contains("active")); // Output: true
```

The classList API simplifies class manipulations compared to working directly with the className property

Event:

44. What is event delegation, and why is it useful in the DOM?

Answer:

1. Event delegation is a technique where a single event listener is attached to a parent element to handle events triggered by its child elements.
2. It leverages the concept of event bubbling, where events propagate from child elements to their parents.
3. Reduces the number of event listeners needed, improving performance, especially for dynamically created elements.
4. Simplifies event handling for dynamically added or removed child elements.
5. Uses the event.target property to identify the specific child element that triggered the event.

```
document.getElementById("parent").addEventListener("click", function (event) {
  if (event.target && event.target.tagName === "BUTTON") {
    console.log(`Button clicked: ${event.target.textContent}`);
  }
});
```

45. How do you handle events in the DOM?

Answer:
Events in the DOM can be handled using:

1. **Inline Handlers**: Specified directly in HTML (not recommended for large applications).

```
<button onclick="alert('Clicked!')">Click Me</button>;
```

Directly Setting Event Handlers: Assign a function to an element’s event property.

```
const button = document.getElementById("btn");
button.onclick = () => alert("Clicked!");
```

Using addEventListener(): Preferred for modern JavaScript because it allows multiple handlers for the same event.

```
button.addEventListener("click", () => alert("Clicked!"));
```

The addEventListener() method also supports options like once (for one-time events) and capture (for capturing phase).

46. What is the difference between capturing and bubbling in event propagation?

Answer:
Event propagation describes how events flow through the DOM tree. It occurs in three phases:

1. **Capturing Phase**: The event starts at the root and travels down to the target element.
2. **Target Phase**: The event reaches the target element.
3. **Bubbling Phase**: The event travels back up from the target to the root.

By default, events are handled in the bubbling phase unless the capture option is set to true in addEventListener().

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

46. How do you prevent the default behaviour of an event?

Answer:.

1. Use the preventDefault() method to stop the browser's default action for an event.
2. It is commonly used to prevent actions like link navigation, form submission, or context menu display.
3. The method is invoked on the event object within the event handler.
4. It allows developers to implement custom behavior in place of the default.
5. Does not stop event propagation; it only blocks the default action associated with the event.

Example:

```
document.querySelector("a").addEventListener("click", function (event) {
  event.preventDefault();
  console.log("Default navigation prevented!");
});
```

var, let, and const:

47. What is the difference between var, let, and const in JavaScript?

Answer:

1. **Scope:**
   - var is function-scoped. It is visible within the function where it is declared and ignored in block scopes.
   - let and const are block-scoped, meaning they are limited to the block, statement, or expression in which they are declared.
2. **Re-declaration:**

   - var allows re-declaration within the same scope.
   - let does not allow re-declaration in the same scope.
   - const also does not allow re-declaration.
3. **Initialization:**
   - Variables declared with var are initialized as undefined if not explicitly assigned.
   - let and const do not initialize the variable until their declaration is executed (temporal dead zone).
4. **Mutability:**
   - Variables declared with var and let can be reassigned.
   - const variables cannot be reassigned, though their properties can be modified if they reference an object.

```
function testScope() {
  if (true) {
    var x = 10;
    let y = 20;
    const z = 30;
  }
  console.log(x); // Output: 10 (function-scoped)
  // console.log(y); // Error: y is not defined (block-scoped)
  // console.log(z); // Error: z is not defined (block-scoped)
}
testScope();
```

48. What is the Temporal Dead Zone (TDZ) in JavaScript?

Answer:

1. The Temporal Dead Zone (TDZ) is the time between the start of a block and when a variable declared with let or const is initialized.
2. During this period, accessing the variable results in a ReferenceError.
3. Unlike var, variables declared with let or const are not initialized as undefined at the start of execution.
4. The TDZ applies only to variables declared with let or const, not var.
5. Ensures safer coding practices by preventing accidental use of variables before their declaration.

```
console.log(x); // Output: undefined (var is hoisted)
console.log(y); // Error: Cannot access 'y' before initialization
var x = 10;
let y = 20;
```

49. Why should var be avoided in modern JavaScript?

Answer:

1. **Function Scope:** var is function-scoped, which can lead to unintended behavior when used inside block statements.

2. **Hoisting Issues:** Variables declared with var are hoisted and initialized as undefined, leading to potential bugs.
3. **Re-declaration:** var allows re-declaration, which can overwrite existing variables and cause confusion.

```
if (true) {
  var x = 10;
}
console.log(x); // Output: 10 (unexpected exposure outside block)

if (true) {
  let y = 20;
}
// console.log(y); // Error: y is not defined (block-scoped)
```

50. How does hoisting differ for var, let, and const?

Answer:

1. **var:** Variables declared with var are hoisted to the top of their scope and initialized as undefined. This can lead to unexpected results.
2. **let and const:** These are hoisted but remain uninitialized until their declaration is executed (TDZ).

```
console.log(a); // Output: undefined (var is hoisted)
var a = 10;

console.log(b); // Error: Cannot access 'b' before initialization
let b = 20;

console.log(c); // Error: Cannot access 'c' before initialization
const c = 30;
```

Understanding hoisting helps avoid runtime errors and unexpected behavior.

51. How do block scope and function scope affect variable visibility?

Answer:

- **Block Scope (let and const):** Variables declared with let or const are limited to the block in which they are defined. They are not accessible outside the block.
- **Function Scope (var):** Variables declared with var are accessible throughout the function in which they are defined, even if declared inside a block.

```
if (true) {
  var a = 10;
  let b = 20;
  const c = 30;
}
```

```
console.log(a); // Output: 10 (function-scoped)
// console.log(b); // Error: b is not defined (block-scoped)
// console.log(c); // Error: c is not defined (block-scoped)
```

Using block-scoped variables (let and const) helps avoid polluting the outer scope.

Spread and Rest:

52. What is the spread operator in JavaScript, and how is it used?

Answer:
The spread operator (...) is a syntax introduced in ES6 that allows you to spread the elements of an array, object, or iterable into individual elements. It is commonly used for cloning, merging, or passing arguments to functions.

Examples:

In Arrays:

```
const arr = [1, 2, 3];
const newArr = [...arr, 4, 5];
console.log(newArr); // Output: [1, 2, 3, 4, 5]
```

In Objects:

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

In Functions:

```
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // Output: 3
```

53. How does the spread operator differ from the rest operator?

Answer:
The spread operator (...) is used to **unpack elements** from arrays or objects, while the rest operator (...) is used to **pack elements** into an array or object.

Example of Spread:

```
const arr = [1, 2, 3];
const newArr = [...arr, 4, 5]; // Spreads elements into a new array
console.log(newArr); // Output: [1, 2, 3, 4, 5]
```

Example of Rest:

```
function sum(...numbers) {
  // Packs arguments into an array
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3)); // Output: 6
```

The spread operator extracts elements, whereas the rest operator gathers them.

54. How do you clone an array or object using the spread operator?

Answer:
The spread operator can be used to create a shallow copy of arrays or objects. This means the copied array or object shares references with the original for nested structures.

Cloning an Array:

```
const arr = [1, 2, 3];
const clonedArr = [...arr];
clonedArr.push(4);
console.log(arr); // Output: [1, 2, 3]
console.log(clonedArr); // Output: [1, 2, 3, 4]
```

Cloning an Object:

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

Use this approach for shallow copies, but for deeply nested objects, a deep cloning method is required.

55. What are the limitations of the spread operator?

Answer:

Shallow Copy Only: The spread operator creates shallow copies of objects and arrays. Nested structures will still share references with the original.

```
const obj = { a: { b: 1 } };
const cloned = { ...obj };
cloned.a.b = 2;
console.log(obj.a.b); // Output: 2 (shared reference)
```

Not Applicable to Non-Iterables: The spread operator only works with iterables (e.g., arrays, strings, sets). Applying it to non-iterable objects throws an error.

```
const obj = 123;
// const result = [...obj]; // Error: obj is not iterable
```

To handle deep copies, use libraries like lodash or structuredClone.

56. How can you merge arrays or objects using the spread operator?

Answer:
The spread operator provides a simple way to merge arrays and objects by spreading their elements or properties into a new structure.

Merging Arrays:

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

Merging Objects:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // Output: { a: 1, b: 3, c: 4 }
```

In objects, later properties overwrite earlier ones if there are conflicts.

Destructuring:

57. What is the difference between array destructuring and object destructuring?

Answer:

Array Destructuring: The order of elements matters, as values are assigned based on their positions in the array.

```
const arr = [1, 2, 3];
const [a, b] = arr;
console.log(a, b); // Output: 1, 2
```

Object Destructuring: The order does not matter, as values are matched by property names.

```
const obj = { name: "Mahesh", age: 25 };
const { age, name } = obj;
console.log(name, age); // Output: Mahesh, 25
```

Array destructuring is positional, while object destructuring is based on property names.

58. How do you use default values with destructuring?

Answer:
You can assign default values to variables during destructuring to handle undefined or missing values. If the property or element is not found, the default value is used instead.

Example:

```
// Array Destructuring with Defaults
const arr = [1];
const [a, b = 2] = arr;
console.log(a, b); // Output: 1, 2

// Object Destructuring with Defaults
const obj = { name: "Mahesh" };
const { name, age = 25 } = obj;
console.log(name, age); // Output: Mahesh, 25
```

Default values ensure robustness when working with incomplete data structures.

59. How do you combine rest and destructuring?

Answer:
You can use the rest operator (...) with destructuring to collect the remaining properties or elements into a separate variable.

Example with Objects:

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

Example with Arrays:

```
const arr = [1, 2, 3, 4];
const [first, ...rest] = arr;
console.log(first); // Output: 1
console.log(rest); // Output: [2, 3, 4]
```

The rest operator gathers remaining values into a new object or array.

60. How can destructuring be used with default function arguments?

Answer:
You can provide default values for function arguments during destructuring. This is useful for handling optional arguments in functions.

Example:

```
function greet({ name = "Guest", age = 0 } = {}) {
  return `Hello, ${name}. You are ${age} years old.`;
}
console.log(greet({ name: "Mahesh" })); // Output: Hello, Mahesh. You are 0 years old.
console.log(greet()); // Output: Hello, Guest. You are 0 years old.
```

Providing defaults ensures functions behave correctly even with missing or incomplete arguments.

Prototype:

61. What is the difference between __proto__ and prototype?

Answer:

1. **__proto__:**
   - A property of an object that points to its prototype.
   - Used for accessing the prototype chain dynamically.
2. **prototype:**
   - A property of constructor functions that is used to define properties and methods for instances created by the constructor.
   - It's not directly accessible on instances but determines the __proto__ of the instances.

Example:

```
function Person() {}
const mahesh = new Person();

console.log(Person.prototype); // Output: Prototype object of Person
console.log(mahesh.__proto__); // Output: Same as Person.prototype
```

The __proto__ links the instance to its constructor's prototype.

62. How can you create objects without a prototype?

Answer:
You can create objects without a prototype using Object.create(null). This is useful when you need a truly plain object without inherited properties.

```
const obj = Object.create(null);
console.log(obj); // Output: {}
console.log(obj.toString); // Output: undefined
```

Here, obj has no prototype, so it doesn't inherit methods like toString or hasOwnProperty.

63. What is the difference between prototypal inheritance and classical inheritance?

Answer:

- **Prototypal Inheritance:** Objects inherit directly from other objects using prototypes. It is more flexible and simpler compared to classical inheritance.
- **Classical Inheritance:** Objects are instantiated from classes (constructors in JavaScript). This is more rigid and mimics inheritance in languages like Java or C++.

Example of Prototypal Inheritance:

```
const animal = { eat: () => "Eating" };
const dog = Object.create(animal);
console.log(dog.eat()); // Output: Eating
```

Example of Classical Inheritance:

```
class Animal {
  eat() {
    return "Eating";
  }
}
class Dog extends Animal {}
const dog = new Dog();
console.log(dog.eat()); // Output: Eating
```

Classes:

64. What is the difference between a class and a constructor function?

Answer:

1. **Syntax:** Classes are more concise and resemble traditional OOP languages. Constructor functions are less structured and use prototypes for inheritance.
2. **Hoisting:** Classes are not hoisted, meaning you cannot use a class before declaring it. Constructor functions are hoisted, but only their declaration.
3. **strict mode:** Classes always operate in strict mode, which helps prevent certain common bugs.
4. **Static Methods:** Classes have built-in support for static methods, while constructor functions require manual implementation.

Example of Constructor Function:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function () {
  return `Hello, ${this.name}`;
};
const mahesh = new Person("Mahesh");
console.log(mahesh.greet()); // Output: Hello, Mahesh
```

65. What is the purpose of the super keyword in classes?

Answer:
The super keyword is used to call the constructor or methods of a parent class. It is essential when a subclass needs to initialize properties defined in the parent class or when it wants to extend a method from the parent class.

Example:

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

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Calls the parent class constructor
    this.breed = breed;
  }
  getDetails() {
    return `${this.name} is a ${this.breed}`;
  }
}

const dog = new Dog("Buddy", "Labrador");
console.log(dog.getDetails()); // Output: Buddy is a Labrador
```

66. What is the difference between class and function constructors in terms of inheritance?

Answer:

1. **Syntax:** Classes use extends for inheritance, while functions rely on manually setting the prototype using Object.create() or Object.setPrototypeOf().
2. **super Keyword:** Classes use super for parent class constructor calls. Functions need explicit calls to the parent constructor.

3. **Code Readability:** Classes offer a cleaner and more intuitive syntax compared to prototype-based inheritance in functions.

Example with Class:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}
class Dog extends Animal {
  constructor(name) {
    super(name);
  }
}
```

Example with Function:

```
function Animal(name) {
  this.name = name;
}
function Dog(name) {
  Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
```

Arrow Function:

67. How do arrow functions differ from traditional functions in terms of this?

Answer:

1. **Arrow functions** do not have their own this context; they inherit this from the surrounding lexical scope.
2. **Traditional functions** have their own this, determined by how the function is called (runtime binding).
3. Arrow functions are particularly useful in callbacks or event listeners, where this might otherwise be lost.
4. Arrow functions do not bind their own this when used in methods or inside constructors.
5. Traditional functions are suitable when dynamic this binding is required.

Example:

```
function Traditional() {
  this.value = 10;
  setTimeout(function () {
    console.log(this.value); // Output: undefined (or throws an error in strict mode)
  }, 100);
}

function ArrowFunction() {
  this.value = 10;
  setTimeout(() => {
    console.log(this.value); // Output: 10 (inherits `this` from ArrowFunction)
  }, 100);
}

new Traditional();
new ArrowFunction();
```

68. Can arrow functions be used as constructors?

Answer:

1. No, arrow functions cannot be used as constructors.
2. They lack their own this context, which is essential for creating objects with the new keyword.
3. Arrow functions do not have a prototype property, making object instantiation impossible.
4. Attempting to use the new keyword with an arrow function will throw an error.
5. Regular functions or classes must be used when a constructor is required.

Example:

```
const ArrowConstructor = () => {};
// const obj = new ArrowConstructor(); // Error: ArrowConstructor is not a constructor
```

If a constructor is required, you must use a regular function or class.

69. Can arrow functions have default parameters?

Answer:

1. Yes, arrow functions support default parameters.
2. Default parameters allow you to assign a value to a parameter when no argument or undefined is provided.
3. The syntax for default parameters is the same as for traditional functions.
4. Helps make arrow functions cleaner and reduces the need for additional checks or conditions.
5. Default parameters can be combined with rest parameters or destructuring for enhanced functionality.

Example:

```
const greet = (name = "Guest") => `Hello, ${name}!`;
console.log(greet()); // Output: Hello, Guest!
console.log(greet("Mahesh")); // Output: Hello, Mahesh!
```

70. What are the limitations of arrow functions?

Answer:

1. **No this Context:** Arrow functions inherit this from the surrounding lexical scope, making them unsuitable for dynamic this contexts like object methods.
2. **No arguments Object:** Arrow functions do not have their own arguments. Use the rest operator (...args) instead.
3. **Cannot Be Used as Constructors:** Arrow functions cannot be used with the new keyword to create objects.
4. **No super Support:** Arrow functions cannot use super directly, which is crucial for extending classes.

Example of Limitation:

```
const obj = {
  value: 10,
  method: () => {
    console.log(this.value); // Output: undefined (not bound to `obj`)
  },
};
obj.method();
```

Closures:

71. What are the practical uses of closures?

Answer:

1. **Data Encapsulation:** Closures help create private variables, mimicking the behavior of private properties.

```
function Counter() {
  let count = 0;
  return {
    increment() {
      count++;
    },
    getCount() {
      return count;
    },
  };
}
const counter = Counter();
counter.increment();
console.log(counter.getCount()); // Output: 1
```

Function Factories: Closures can create functions with preset configurations.

```
function multiplier(factor) {
  return function (number) {
    return number * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // Output: 10
```

Event Handlers: Closures allow event handlers to retain state from their creation context.

72. How do closures work with asynchronous code?

Answer:
Closures retain their reference to variables even when the containing function has completed execution. This is useful in asynchronous operations like setTimeout or promises.

Example:

```
function delayedMessage(message, delay) {
  setTimeout(() => {
    console.log(message); // Accesses `message` from the outer scope
  }, delay);
}
delayedMessage("Hello after 1 second", 1000); // Output: Hello after 1 second
```

The closure ensures the message variable remains accessible to the callback even after delayedMessage has finished executing.

73. How are callbacks used in asynchronous programming?

Answer:
Callbacks are crucial in asynchronous programming, as they execute once the asynchronous operation completes, ensuring non-blocking code execution.

Example:

```
setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);
console.log("This runs first");
```

Output:

```
This runs first
Executed after 2 seconds
```

Here, the callback passed to setTimeout runs after the specified delay, while the main thread continues.

74. What are common issues with callbacks, and how can they be solved?

Answer:

1. **Callback Hell:** When multiple nested callbacks create deeply indented code, making it hard to read and maintain.

```
fetchData((data) => {
  processData(data, (result) => {
    saveData(result, (status) => {
      console.log("Data saved!");
    });
  });
});
```

Solution: Use Promises or async/await to manage asynchronous code.

Error Handling: Errors in callbacks can go unnoticed if not explicitly handled.

```
function asyncOperation(callback) {
  try {
    // Simulating an error
    throw new Error("Something went wrong");
  } catch (error) {
    callback(error);
  }
}
asyncOperation((err) => {
  if (err) console.error(err.message); // Output: Something went wrong
});
```

75. What is the difference between callbacks and Promises?

Answer:

1. **Readability:** Promises simplify chaining and improve code readability compared to nested callbacks.
2. **Error Handling:** Promises handle errors using .catch() or try-catch in async/await.
3. **Multiple Handlers:** Promises can have multiple .then() handlers, while callbacks are executed once.

Example with Promises:

```

fetchData()
  .then(processData)
  .then(saveData)
  .then(() => console.log("Data saved!"))
```

```
 .catch((error) => console.error(error));
```

Promises reduce the complexity of managing asynchronous operations compared to callbacks.

HOF => Map, Filter, Reduce:

76. What are the advantages of using Higher-Order Functions?

Answer:

1. **Code Reusability:** HOFs allow you to define logic in one place and reuse it with different callbacks.
2. **Abstraction:** They help abstract repetitive tasks, like iteration or filtering, into reusable functions.
3. **Improved Readability:** They simplify code by removing boilerplate, especially when working with arrays or asynchronous tasks.

Example of Abstraction:

```
function processArray(arr, callback) {
  return arr.map(callback);
}

const numbers = [1, 2, 3];
console.log(processArray(numbers, num => num * 2)); // Output: [2, 4, 6]
```

77. How do Higher-Order Functions work with Promises?

Answer:
HOFs like .then() and .catch() in Promises take callback functions as arguments to handle asynchronous tasks. This allows chaining operations and handling errors cleanly.

Example:

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

Here, .then() and .catch() are Higher-Order Functions that accept callbacks for success and error handling.

78. How does map() handle empty slots in arrays?

Answer:
If an array contains empty slots (e.g., [1, , 3]), the map() method skips these slots but maintains their positions in the new array.

Example:

```
const arr = [1, , 3];
const result = arr.map((x) => x || 0); // Replace empty slots with 0
console.log(result); // Output: [1, 0, 3]
```

Empty slots are ignored by map() unless explicitly handled in the callback.

79. What happens if filter() does not find any matching elements?

Answer:
If no elements match the condition, the filter() method returns an empty array.

Example:

```
const numbers = [1, 2, 3];
const greaterThanTen = numbers.filter((num) => num > 10);
console.log(greaterThanTen); // Output: []
```

80. How do the initialValue and accumulator work in reduce()?

Answer:

- **initialValue**: The starting value for the accumulator. If not provided, the first element of the array is used as the initial value.
- **accumulator**: Holds the cumulative result of the callback function across iterations.

Example:

```
const numbers = [1, 2, 3, 4];

// Without initialValue
const sum1 = numbers.reduce((acc, curr) => acc + curr);
console.log(sum1); // Output: 10

// With initialValue
const sum2 = numbers.reduce((acc, curr) => acc + curr, 5);
console.log(sum2); // Output: 15
```

81. How is the map() function an example of a Higher-Order Function?

Answer:
The map() function is a built-in HOF that applies a callback function to every element of an

array and returns a new array with the results. It demonstrates how HOFs abstract behavior by allowing custom logic to be passed as a function.

Example:

```
const numbers = [1, 2, 3, 4];
const squared = numbers.map((num) => num * num);
console.log(squared); // Output: [1, 4, 9, 16]
```

Here, the map() function takes the callback (num => num * num) as an argument to apply the squaring logic to each element.

82. How does filter() work as a Higher-Order Function?

Answer:
The filter() function takes a callback that returns a boolean value. It creates a new array containing only the elements that pass the test defined in the callback.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter((num) => num % 2 === 0);
console.log(evens); // Output: [2, 4]
```

In this example, filter() uses the callback (num => num % 2 === 0) to decide which numbers to include in the result.

83. How is reduce() an example of a Higher-Order Function?

Answer:
The reduce() method takes a callback function and an optional initial value. It applies the callback to each element of the array, accumulating a single output value.

Example:

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

Here, reduce() is a HOF because it takes a callback (acc, curr) => acc + curr to define the accumulation logic.

Debouncing or Throttling:

84. What is debouncing in JavaScript?

Answer:

1. **Debouncing** is a technique to limit the number of times a function is executed by ensuring it is only invoked after a specified delay.
2. It is used to improve performance by reducing the frequency of event handling in scenarios like window resizing, button clicks, or keystrokes.
3. Prevents a function from being called repeatedly in rapid succession, such as during continuous user interactions.
4. Involves resetting a timer (setTimeout) every time the event is triggered, delaying the function execution until no events occur during the delay period.
5. Commonly implemented in scenarios where repeated executions of a function can degrade performance, such as handling scroll or resize events.

Example:

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

// Example Usage
const log = () => console.log("Debounced function executed");
const debouncedLog = debounce(log, 1000);

window.addEventListener("resize", debouncedLog);
```

Here, the log function is called only after the user stops resizing the window for 1 second.

85. What are the practical use cases of debouncing?

Answer:

1. **Search Bar Suggestions:** Sending API requests only after the user has stopped typing.
2. **Window Resize Events:** Avoiding performance-intensive operations when the window is continuously resized.
3. **Form Validations:** Validating input fields after the user finishes typing.
4. **Scroll Events:** Preventing frequent DOM updates while scrolling.

Example:

```
const search = (query) => console.log(`Searching for ${query}`);
const debouncedSearch = debounce(search, 300);
```

```
document.getElementById("searchInput").addEventListener("input", (e) => {
  debouncedSearch(e.target.value);
});
```

This example debounces the search function to avoid sending too many API requests.

86. What is throttling in JavaScript?

Answer:

1. **Throttling** ensures that a function is executed at most once in a specified time interval, regardless of how many times it is triggered.
2. It controls the rate of function execution, making it useful for events that fire frequently, such as scrolling, resizing, or mouse movements.
3. Improves performance by reducing the number of function calls during high-frequency events.
4. A timer is used to control the execution frequency of the function within the specified interval.
5. Throttling is ideal when consistent periodic updates are required, such as updating UI elements during scrolling.

Example:

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

// Example Usage
const log = () => console.log("Throttled function executed");
const throttledLog = throttle(log, 1000);

window.addEventListener("scroll", throttledLog);
```

Here, the log function is called at most once every second while scrolling.

87. What are the practical use cases of throttling?

Answer:

1. **Scroll Events:** Reducing the frequency of updates to UI elements during scrolling.
2. **API Rate Limiting:** Preventing excessive API calls within a short time frame.
3. **Resize Events:** Adjusting layouts or dimensions while resizing the window.
4. **Button Clicks:** Preventing multiple clicks on a button in quick succession.

Example:

```
const handleScroll = () => console.log("Scrolled!");
const throttledScroll = throttle(handleScroll, 500);

window.addEventListener("scroll", throttledScroll);
```

This ensures that the handleScroll function executes at most once every 500ms during scrolling.

88. What is the difference between debouncing and throttling?

Answer:

Feature	Debouncing	Throttling
Purpose	Delays function execution until after a delay following the last event.	Limits the function execution rate to at most once in a given interval.
Execution	Executes the function once, after the event stops firing.	Executes the function at regular intervals while the event keeps firing.
Use Cases	Search bars, input fields, form validations.	Scroll events, resize events, API rate limiting.

Example Comparison:

```
// Debounce Example
const debouncedResize = debounce(() => console.log("Debounced Resize!"), 300);
window.addEventListener("resize", debouncedResize);

// Throttle Example
const throttledResize = throttle(() => console.log("Throttled Resize!"), 300);
window.addEventListener("resize", throttledResize);
```

89. Can you implement a combination of debouncing and throttling?

Answer:
Yes, you can combine debouncing and throttling to achieve a hybrid solution. For example, you may want to throttle events but ensure the function is called one final time after the events stop firing.

Example:

```
function debounceThrottle(func, delay, limit) {
  let lastCall = 0;
  let timeout;
  return function (...args) {
    const now = Date.now();
    clearTimeout(timeout);

    if (now - lastCall >= limit) {
      lastCall = now;
      func.apply(this, args);
    }

    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}

// Example Usage
const log = () => console.log("Debounce + Throttle function executed");
const debouncedThrottledLog = debounceThrottle(log, 1000, 500);

window.addEventListener("resize", debouncedThrottledLog);
```

This ensures the log function executes at most once every 500ms and one final time after 1 second of inactivity.

90. Which one should you choose, debouncing or throttling?

Answer:

- **Choose Debouncing:** When you need to wait for the user to finish an action (e.g., typing in a search bar or resizing a window).
- **Choose Throttling:** When you need consistent execution over time, even while an action is continuously occurring (e.g., updating scroll positions or limiting API calls).

The choice depends on the specific requirements of your application.

Modules:

91. What is the difference between named exports and default exports?

Answer:

1. **Named Exports:** Allow you to export multiple values from a module using their names. These must be imported using the exact same names. **Example:**

```
// Exporting
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

```
// Importing
import { add, subtract } from "./math.js";
```

2. **Default Exports:** Allow you to export a single value or function. They can be imported with any name. **Example:**

```
// Exporting
export default (a, b) => a + b;

// Importing
import sum from "./math.js";
console.log(sum(2, 3)); // Output: 5
```

You can use both named and default exports in the same file.

92. What is the difference between ES Modules and CommonJS?

Answer:

Feature	ES Modules (ESM)	CommonJS (CJS)
Syntax	Uses import and export	Uses require and module.exports
Support	Native support in modern browsers and Node.js (from v12)	Primarily used in Node.js
Static/Dynamic	Statically analyzed (tree-shaking supported)	Dynamically loaded
Default Export	export default	module.exports =

Example of CommonJS:

```
// Exporting
module.exports = { greet };

// Importing
const { greet } = require("./module.js");
```

93. What is the purpose of the export * from syntax?

Answer:
The export * from syntax re-exports all exports from another module. It is commonly used to create aggregated modules that consolidate exports from multiple modules.

Example:

```
// moduleA.js
export const a = 1;
```

```
// moduleB.js
export const b = 2;

// index.js
export * from "./moduleA.js";
export * from "./moduleB.js";

// main.js
import { a, b } from "./index.js";
console.log(a, b); // Output: 1, 2
```

This approach simplifies importing multiple related modules.

94. What are the advantages of using modules?

Answer:

1. **Code Reusability:** Modules allow you to write reusable, modular pieces of code.
2. **Encapsulation:** Variables and functions are scoped to the module, avoiding global namespace pollution.
3. **Maintainability:** Code is easier to maintain and debug when broken into smaller parts.
4. **Lazy Loading:** Modules can be dynamically imported to reduce the initial load time of applications.

```
// Lazy loading a module
document.getElementById("button").addEventListener("click", async () => {
  const { greet } = await import("./module.js");
  console.log(greet("Bob"));
});
```

95. What are cookies in JavaScript?

Answer:

1. Cookies are small pieces of data stored on the user's browser by a website.
2. They are used to store information such as session data, user preferences, or tracking data.
3. Cookies are automatically sent back and forth between the client and server with every HTTP request.
4. They are essential for maintaining state in stateless HTTP connections.
5. Cookies can be set, accessed, or modified using JavaScript through the `document.cookie` property.

Example:

```
document.cookie = "username=Mahesh";
console.log(document.cookie); // Output: username=Mahesh
```

96. How do you set a cookie in JavaScript?

Answer:
You can set a cookie using the document.cookie property. Each cookie is added as a string in the format name=value; followed by optional attributes like expires or path.

Example:

```
// Setting a cookie
document.cookie =
  "username=Mahesh; expires=Fri, 31 Jan 2025 23:59:59 GMT; path=/; secure";
```

97. What are the differences between session cookies and persistent cookies?

Answer:

1. **Session Cookies:**
   - Do not have an expiration date or max-age specified.
   - Stored in memory and are deleted when the browser is closed.
2. **Persistent Cookies:**
   - Have an expires or max-age attribute specified.
   - Stored on the user's disk and persist across browser sessions until they expire or are deleted.

Example of Persistent Cookie:

```
document.cookie = "theme=dark; max-age=86400"; // Expires in 1 day
```

local storage and session storage:

98. What is the difference between local storage and session storage?

Answer:
Local storage and session storage are part of the Web Storage API and allow storing key-value pairs in the browser. The key differences are:

1. **Lifetime:**
   - **Local Storage:** Data persists indefinitely until explicitly cleared.
   - **Session Storage:** Data is cleared when the page session ends (e.g., when the tab or browser is closed).
2. **Scope:**
   - **Local Storage:** Shared across all tabs/windows from the same origin.
   - **Session Storage:** Limited to the specific tab/window.

Example:

```
// Local Storage
localStorage.setItem("username", "Mahesh");
console.log(localStorage.getItem("username")); // Output: Mahesh

// Session Storage
```

```
sessionStorage.setItem("sessionID", "12345");
console.log(sessionStorage.getItem("sessionID")); // Output: 12345
```

99. How do cookies differ from local storage and session storage?

Answer:

Feature	Cookies	Local Storage	Session Storage
Storage	4KB per cookie	5MB per origin	5MB per origin
Lifetime	Depends on expires or max-age	Persistent until manually cleared	Cleared when the browser is closed
Scope	Sent with HTTP requests	Accessible only via JavaScript	Accessible only via JavaScript
Security	Vulnerable without HttpOnly	More secure, no automatic transfer	More secure, no automatic transfer

Cookies are best suited for server-client communication, while local and session storage are ideal for storing client-side data.

100. How do you store complex objects in local storage?

Answer:
Since local storage only stores strings, you need to serialize objects into JSON strings using JSON.stringify and deserialize them using JSON.parse.

Example:

```
const user = { name: "Mahesh", age: 25 };

// Storing an object
localStorage.setItem("user", JSON.stringify(user));

// Retrieving an object
const storedUser = JSON.parse(localStorage.getItem("user"));
console.log(storedUser.name); // Output: Mahesh
```

101. What are the limitations of local storage?

Answer:

1. **Storage Limit:** Local storage has a limit of about **5MB** per origin in most browsers.
2. **String-Only Data:** Only strings can be stored, requiring serialization for non-string data.
3. **No Expiration:** Data persists indefinitely unless explicitly cleared.
4. **Security Risks:** Data is accessible to JavaScript, making it vulnerable to cross-site scripting (XSS) attacks.

102. How do you check the available storage capacity?

Answer:
There is no direct method to check the available capacity of local or session storage, but you can calculate it by repeatedly adding data until an error occurs.

Example:

```
function checkStorageLimit(storage) {
  let data = "x";
  try {
    while (true) {
      storage.setItem("test", data);
      data += data; // Exponentially increase data size
    }
  } catch (e) {
    console.log(`Approximate storage limit: ${data.length / 1024} KB`);
    storage.removeItem("test");
  }
}

checkStorageLimit(localStorage);
```

103. When should you use local storage and session storage?

Answer:

1. **Use Local Storage:**
   - When the data needs to persist across browser sessions.
   - For non-sensitive data such as theme preferences, application settings, or caching small amounts of data.
2. **Use Session Storage:**
   - For temporary data that is only needed during the user's session (e.g., form inputs, tab-specific states).

Example:

```
// Local Storage for theme preferences
localStorage.setItem("theme", "dark");

// Session Storage for temporary form data
sessionStorage.setItem(
  "draft",
  JSON.stringify({ title: "My Post", content: "..." })
);
```

Fetch API:

104. How does the Fetch API handle HTTP errors?

Answer:
The Fetch API only rejects a promise if a **network error** occurs. It does not reject for HTTP errors like 404 or 500. These must be handled manually by checking the response.ok property or the response.status code.

Example:

```
fetch("https://api.example.com/data")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP Error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

105. How can you include query parameters in a fetch request?

Answer:
Query parameters can be added directly to the URL by appending a query string.

Example:

```
const params = new URLSearchParams({
  search: "JavaScript",
  page: 1,
});

fetch(`https://api.example.com/data?${params}`)
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

Here, URLSearchParams constructs the query string from an object

106. How do you handle JSON and non-JSON responses with the Fetch API?

Answer:
You can use the Content-Type header in the response to determine how to handle the data. For example, you might process JSON responses with .json() and plain text responses with .text().

Example:

```
fetch("https://api.example.com/data")
  .then((response) => {
    const contentType = response.headers.get("Content-Type");
    if (contentType.includes("application/json")) {
      return response.json();
    } else if (contentType.includes("text/plain")) {
      return response.text();
    } else {
      throw new Error("Unsupported content type");
    }
  })
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

107. How can you use async/await with the Fetch API?

Answer:
You can use async/await to simplify asynchronous fetch operations and handle errors using try-catch.

Example:

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) {
      throw new Error(`HTTP Error! Status: ${response.status}`);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}

fetchData();
```

108. How do you include headers in a fetch request?

Answer:
You can include headers by specifying them in the headers property of the options object. Common headers include Authorization for tokens and Content-Type for specifying the request body format.

Example:

```
fetch("https://api.example.com/data", {
  method: "GET",
  headers: {
    Authorization: "Bearer your-token",
    "Content-Type": "application/json",
  },
})
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

JavaScript 60+ Output Based Questions

Introduction of JavaScript, Datatypes and operators:

1. What will be the output of the following JavaScript code?

```
console.log(typeof foo);
var foo = function () {
  console.log("Hello, World!");
};
console.log(typeof foo);
```

Output:

```
undefined
function
```

Explanation:

- Before the function is assigned to foo, foo is declared but doesn't have a value—it’s undefined.
- The first console.log checks the type of foo at this point, so it outputs "undefined".
- After assigning the function to foo, its type becomes "function".
- The second console.log reflects this change, outputting "function".

2. What will be the output of the following JavaScript code?

```
console.log([] + []);
console.log([] + {});
console.log({} + []);
console.log({} + {});
console.log(+true + "1" + false);
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

Output:

```
"";
"[object Object]";
0;
("[object Object][object Object]");
("11false");
true;
false;
```

Explanation:

1. **[] + [] → ""**
   - Arrays are converted to strings. An empty array becomes an empty string "".
   - "" + "" results in "" (an empty string).

---

2. **[] + {} → "[object Object]"**
   - The empty array [] converts to "" (empty string).
   - The object {} converts to its string representation: "[object Object]".
   - Concatenation: "" + "[object Object]" gives "[object Object]".

---

3. **{} + [] → 0**
   - {} at the start is treated as a block (not an object).
   - + [] coerces [] into an empty string "", and + "" converts it to 0.
   - Result: 0.

---

4. **{} + {} → "[object Object][object Object]"**
   - Both {} are treated as objects.
   - Objects convert to strings: "[object Object]".
   - Concatenation: "[object Object]" + "[object Object]" results in "[object Object][object Object]".

---

5. **+true + "1" + false → "11false"**
   - +true converts true into 1 (using unary +).
   - 1 + "1" results in "11" (number 1 coerces to string).
   - "11" + false results in "11false" (false coerces to string).

---

6. **1 < 2 < 3 → true**
   - JavaScript evaluates left-to-right.
   - 1 < 2 is true.
   - true is coerced to 1 in 1 < 3, which evaluates to true.

---

7. **3 > 2 > 1 → false**
   - Left-to-right evaluation.
   - 3 > 2 is true.
   - true is coerced to 1 in 1 > 1, which evaluates to false.

3. What will be the output of the following JavaScript code ?

```
console.log(+"123");
console.log(+null);
console.log(+undefined);
console.log(+[]);
console.log(+{});
```

Output:

```
123;
0;
NaN;
0;
NaN;
```

Explanation:

- +"123": Converts the string "123" to a number → 123.
- +null: Null coerces to 0 → 0.
- +undefined: Undefined cannot be converted to a number → NaN.
- +[]: An empty array coerces to an empty string, which converts to 0 → 0.
- +{}: Objects cannot be directly converted to numbers → NaN.

4. What will be the output of the following JavaScript code ?

```
console.log("5" + 3 - 2);
console.log("5" - "3" + "2");
console.log("10" - "5" + 2 + "px");
```

Output:

```
51
22
7px
```

Explanation:

1. "5" + 3 - 2: "5" + 3 concatenates to "53". "53" - 2 coerces "53" to 53 → 51.
2. "5" - "3" + "2": "5" - "3" becomes 2. 2 + "2" concatenates to "22".
3. "10" - "5" + 2 + "px": "10" - "5" becomes 5. 5 + 2 is 7. 7 + "px" concatenates to "7px".

5. What will be the output of the following JavaScript code ?

```
console.log(null == 0);
console.log(null >= 0);
console.log(null + []);
console.log(true + "1" == "true1");
```

Output:

```
false;
true;
("null");
false;
```

Explanation:

1. null == 0: Null only equals undefined (not numbers) → false.
2. null >= 0: Comparison converts null to 0, so 0 >= 0 → true.
3. null + []: null coerces to "null", and + [] to "". "null" + "" → "null".
4. true + "1" == "true1": true + "1" results in "true1". Comparing with "true1" → false.

6. What will be the output of the following JavaScript code ?

```
console.log([] == ![]);
console.log([1, 2] + [3, 4]);
console.log({} == {});
console.log("0" == false);
```

Output:

```
true;
("1,23,4");
false;
true;
```

Explanation:

1. [] == ![]: ![] is false, and [] == false (both coerced to "") → true.
2. [1, 2] + [3, 4]: Arrays are coerced to strings. "1,2" + "3,4" → "1,23,4".
3. {} == {}: Objects are compared by reference, not value → false.
4. "0" == false: false coerces to 0, and "0" coerces to 0 → true.

7. What will be the output of the following JavaScript code ?

```
console.log(typeof null);
console.log(null instanceof Object);
console.log(typeof NaN);
console.log(isNaN("abc" - 3));
```

Output:

```
object;
false;
number;
true;
```

Explanation:

1. typeof null: Legacy behavior; null is treated as object.
2. null instanceof Object: null is not an instance of Object → false.
3. typeof NaN: NaN is a special value of type number → number.
4. isNaN("abc" - 3): "abc" - 3 results in NaN, and isNaN(NaN) → true.

8. What will be the output of the following JavaScript code ?

```
console.log(null == undefined);
console.log(null === undefined);
console.log(!!null);
console.log(Boolean({}));
```

Output:

```
true;
false;
false;
true;
```

Explanation:

1. null == undefined: Loose equality considers null and undefined as equal → true.
2. null === undefined: Strict equality requires matching types; null and undefined are different → false.
3. !!null: null is falsy, so !!null is false.
4. Boolean({}): Non-empty objects are always truthy → true.

9. What will be the output of the following JavaScript code ?

```
console.log(typeof 42n);
console.log(10n + 20n);
console.log(10n === 10);
console.log(typeof Symbol("id"));
```

Output:

```
bigint;
30n;
false;
symbol;
```

Explanation:

1. typeof 42n: The n suffix denotes a BigInt, and its type is "bigint".
2. 10n + 20n: Arithmetic operations on BigInt values result in a BigInt → 30n.
3. 10n === 10: BigInt and Number are different types → false.
4. typeof Symbol("id"): Symbols are unique identifiers with type "symbol".

10. What will be the output of the following JavaScript code ?

```
console.log(3 || 0 && 2);
console.log(0 && 3 || 2);
console.log(0 ?? 5);
console.log(undefined ?? 5);
console.log(null ?? 0 ?? 10);
```

Output:

```
3;
2;
5;
5;
0;
```

Explanation:

1. 3 || 0 && 2: && first. 0 && 2 → 0. Then 3 || 0 → 3.
2. 0 && 3 || 2: && first. 0 && 3 → 0. Then 0 || 2 → 2.
3. 0 ?? 5: Nullish coalescing returns the right-hand operand if the left is null or undefined. 0 is not nullish → 0.
4. undefined ?? 5: Left operand is undefined. Result → 5.
5. null ?? 0 ?? 10: null ?? 0 → 0, then 0 ?? 10 → 0.

conditional statements:

11. What will be the output of the following JavaScript code ?

```
let x = 0;
if ((x = 5)) {
  console.log("x is truthy");
} else {
  console.log("x is falsy");
}
```

Output:

```
x is truthy
```

Explanation:

- x = 5 is an assignment, not a comparison.
- The condition evaluates to 5, which is truthy, so "x is truthy" is logged.

12. What will be the output of the following JavaScript code ?

```
let c = 0;
let d = null;
if (c ?? d) {
  console.log("c or d is not nullish");
} else {
  console.log("Both are nullish");
}
```

Output:

```
Both are nullish
```

Explanation:

- c ?? d evaluates to c since c is not null or undefined.
- c is 0, which is falsy, so the else block executes.

switch and loops:

13. What will be the output of the following JavaScript code ?

```
const num = 10;
switch (true) {
  case num < 5:
    console.log("Less than 5");
  case num < 15:
    console.log("Less than 15");
  default:
    console.log("Default case");
}
```

Output:

```
Less than 15
Default case
```

Explanation:

- The first case fails, but num < 15 evaluates to true.
- Without a break, it falls through to the default case.

14. What will be the output of the following JavaScript code ?

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) break outer;
    console.log(i, j);
  }
}
```

Output:

```
0 0
0 1
0 2
1 0
```

```
1 1
```

Explanation:

- The break outer exits the outer loop when i === 1 and j === 1.
- All iterations before this condition are logged.

15. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3];
arr[5] = 10;
console.log(arr);
console.log(arr.length);
```

Output:

```
[1, 2, 3, <2 empty items>, 10]
6
```

Explanation:

- Assigning arr[5] = 10 creates a sparse array with two empty slots.
- The length of the array becomes 6 due to the highest index (5) + 1.

16. What will be the output of the following JavaScript code ?

```
const arr = [1, [2, [3, [4, [5]]]]];
const result = arr.flat(Infinity);
console.log(result);
```

Output:

```
[1, 2, 3, 4, 5];
```

Explanation:

- flat(Infinity) flattens the array completely, regardless of depth, into a single-level array.

17. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3];
const mapResult = arr.map((x) => x * 2);
const forEachResult = arr.forEach((x) => x * 2);
console.log(mapResult);
console.log(forEachResult);
```

Output:

```
[2, 4, 6];
undefined;
```

Explanation:

- map creates a new array by applying the callback function to each element.
- forEach does not return a new array; it only executes the callback for each element and returns undefined.

18. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3, 4, 5];
const result = arr
  .filter((x) => x % 2 === 0)
  .map((x) => x * 2)
  .reduce((acc, curr) => acc + curr, 0);
console.log(result);
```

Output:

```
12;
```

Explanation:

- filter(x => x % 2 === 0): Keeps even numbers → [2, 4].
- map(x => x * 2): Doubles each value → [4, 8].
- reduce((acc, curr) => acc + curr, 0): Sums the values → 4 + 8 = 12.

19. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3, 4];
arr.splice(1, 0, "a", "b");
console.log(arr);
arr.splice(-2, 1, "x");
console.log(arr);
```

Output:

```
[1, "a", "b", 2, 3, 4]
[1, "a", "b", 2, "x", 4]
```

Explanation:

1. splice(1, 0, "a", "b"): Adds "a" and "b" at index 1 without removing anything.
2. splice(-2, 1, "x"): At position -2 (second to last), removes 1 element (3) and inserts "x".

20. What will be the output of the following JavaScript code ?

```
const arr = [10, 5, 20, 15];
arr.sort((a, b) => b - a);
console.log(arr);
```

Output:

```
[20, 15, 10, 5];
```

Explanation:

- The custom comparator sorts the array in descending order:
    - b - a ensures higher numbers come before lower numbers.

21. What will be the output of the following JavaScript code ?

```
const arr = [1, , 3];
console.log(arr.map((x) => x || 0));
console.log(arr.filter((x) => x !== undefined));
console.log(arr.length);
```

Output:

```
[1, 0, 3]
[1, 3]
3
```

Explanation:

1. map(x => x || 0): Maps empty slots to 0.
2. filter(x => x !== undefined): Excludes undefined values (skips empty slots).
3. .length: Includes empty slots, so the length is 3.

22. What will be the output of the following JavaScript code ?

```
const arr = [10, 20, 30];
console.log(arr.some((x) => x > 25));
console.log(arr.every((x) => x > 25));
```

Output:

```
true;
false;
```

Explanation:

1. some(x => x > 25): Returns true if at least one element satisfies the condition (30 > 25).
2. every(x => x > 25): Returns false because not all elements satisfy the condition.

23. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3];
const reversed = arr.reverse();
console.log(reversed);
console.log(arr);
```

Output:

```
[3, 2, 1]
[3, 2, 1];
```

Explanation:

- reverse reverses the array **in place** and also returns the reversed array.
- Both reversed and arr reference the same mutated array.

24. What will be the output of the following JavaScript code ?

```
const arr1 = Array.from("hello");
const arr2 = Array.of(1, 2, 3);
console.log(arr1);
console.log(arr2);
```

Output:

```
["h", "e", "l", "l", "o"]
[1, 2, 3]
```

Explanation:

1. Array.from("hello"): Creates an array from the string by splitting each character.
2. Array.of(1, 2, 3): Creates an array from the given arguments.

25. What will be the output of the following JavaScript code ?

```
const arr = [1, -2, 3, -4];
const index = arr.findIndex((x) => x < 0);
console.log(index);
console.log(arr[index]);
```

Output:

```
1
-2
```

Explanation:

1. findIndex(x => x < 0): Returns the index of the first element less than 0 → 1.
2. arr[index]: Accesses the element at index 1 → -2.

26. What will be the output of the following JavaScript code ?

```
function testArgs(a, b) {
  console.log(arguments[0]);
  console.log(arguments[1]);
}
testArgs(10, 20);
```

Output:

```
10;
20;
```

Explanation:

- The arguments object holds all arguments passed to the function, even if parameters are not explicitly defined.

27. What will be the output of the following JavaScript code ?

```
const obj = { value: 10 };
function getValue(multiplier) {
  return this.value * multiplier;
}
console.log(getValue.call(obj, 2));
console.log(getValue.apply(obj, [2]));
const boundFn = getValue.bind(obj, 2);
console.log(boundFn());
```

Output:

```
20;
20;
20;
```

Explanation:

- .call: Invokes the function with this set to obj and arguments provided individually.
- .apply: Invokes the function with this set to obj and arguments as an array.
- .bind: Returns a new function with this permanently set to obj and arguments pre-applied.

30. What will be the output of the following JavaScript code ?

```
const key = "dynamicKey";
const obj = {
  [key]: 42,
};
console.log(obj.dynamicKey);
```

Output:

```
42;
```

Explanation:

- Computed property [key] evaluates to "dynamicKey".
- The property dynamicKey is created with a value of 42.

31. What will be the output of the following JavaScript code ?

```
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { ...obj1 };
obj2.b.c = 42;
console.log(obj1.b.c);
```

Output:

```
42
```

Explanation:

- The spread operator creates a **shallow copy**.
- Modifying obj2.b.c affects obj1.b.c because b is a reference.

32. What will be the output of the following JavaScript code ?

```
const obj = { x: 1 };
Object.freeze(obj);
obj.x = 42;
console.log(obj.x);
```

Output:

```
1;
```

Explanation:

- Object.freeze prevents modifications to the object.
- Attempts to modify x are ignored.

33. What will be the output of the following JavaScript code ?

```
const obj = Object.create({ a: 1 });
obj.b = 2;
console.log(obj.a);
console.log(obj.b);
console.log("a" in obj);
console.log(obj.hasOwnProperty("a"));
```

Output:

```
1;
2;
true;
false;
```

Explanation:

- a is accessed through the prototype chain.
- b is directly defined on obj.
- "a" in obj checks for a in the object or its prototype.
- hasOwnProperty only checks properties directly on obj.

JavaScript 60+ Output Based Questions

DOM:

34. What will be the output of the following JavaScript code ?

```
<!DOCTYPE html>
<html>
<body>
  <div id="box" class="red"></div>
  <script>
    const box = document.getElementById("box");
    box.classList.add("blue");
    box.classList.remove("red");
    console.log(box.className);
  </script>
</body>
</html>
```

Output:

```
blue;
```

Explanation:

- classList.add("blue") adds the blue class.
- classList.remove("red") removes the red class.

Event Handler:

35. What will be the output of the following JavaScript code ?

```
<!DOCTYPE html>
<html>
<body>
  <button id="btn">Click Me</button>
  <script>
    const button = document.getElementById("btn");
    button.addEventListener("click", () => console.log("Clicked"), { once: true });
    button.click();
    button.click();
  </script>
</body>
</html>
```

Output:

```
Clicked;
```

Explanation:

- The once: true option ensures the event listener is automatically removed after the first invocation.

Event Loop:

36. What will be the output of the following JavaScript code ?

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

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

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

console.log("End");
```

Output:

```
Start;
End;
Promise;
Timeout;
```

Explanation:

1. "Start" and "End" are logged synchronously.
2. Promise.then is a microtask and executes before the macrotask (setTimeout).
3. "Promise" is logged before "Timeout".

37. What will be the output of the following JavaScript code ?

```
console.log("1");

Promise.resolve().then(() => {
  console.log("2");
  Promise.resolve().then(() => {
    console.log("3");
  });
});

setTimeout(() => {
  console.log("4");
}, 0);

console.log("5");
```

Output:

```
1;
5;
2;
3;
```

```
4;
```

Explanation:

1. "C" is logged synchronously.
2. "A" is logged before the await.
3. "D" is logged synchronously while the microtask from await is queued.
4. "B" is logged when the microtask executes.

38. What will be the output of the following JavaScript code ?

```
setTimeout(() => console.log("Timeout"), 0);
setImmediate(() => console.log("Immediate"));
```

Output:

```
Immediate;
Timeout;
```

Explanation:

- In Node.js, setImmediate executes before setTimeout when both are scheduled simultaneously.

39. What will be the output of the following JavaScript code ?

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

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

const start = Date.now();
while (Date.now() - start < 1000) {}

console.log("End");
```

Output:

```
Start;
End;
Timeout;
```

Explanation:

1. The synchronous while loop blocks the event loop for 1 second.
2. "Timeout" is logged only after the loop ends and the event loop processes the macrotask.

40. What will be the output of the following JavaScript code ?

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
```

Output:

```
0;
1;
2;
```

Explanation:

- let creates a block-scoped variable i for each iteration, so each setTimeout captures its own value.

41. What will be the output of the following JavaScript code ?

```
async function test() {
  console.log("Start");
  const res1 = await Promise.resolve("A");
  console.log(res1);
  const res2 = await Promise.resolve("B");
  console.log(res2);
}
test();
console.log("End");
```

Output:

```
Start;
End;
A;
B;
```

Explanation:

1. "Start" is logged synchronously.
2. "End" is logged while the microtasks (await) are queued.
3. "A" and "B" are logged as the await resolves sequentially.

De-structuring:

42. What will be the output of the following JavaScript code ?

```
let a = 5, b = 10;
[a, b] = [b, a];
console.log(a, b);
```

Output:

```
10 5
```

Explanation:

- Destructuring swaps the values of a and b without using a temporary variable.

43. What will be the output of the following JavaScript code ?

```
const obj = { a: { b: { c: 42 } } };
const { a: { b: { c } } } = obj;
console.log(c);
```

Output:

```
42;
```

Explanation:

- Nested destructuring extracts the deeply nested value c.

44. What will be the output of the following JavaScript code ?

```
const key = "name";
const obj = { name: "Mahesh", age: 25 };
const { [key]: value } = obj;
console.log(value);
```

Output:

```
Mahesh;
```

Explanation:

- Dynamic property names ([key]) allow destructuring using variables.

Prototype:

45. What will be the output of the following JavaScript code ?

```
function Gadget() {}
Gadget.prototype.type = "Generic";

const gadget1 = new Gadget();
Gadget.prototype.type = "Updated";

console.log(gadget1.type);
```

Output:

```
Updated;
```

Explanation:

- Changing Gadget.prototype.type affects all instances that inherit from it unless they have an overridden type property.

46. What will be the output of the following JavaScript code ?

```
function Machine() {}
const machine = new Machine();

console.log(Machine.prototype === machine.__proto__);
```

Output:

```
true;
```

Explanation:

- The prototype property of the constructor function (Machine) is the prototype of the instances (machine.__proto__).

47. What will be the output of the following JavaScript code ?

```
function Parent() {}
Parent.prototype.greet = "Hello";

function Child() {}
Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;

const child = new Child();
console.log(child.greet);
console.log(child instanceof Parent);
console.log(child.constructor === Child);
```

Output:

```
Hello
true
true
```

Explanation:

1. Child.prototype = Object.create(Parent.prototype) makes Child inherit from Parent.
2. instanceof verifies the inheritance chain.
3. constructor is explicitly set to Child.

48. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3];
console.log(arr.__proto__ === Array.prototype);
console.log(Array.prototype.__proto__ === Object.prototype);
```

Output:

```
true;
true;
```

Explanation:

- Arrays inherit from Array.prototype, which in turn inherits from Object.prototype.

49. What will be the output of the following JavaScript code ?

```
class Calculator {
  static add(a, b) {
    return a + b;
  }
}

console.log(Calculator.add(5, 3));
const calc = new Calculator();
console.log(calc.add(2, 2));
```

Output:

```
8
TypeError: calc.add is not a function
```

Explanation:

- add is a static method, accessible via the class Calculator.add(5, 3).
- Static methods are not available on instances.
- Attempting to call calc.add(2, 2) results in a TypeError.

50. What will be the output of the following JavaScript code ?

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  get area() {
    return this.width * this.height;
  }
}

const rect = new Rectangle(5, 10);
console.log(rect.area);
rect.area = 100;
console.log(rect.area);
```

Output:

```
50;
50;
```

Explanation:

- The area property is a getter; it's computed from width and height.

- Attempting to set rect.area has no effect, as there is no setter defined.
- rect.area remains 50.

51. What will be the output of the following JavaScript code ?

```
const methodName = "greet";

class Greeter {
  [methodName]() {
    return "Hello!";
  }
}

const greeter = new Greeter();
console.log(greeter.greet());
```

Output:

```
Hello!
```

Explanation:

- Computed method names allow dynamic method names in classes.
- The method greet is defined using [methodName]().

52. What will be the output of the following JavaScript code ?

```
const p = new Person("John");
class Person {
  constructor(name) {
    this.name = name;
  }
}

console.log(p.name);
```

Output:

```
ReferenceError: Cannot access 'Person' before initialization
```

Explanation:

- Unlike function declarations, class declarations are not hoisted.
- Attempting to create an instance before the class declaration results in a ReferenceError.

Arrow Functions, Closures, and Callbacks:

53. What will be the output of the following JavaScript code ?

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

const increment = counter();
console.log(increment());
console.log(increment());
console.log(increment());
```

Output:

```
1;
2;
3;
```

Explanation:

- The inner function forms a **closure**, retaining access to the count variable in the outer function.
- Each call to increment() increments and returns the same count variable.

54. What will be the output of the following JavaScript code ?

```
function test() {
  const arrow = () => console.log(arguments[0]);
  arrow();
}

test(10);
```

Output:

```
10;
```

Explanation:

- Arrow functions do not have their own arguments object; they inherit it from the enclosing regular function test.

55. What will be the output of the following JavaScript code ?

```
console.log("1");

setTimeout(() => console.log("2"), 0);

Promise.resolve().then(() => console.log("3"));

console.log("4");
```

Output:

```
1;
4;
3;
2;
```

Explanation:

1. "1" and "4" are synchronous.
2. Promise.then (microtask) executes before setTimeout (macrotask).

56. What will be the output of the following JavaScript code ?

```
function outer() {
  const name = "Outer";
  const arrowFunc = () => {
    console.log(name);
  };
  arrowFunc();
}

outer();
```

Output:

```
Outer;
```

Explanation:

- Arrow functions have **lexical scope** and access the name variable from the enclosing outer function.

57. What will be the output of the following JavaScript code ?

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

Output:

```
3;
3;
3;
```

Explanation:

- var is function-scoped, so the same i is referenced in each iteration.
- By the time setTimeout runs, i is 3.

58. What will be the output of the following JavaScript code ?

```
for (var i = 0; i < 3; i++) {
  (function (j) {
    setTimeout(() => console.log(j), 1000);
  })(i);
}
```

Output:

```
0;
1;
2;
```

Explanation:

- Using an IIFE with j creates a closure, capturing the correct value of i for each iteration.

Debouncing and Throttling:

59. What will be the output of the following JavaScript code ?

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

const search = debounce((query) => console.log("Searching for:", query), 500);

search("A");
search("AB");
search("ABC");
setTimeout(() => search("ABCD"), 600);
```

Output:

```
Searching for: ABC
Searching for: ABCD
```

Explanation:

1. search("A"), search("AB"), and search("ABC") are called in quick succession.
2. The timer resets on each call, and only "ABC" is logged after 500ms.
3. The setTimeout after 600ms triggers a final call with "ABCD".

60. What will be the output of the following JavaScript code ?

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

const log = throttle(() => console.log("Throttled!"), 1000);

log();
setTimeout(log, 500);
setTimeout(log, 1000);
setTimeout(log, 2000);
```

Output:

```
Throttled!
Throttled!
Throttled!
```

Explanation:

1. The first call executes immediately.
2. The second call at 500ms is ignored because it's within the 1-second throttle delay.
3. The calls at 1000ms and 2000ms execute because the delay has passed.

Modules, Cookies, Local Storage, and Session Storage:

61. What will be the output of the following JavaScript code ?

```
// module.js
export const greet = "Hello, World!";
export function sayHi(name) {
  return `Hi, ${name}`;
}

// main.js
import { greet, sayHi } from "./module.js";

console.log(greet);
console.log(sayHi("Mahesh"));
```

Output:

```
Hello, World!
Hi, Mahesh
```

Explanation:

1. **export** allows you to export specific variables or functions.
2. **import** retrieves them in another file.
3. greet and sayHi are imported as named exports.

62. What will be the output of the following JavaScript code ?

```
document.cookie = "username=Mahesh; path=/";
console.log(document.cookie);
```

Output:

```
username = Mahesh;
```

Explanation:

- Assigning a new value to the same cookie (username) updates it.

63. What will be the output of the following JavaScript code ?

```
sessionStorage.setItem("sessionKey", "active");
console.log(sessionStorage.getItem("sessionKey"));
sessionStorage.clear();
console.log(sessionStorage.getItem("sessionKey"));
```

Output:

```
active;
null;
```

Explanation:

1. sessionStorage persists only for the session.
2. clear removes all data from sessionStorage.

64. What will be the output of the following JavaScript code ?

```

localStorage.setItem("a", 1);
localStorage.setItem("b", 2);

for (let i = 0; i < localStorage.length; i++) {
  const key = localStorage.key(i);
  console.log(key, localStorage.getItem(key));
}
```

Output:

```
a 1
b 2
```

Explanation:

- localStorage.key(i) retrieves the key at a specific index.

65. What will be the output of the following JavaScript code ?

```
document.cookie = "auth=true; path=/; max-age=3600";
```

```
localStorage.setItem("theme", "dark");
sessionStorage.setItem("sessionID", "123");

console.log(document.cookie);
console.log(localStorage.getItem("theme"));
console.log(sessionStorage.getItem("sessionID"));
```

Output:

```
auth = true;
dark;
123;
```

Explanation:

- Cookies, localStorage, and sessionStorage can coexist to store different kinds of data.

Conditional Statements, Switch Statements and Loops:

Task 1: Write a program to print all prime numbers between 1 and 20.

Solution:

```
function isPrime(num) {
  if (num <= 1) return false; // Numbers <= 1 are not prime
  for (let i = 2; i <= Math.sqrt(num); i++) {
    if (num % i === 0) return false; // If divisible by any number, it's not prime
  }
  return true; // If no divisors found, the number is prime
}

function printPrimes(n) {
  for (let i = 1; i <= n; i++) {
    if (isPrime(i)) {
      console.log(i); // Print the number if it's prime
    }
  }
}

printPrimes(20);
```

Output:

```
2;
3;
5;
7;
11;
13;
17;
19;
```

isPrime Function

Breakdown of the Steps:

1. **Check if num <= 1:**
   - Numbers less than or equal to 1 (like 0 and 1) are **not prime**.
   - Hence, the function returns false.
2. **Optimize Prime Check Using Math.sqrt:**
   - We loop from 2 to Math.sqrt(num).
   - Why Math.sqrt(num)?
     - If a number has a factor larger than its square root, it must also have a smaller factor that has already been checked.
     - For example:

        - To check if 16 is prime, you only need to test divisors 2, 3, and 4 (since 4 * 4 = 16).
3. **Check for Divisors:**
    - If the number is divisible by any i in the loop (num % i === 0), the function returns false because it has a factor other than 1 and itself.
4. **Return true:**
    - If no divisors are found, the number is prime, and the function returns true.

printPrimes Function

Steps:

1. Use a for loop to iterate from 1 to n (in this case, 20).
2. For each number i, call the isPrime function.
3. If isPrime(i) returns true, print the number (console.log(i)).

Execution of printPrimes(20)

1. Start with i = 1:
    - isPrime(1) → false → Not printed.
2. Move to i = 2:
    - isPrime(2) → true → Print 2.
3. i = 3:
    - isPrime(3) → true → Print 3.
4. i = 4:
    - isPrime(4) → false (divisible by 2) → Not printed.
5. Continue this for all numbers up to 20.
   The prime numbers are **2, 3, 5, 7, 11, 13, 17, 19**.

Task 2 : Write a program to print the Fibonacci sequence up to n terms using a while loop.

Solution:

```
function fibonacci(n) {
  let a = 0, // First term
    b = 1, // Second term
    nextTerm; // Variable to store the next term
  let count = 0; // Counter for the loop

  while (count < n) {
    console.log(a); // Print the current term
    nextTerm = a + b; // Calculate the next term
    a = b; // Move 'b' to 'a'
    b = nextTerm; // Move 'nextTerm' to 'b'
    count++; // Increment the counter
  }
}

fibonacci(7);
```

Output:

```
0;
1;
1;
2;
3;
5;
8;
```

How It Works

1. **Initialize Values:**
   - Start with a = 0 (first term) and b = 1 (second term).
   - nextTerm is used to calculate the next term in the sequence.
   - count keeps track of how many terms have been printed.
2. **While Loop:**
   - The condition count < n ensures the loop runs until n terms are printed.
   - Print the current value of a (the Fibonacci term).
   - Calculate the **next term**: nextTerm = a + b.
   - Update values:
     - Move b to a.
     - Move nextTerm to b.
   - Increment the count.
3. **Repeat:**
   - The loop continues until n terms are printed.

Task 3 : Write a program to print the following pattern:

```
*
* *
* * *
* * * *
* * * * *
```

Solution:

```
function printPattern(n) {
  for (let i = 1; i <= n; i++) {
    let row = ""; // Initialize an empty string for each row
    for (let j = 1; j <= i; j++) {
      row += "* "; // Append a star and a space for each column
    }
    console.log(row); // Print the row
  }
}

printPattern(5);
```

How It Works

1. **Outer Loop (Rows):**
   - The **outer loop** runs from i = 1 to n.
   - Each iteration represents a new row.
2. **Inner Loop (Stars in Each Row):**
   - The **inner loop** runs from j = 1 to i (the current row number).
   - For each iteration of the inner loop, a star (*) followed by a space ( ) is appended to the row string.
3. **Print Row:**
   - After the inner loop completes, print the row string using console.log.
4. **Repeat for Next Row:**
   - The outer loop increments, and the process repeats for the next row

Array:

Task 4 : Write a function removeDuplicates that removes duplicate elements from an array.

```
[1, 2, 2, 3, 4, 4, 5];
```

Solution:

```
function removeDuplicates(arr) {
  return [...new Set(arr)]; // Step 1: Convert the array to a Set, then spread it into a new array.
}

console.log(removeDuplicates([1, 2, 2, 3, 4, 4, 5])); // Output: [1, 2, 3, 4, 5]
```

Step-by-Step Explanation

1. **Set**:
   - A Set automatically removes duplicate values.
   - Example: new Set([1, 2, 2, 3]) → {1, 2, 3}.
2. **Spread Operator**:
   - Converts the Set back into an array.
   - Example: [...new Set([1, 2, 2])] → [1, 2].

Task 5 : Write a function flattenArray that takes a deeply nested array and returns a single flattened array.

```
1, [2, [3, [4, 5]]]]);
// Output: [1, 2, 3, 4, 5]
```

Solution:

```
function flattenArray(arr) {
  return arr.flat(Infinity); // Step 1: Use flat() with Infinity to flatten all levels
}

console.log(flattenArray([1, [2, [3, [4, 5]]]]));
// Output: [1, 2, 3, 4, 5]
```

Explanation

1. **flat(Infinity)**:
   - The flat() method flattens nested arrays by a specified depth.
   - Passing Infinity ensures all levels of nesting are flattened.
2. **Return**:
   - The function directly returns the flattened array.

Task 6 : Write a function to find all pairs of numbers in an array that add up to a given target.

Solution:

```
function findPairs(arr, target) {
  let result = [];
  let seen = new Set();

  for (let num of arr) {
    let complement = target - num;
    if (seen.has(complement)) {
      result.push([complement, num]); // If complement exists, add the pair
    }
    seen.add(num); // Add the current number to the set
  }
  return result;
}

console.log(findPairs([2, 4, 3, 5, 7, 8, 1], 9));
// Output: [[5, 4], [7, 2], [8, 1]]
```

Step-by-Step Explanation

1. **Initialize Storage**:
   - Use a Set to keep track of visited numbers.
   - Use an array result to store pairs.
2. **Calculate Complement**:
   - For each number, calculate the complement: target - num.
3. **Check for Complement**:
   - If the complement exists in the Set, it means we found a pair.
4. **Update Set**:
   - Add the current number to the Set for future checks.
5. **Return Result**:
   - Return the array of pairs.

Task 7 : Write a function findMissingNumbers that takes an array of numbers in a sequence and returns the numbers that are missing from the sequence.

```
findMissingNumbers([1, 2, 4, 6, 7]);
```

```
// Output: [3, 5]

findMissingNumbers([10, 12, 14]);
// Output: [11, 13]
```

Solution:

```
function findMissingNumbers(arr) {
  const max = Math.max(...arr); // Step 1: Find the largest number
  const min = Math.min(...arr); // Step 2: Find the smallest number
  const fullRange = Array.from({ length: max - min + 1 }, (_, i) => i + min); // Step 3:
Generate the full range
  return fullRange.filter((num) => !arr.includes(num)); // Step 4: Filter numbers not in the
input array
}
console.log(findMissingNumbers([1, 2, 4, 6, 7])); // Output: [3, 5]
console.log(findMissingNumbers([10, 12, 14])); // Output: [11, 13]
```

Step-by-Step Explanation

1. **Find Minimum and Maximum Values**:
   - Use Math.min(...arr) and Math.max(...arr) to get the smallest and largest numbers in the input array.
2. **Generate the Full Range**:
   - Use Array.from to create an array containing all numbers from min to max.
   - Example: If min = 1 and max = 7, generate [1, 2, 3, 4, 5, 6, 7].
3. **Filter Missing Numbers**:
   - Use filter() to keep only those numbers from the full range that are **not included** in the input array.
   - Check membership using arr.includes(num).
4. **Return Result**:
   - Return the array of missing numbers.

Task 8 : Write a function to find the most frequent element in an array.

Solution:

```
function mostFrequent(arr) {
  let counts = {};
  let maxElement = null;
  let maxCount = 0;

  arr.forEach((item) => {
    counts[item] = (counts[item] || 0) + 1; // Update count
    if (counts[item] > maxCount) {
      maxCount = counts[item]; // Track the highest count
      maxElement = item; // Track the most frequent element
    }
  });
  return maxElement;
}
```

```
console.log(mostFrequent([1, 3, 2, 3, 4, 1, 3])); // Output: 3
```

Step-by-Step Explanation

1. **Track Counts**:
   - Use an object counts to store occurrences of each element.
2. **Check Most Frequent**:
   - Track the element with the highest count using maxCount and maxElement.
3. **Update Values**:
   - For each occurrence, compare the count to maxCount and update accordingly.
4. **Return Result**:
   - Return the element with the highest count.

Function:

Task 9 : Write a function factorial that calculates the factorial of a given number using recursion.

Solution:

```
function factorial(n) {
  if (n === 0 || n === 1) return 1; // Base case: factorial(0) = 1 and factorial(1) = 1
  return n * factorial(n - 1); // Recursive case
}

console.log(factorial(5)); // Output: 120
```

Step-by-Step Explanation

1. **Base Case**:
   - If n is 0 or 1, the factorial is 1.
2. **Recursive Case**:
   - Multiply n by the factorial of n - 1.
3. **Execution**:
   - The function calls itself with smaller values until it reaches the base case.

Task 10 : Write a function fibonacci that generates the nth Fibonacci number using recursion

Solution:

```
function fibonacci(n) {
  if (n === 0) return 0; // Base case 1
  if (n === 1) return 1; // Base case 2
  return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

console.log(fibonacci(6)); // Output: 8
```

Step-by-Step Explanation

1. **Base Cases**:
   - If n is 0, return 0.
   - If n is 1, return 1.
2. **Recursive Case**:
   - Sum the results of the n-1 and n-2 Fibonacci numbers.
3. **Execution**:
   - Calls continue until base cases are reached, then results are summed up.

Task 11 : Write a function objectToArray that converts an object into an array of [key, value] pairs.

Solution:

```
function objectToArray(obj) {
  return Object.entries(obj); // Convert object to array of [key, value] pairs
}

const obj = { a: 1, b: 2, c: 3 };
console.log(objectToArray(obj));
// Output: [["a", 1], ["b", 2], ["c", 3]]
```

Step-by-Step Explanation

1. **Object.entries**:
   - Converts the object into an array where each element is a [key, value] pair.
2. **Return the Result**:
   - The resulting array contains all key-value pairs.

Task 12 : Write a function isEmptyObject that checks if an object is empty.

Solution:

```
function isEmptyObject(obj) {
  return Object.keys(obj).length === 0; // Check if the object has no keys
}

console.log(isEmptyObject({})); // Output: true
console.log(isEmptyObject({ a: 1 })); // Output: false
```

Step-by-Step Explanation

1. **Get Keys**:
   - Use Object.keys(obj) to get an array of keys.
2. **Check Length**:
   - If the array length is 0, the object is empty.

Event Handlers, Event Propagation, and Event Delegation:

Task 13 : Write a function toggleClassOnClick that adds or removes a CSS class when a button is clicked.

Solution:

```
function toggleClassOnClick(buttonId, targetId, className) {
  const button = document.getElementById(buttonId);
  const target = document.getElementById(targetId);

  button.addEventListener("click", () => {
    target.classList.toggle(className); // Step 3: Toggle the class
  });
}

toggleClassOnClick("toggleButton", "myDiv", "highlight");
```

Step-by-Step Explanation

1. **Select Elements**:
   - Use document.getElementById to select the button and target element.
2. **Add an Event Listener**:
   - Use .addEventListener("click") to listen for click events on the button.
3. **Toggle Class**:
   - Use classList.toggle(className) to add or remove the class.

Task 14 : Write a function animateElement that moves an element 10px to the right every 100ms.

Solution:

```
function animateElement(id) {
  const element = document.getElementById(id); // Step 1: Select the element by ID
  let position = 0;

  const interval = setInterval(() => {
    position += 10; // Step 2: Increment position
    element.style.transform = `translateX(${position}px)`; // Step 3: Update position
    if (position >= 100) {
      clearInterval(interval); // Step 4: Stop animation
    }
  }, 100);
}

animateElement("myDiv");
```

Step-by-Step Explanation

1. **Select the Element**:
   - Use document.getElementById(id) to find the element.
2. **Initialize Position**:
   - Use a variable position to track the current position.
3. **Animate Using setInterval**:
   - Update the position every 100ms using style.transform.
4. **Stop the Animation**:
   - Use clearInterval() when the position reaches 100px.

Task 15 : Write a function logEventPhases that adds event listeners in both the capturing and bubbling phases to demonstrate event propagation.

Solution:

```
function logEventPhases() {
  const parent = document.getElementById("parent");
  const child = document.getElementById("child");

  parent.addEventListener("click", () => {
    console.log("Parent - Bubble Phase");
  });

  parent.addEventListener(
    "click",
    () => {
      console.log("Parent - Capture Phase");
    },
    true // Enable capture phase
  );

  child.addEventListener("click", (event) => {
    console.log("Child Clicked");
  });
}

logEventPhases();
```

Step-by-Step Explanation

1. **Capture Phase**:

   - Add an event listener to parent with the third argument as true to enable capturing.
2. **Bubble Phase**:
   - Add a standard event listener to parent (default is bubbling).
3. **Trigger Child Event**:
   - Click on the child to see how the event flows through the DOM.

Task 16 : Write a program to demonstrate the execution order of microtasks and macrotasks.

Solution:

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

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

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

console.log("End");
```

Output:

```
Start;
End;
Microtask;
Macrotask;
```

Step-by-Step Explanation:

1. **Synchronous Code**:
   - Logs "Start" and "End" first.
2. **Microtask**:
   - The Promise callback executes before the macrotask because microtasks have higher priority.
3. **Macrotask**:
   - The setTimeout callback is executed after the microtask.

Task 17 : Write a function to demonstrate the difference between setImmediate (Node.js) and setTimeout.

Solution:

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

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

```
setImmediate(() => {
  console.log("Immediate");
});
console.log("End");
```

Output:

```
Start;
End;
Immediate;
Timeout;
```

Step-by-Step Explanation:

1. **Synchronous Code**:
   - Logs "Start" and "End".
2. **setImmediate**:
   - Executes before setTimeout because setImmediate is added to the **check phase**, while setTimeout is in the **timer phase**.

Spread and Rest:

Task 18 : Create a shallow copy of an object or array using the spread operator. Demonstrate how changes in nested properties affect the copied version.

Solution:

```
// Shallow Copy of an Object
const originalObject = { a: 1, b: { c: 2 } };
const shallowCopyObject = { ...originalObject };

shallowCopyObject.b.c = 99;

console.log(originalObject); // Output: { a: 1, b: { c: 99 } }
console.log(shallowCopyObject); // Output: { a: 1, b: { c: 99 } }

// Shallow Copy of an Array
const originalArray = [1, [2, 3]];
const shallowCopyArray = [...originalArray];

shallowCopyArray[1][0] = 99;

console.log(originalArray); // Output: [1, [99, 3]]
console.log(shallowCopyArray); // Output: [1, [99, 3]]
```

Explanation

1. **Shallow Copy**:
   - Copies only the first level of the object or array.
   - Nested objects or arrays are still **referenced**, meaning changes in the nested structure will affect both the original and the copy.

2. **Spread Operator**:
    - Expands the properties of the object or array into a new container.

Task 19 : Create a deep copy of an object or array so that changes in nested properties do not affect the original.

Solution:

```
const originalObject = { a: 1, b: { c: 2 } };
const deepCopyObject = JSON.parse(JSON.stringify(originalObject));

deepCopyObject.b.c = 99;

console.log(originalObject); // Output: { a: 1, b: { c: 2 } }
console.log(deepCopyObject); // Output: { a: 1, b: { c: 99 } }
```

Explanation

1. **Deep Copy**:
    - Creates an entirely new copy of the object or array, including all nested properties.
    - Modifications to the copied object or array do not affect the original.
2. **JSON Method**:
    - JSON.stringify() converts the object to a string.
    - JSON.parse() converts the string back to a new object.
    - Limitation: It does not handle functions or undefined.

Task 20 : Highlight the differences between shallow copy and deep copy with examples.

Solution:

```
const original = { a: 1, b: { c: 2 } };

// Shallow Copy
const shallowCopy = { ...original };
shallowCopy.b.c = 99;

console.log("Shallow Copy:");
console.log("Original:", original); // Output: { a: 1, b: { c: 99 } }
console.log("Copy:", shallowCopy); // Output: { a: 1, b: { c: 99 } }

// Deep Copy
const deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.b.c = 42;

console.log("Deep Copy:");
console.log("Original:", original); // Output: { a: 1, b: { c: 99 } }
console.log("Copy:", deepCopy); // Output: { a: 1, b: { c: 42 } }
```

Creating the Shallow Copy:

- The **spread operator** ({ ...original }) creates a **shallow copy** of the original object.
- A shallow copy only duplicates the **first level** of properties.
  - The property a is copied as a new value because it's a primitive.
  - The property b (a nested object) is **referenced**, not duplicated.

Modifying the Nested Object:

- When shallowCopy.b.c = 99 is executed, it modifies the **same nested object (b)** that exists in the original object.
- This happens because b is **shared between the original and the shallow copy**.

Both the original and shallowCopy objects reflect the change to b.c, showing c: 99.

Creating the Deep Copy:

- The JSON.stringify(original) converts the original object into a JSON string: {"a":1,"b":{"c":2}}.
- The JSON.parse() then parses the JSON string back into a **new object**.
- This creates a **deep copy** of the original object, duplicating all levels of nested properties.

Modifying the Nested Object:

- When deepCopy.b.c = 42 is executed, it modifies the b object in the deepCopy, which is **independent** of the b object in the original.
- Changes to deepCopy do not affect the original.

The original object remains unchanged with b: { c: 99 }.

The deepCopy has its own b object with c: 42.

Destructuring:

Task 21 : Given an object representing a user

```
const user = {
  id: 1,
  details: {
    userName: "Mahesh",
    age: 25,
  },
  address: {
    city: "HYD",
    zip: 10001,
  },
};
```

You need to:

1. **Extract the userName property from the details object.**
2. **Rename the extracted userName to name.**
3. **Extract the city property from the address object.**

Solution:

```
const { details: { userName: name },  address: { city },} = user;

console.log(name); // Output: Mahesh
console.log(city); // Output: HYD
```

Explanation

Here's the destructuring explained step by step:

1. **Syntax**:

```
const {  details: { userName: name },  address: { city },} = user;
```

   - { details: { userName: name } }:
     - Access the details object, extract userName, and rename it to name.
   - { address: { city } }:
     - Access the address object and extract city.
2. **Final Variables**:
   - name holds the value "Mahesh".
   - city holds the value "HYD".

Task 22 : : Given an object

```
const product = {
  id: 101,
  name: "Laptop",
};
```

Extract the name property.
Extract the stock property, but if it's missing, assign it a default value of 0.

Solution:

```
const { name, stock = 0 } = product;

console.log(name); // Output: Laptop
console.log(stock); // Output: 0
```

Explanation:

- **Default Value**:
  - stock = 0: If stock doesn't exist in the object, it will default to 0.

Task 23 : Given three variables:

```
let a = 1,
  b = 2,
  c = 3;
```

Swap their values such that a = 3, b = 1, and c = 2.

Solution:

```
[a, b, c] = [c, a, b];

console.log(a); // Output: 3
console.log(b); // Output: 1
console.log(c); // Output: 2
```

Explanation:

1. **Array Destructuring**:
   - [a, b, c] = [c, a, b]: Creates a new array with values rearranged and assigns them back to a, b, and c.

Task 24 : Given a nested array:

```
const matrix = [
  [1, 2],
  [3, 4],
  [5, 6],
];
```

Extract the first element of each sub-array.

Solution:

```
const [[a], [b], [c]] = matrix;

console.log(a); // Output: 1
console.log(b); // Output: 3
console.log(c); // Output: 5
```

Explanation:

1. **Nested Destructuring**:
   - Use [a], [b], [c] to destructure each sub-array and extract its first element.

HOF:

Task 25 :

Write a function multiplyBy that:

- Accepts a number factor.
- Returns a new function that multiplies its input by factor.

Solution:

```
function multiplyBy(factor) {
  return (num) => num * factor;
}

const double = multiplyBy(2);
const triple = multiplyBy(3);

console.log(double(5)); // Output: 10
console.log(triple(5)); // Output: 15
```

Explanation:

1. **Higher-Order Function**:
   - multiplyBy accepts factor and returns a function.
2. **Reusable Functions**:
   - double and triple are created by passing specific factor values to multiplyBy.

Advanced Array methods:

Task 26 : From an array of numbers

```
const numbers = [1, 2, 3, 4, 5, 6];
```

create a new array of squares for even numbers only.

Solution:

```
const numbers = [1, 2, 3, 4, 5, 6];
const evenSquares = numbers
  .filter((num) => num % 2 === 0)
  .map((num) => num * num);

console.log(evenSquares); // Output: [4, 16, 36]
```

Explanation:

1. **filter**:
   - Filters the array to include only even numbers [2, 4, 6].
2. **map**:
   - Squares each number in the filtered array.

Task 27 : Given an array of strings:

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];
```

Count how many times each fruit appears.

Solution:

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];
const counts = fruits.reduce((acc, fruit) => {
  acc[fruit] = (acc[fruit] || 0) + 1;
  return acc;
}, {});

console.log(counts); // Output: { apple: 3, banana: 2, orange: 1 }
```

Explanation:

1. **reduce Function**:
   - Uses an object (acc) to store counts for each fruit.
   - Updates the count for each fruit: (acc[fruit] || 0) + 1.
2. **Output**:
   - An object { apple: 3, banana: 2, orange: 1 }.

Task 28 : Given a nested array:

```
const nestedArray = [
  [1, 2],
  [3, 4],
  [5, 6],
];
```

Flatten it into a single array.

Solution:

```
const nestedArray = [
  [1, 2],
  [3, 4],
  [5, 6],
];
const flatArray = nestedArray.reduce((acc, arr) => acc.concat(arr), []);

console.log(flatArray); // Output: [1, 2, 3, 4, 5, 6]
```

Explanation:

1. **reduce Function**:
   - Combines each sub-array into the accumulator (acc).
   - Uses concat to merge arrays.
2. **Output**:
   - A flattened array [1, 2, 3, 4, 5, 6].

Task 29 : Given an array with duplicates:

```
const numbers = [1, 2, 3, 2, 4, 1, 5];
```

- **Create a new array with unique numbers.**

Solution:

```
const numbers = [1, 2, 3, 2, 4, 1, 5];
const uniqueNumbers = numbers.filter(
  (num, index, arr) => arr.indexOf(num) === index
);

console.log(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

Explanation:

1. **filter Function**:
   - Keeps only the first occurrence of each number.
2. **indexOf**:
   - Returns the index of the first occurrence of a number.
3. **Output**:
   - A new array [1, 2, 3, 4, 5].

Task 30 : Given an array of objects:

```
const employees = [
  { name: "Mahesh", age: 30 },
  { name: "Hema", age: 25 },
  { name: "TATA", age: 35 },
];
```

Sort the array by age in ascending order.

Solution:

```
employees.sort((a, b) => a.age - b.age);

console.log(employees);
```

```
// Output: [
//   { name: "Hema", age: 25 },
//   { name: "Mahesh", age: 30 },
//   { name: "TATA", age: 35 }
// ]
```

Explanation:

1. **sort Method**:
   - Compares two elements a and b based on their age property.
   - a.age - b.age ensures ascending order.
2. **Result**:
   - The array is sorted from the youngest to the oldest.

Task 31 : Given an array of objects:

```
const students = [
  { name: "Mahesh", grade: "A" },
  { name: "Hema", grade: "B" },
  { name: "TATA", grade: "A" },
  { name: "Sudha", grade: "C" },
];
```

Group the students by their grade.

Solution:

```
const groupedByGrade = students.reduce((acc, student) => {
  const { grade } = student;
  acc[grade] = acc[grade] || [];
  acc[grade].push(student);
  return acc;
}, {});

console.log(groupedByGrade);
// Output: {
//   A: [{ name: "Mahesh", grade: "A" }, { name: "TATA", grade: "A" }],
//   B: [{ name: "Hema", grade: "B" }],
//   C: [{ name: "Sudha", grade: "C" }]
// }
```

Explanation:

1. **reduce**:
   - Iterates over the array, adding each student to their respective grade group in the accumulator (acc).
2. **Grouping**:
   - acc[grade] = acc[grade] || [] initializes the group if it doesn't exist.
3. **Result**:

- A grouped object by grade.

Task 32 : Given an array of objects:

```
const data = [
  { id: 1, name: "Sudha" },
  { id: 2, name: "Mahesh" },
  { id: 1, name: "Sudha" },
  { id: 3, name: "Hema" },
];
```

Remove duplicates based on the id.

Solution:

```
const uniqueData = data.filter(
  (obj, index, self) => index === self.findIndex((el) => el.id === obj.id)
);

console.log(uniqueData);
// Output: [
//   { id: 1, name: "Sudha" },
//   { id: 2, name: "Mahesh" },
//   { id: 3, name: "Hema" }
// ]
```

Explanation:

1. **filter**:
   - Filters elements by comparing their id with the index of the first occurrence in the array.
2. **findIndex**:
   - Ensures only the first instance of each id is included.

Task 33 : Given two arrays:

```
const array1 = [
  { id: 1, name: "Hema" },
  { id: 2, name: "Mahesh" },
];
const array2 = [
  { id: 1, age: 25 },
  { id: 2, age: 30 },
];
```

Merge them into a single array of objects by id.

Solution:

```
const mergedArray = array1.map((item) => ({
  ...item,
  ...array2.find((el) => el.id === item.id),
}));

console.log(mergedArray);
// Output: [
//   { id: 1, name: "Hema", age: 25 },
//   { id: 2, name: "Mahesh", age: 30 }
// ]
```

Explanation:

1. **map**:
   - Iterates over array1 and creates a new object for each element.
2. **find**:
   - Finds the matching object in array2 by id.
3. **Spread Operator**:
   - Combines properties from both objects.

Task 34 : Given an array of objects

```
const items = [
  { id: 1, value: 10 },
  { id: 2, value: 50 },
  { id: 3, value: 30 },
];
```

Find the object with the maximum value.

Solution:

```
const maxValueItem = items.reduce((max, item) =>
  item.value > max.value ? item : max
);

console.log(maxValueItem); // Output: { id: 2, value: 50 }
```

Explanation:

1. **reduce**:
   - Compares each object’s value with the current maximum and updates the accumulator if needed.
2. **Result**:
   - The object with the highest value is { id: 2, value: 50 }.

Task 35 : Given an array of objects:

```
const products = [
  { name: "Laptop", price: 1000, category: "Electronics" },
  { name: "Shirt", price: 50, category: "Clothing" },
  { name: "Phone", price: 800, category: "Electronics" },
  { name: "Pants", price: 60, category: "Clothing" },
];
```

Filter only products in the "Electronics" category.
Extract their price.
Calculate the total price of all filtered products.

Solution:

```
const totalElectronicsPrice = products
  .filter((product) => product.category === "Electronics") // Step 1: Filter
  .map((product) => product.price) // Step 2: Extract price
  .reduce((total, price) => total + price, 0); // Step 3: Sum prices

console.log(totalElectronicsPrice); // Output: 1800
```

Explanation:

1. **filter**:
   - Selects only products in the "Electronics" category.
   - Result: [{ name: "Laptop", price: 1000 }, { name: "Phone", price: 800 }].
2. **map**:
   - Extracts the price property.
   - Result: [1000, 800].
3. **reduce**:
   - Sums the prices.
   - Result: 1800.

Task 36: given an array of numbers:

```
const numbers = [1, 2, 3, 4, 5, 6];
```

Filter out odd numbers.
Square the remaining numbers.
Calculate their sum.

Solution:

```
const result = numbers
  .filter((num) => num % 2 === 0) // Step 1: Filter evens
  .map((num) => num * num) // Step 2: Square them
  .reduce((sum, num) => sum + num, 0); // Step 3: Sum them

console.log(result); // Output: 56
```

Explanation:

1. **filter**:
   - Keeps only even numbers [2, 4, 6].

2. **map**:
   - Squares each even number [4, 16, 36].
3. **reduce**:
   - Sums the squared numbers 56.

Task 37: given an array of objects:

```
const students = [
  { name: "Hema", marks: [85, 90, 80] },
  { name: "Mahesh", marks: [70, 75, 78] },
  { name: "Sudha", marks: [88, 92, 86] },
];
```

Calculate the average marks of all students.

Solution:

```
const averageMarks = students
  .map((student) => student.marks.reduce((a, b) => a + b, 0) / student.marks.length) // Step 1: Calculate each student's average
  .reduce((acc, avg) => acc + avg, 0) / students.length; // Step 2: Find the overall average

console.log(averageMarks); // Output: 83.5
```

Explanation:

map:

- Calculates the average marks for each student.
- Result: [85, 74.33, 88.67].

reduce:

- Sums the averages and divides by the total number of students.
- Result: 83.5.

Task 38: given an array of products

```
const products = [
  { name: "Laptop", category: "Electronics" },
  { name: "Shirt", category: "Clothing" },
  { name: "Phone", category: "Electronics" },
  { name: "Pants", category: "Clothing" },
  { name: "Fridge", category: "Appliances" },
];
```

Extract all unique categories.
Count how many products belong to each category.

Solution:

```
const categoryCounts = products
  .map((product) => product.category) // Step 1: Extract categories
  .reduce((acc, category) => {
    acc[category] = (acc[category] || 0) + 1;
    return acc;
  }, {});

console.log(categoryCounts);
// Output: { Electronics: 2, Clothing: 2, Appliances: 1 }
```

Explanation:

map:

- Extracts the category property from each product.
- Result: ["Electronics", "Clothing", "Electronics", "Clothing", "Appliances"].

reduce:

- Counts occurrences of each category.
- Result: { Electronics: 2, Clothing: 2, Appliances: 1 }.

Task 39: given an array of products

```
const products = [
  { id: 1, name: "Laptop", price: 1000, category: "Electronics" },
  { id: 2, name: "Shirt", price: 50, category: "Clothing" },
  { id: 3, name: "Phone", price: 800, category: "Electronics" },
  { id: 4, name: "Pants", price: 60, category: "Clothing" },
];
```

Filter out products in the "Electronics" category.
Add a discountedPrice (10% off).
Calculate the total discountedPrice of these products.

Solution:

```
const totalDiscountedPrice = products
  .filter((product) => product.category === "Electronics") // Step 1: Filter
  .map((product) => ({ ...product, discountedPrice: product.price * 0.9 })) // Step 2: Add
discountedPrice
  .reduce((total, product) => total + product.discountedPrice, 0); // Step 3: Calculate total

console.log(totalDiscountedPrice); // Output: 1620
```

Explanation:

1. **filter**:
   - Filters products in the "Electronics" category.
   - Result: [{ id: 1, price: 1000 }, { id: 3, price: 800 }].
2. **map**:
   - Adds a discountedPrice (10% off).

   - Result: [{ id: 1, discountedPrice: 900 }, { id: 3, discountedPrice: 720 }].

3. **reduce**:
   - Sums up the discountedPrice.
   - Result: 1620.

Task 40: given an array of orders, each containing a list of purchased products:

```
const orders = [
  { id: 1, products: ["Laptop", "Phone"] },
  { id: 2, products: ["Shirt", "Phone"] },
  { id: 3, products: ["Laptop", "Pants"] },
];
```

Flatten the list of all products.
Remove duplicate products.

Solution:

```
const uniqueProducts = orders
  .flatMap((order) => order.products) // Step 1: Flatten product lists
  .filter((product, index, self) => self.indexOf(product) === index); // Step 2: Remove
duplicates

console.log(uniqueProducts);
// Output: ["Laptop", "Phone", "Shirt", "Pants"]
```

Explanation:

flatMap:

- Flattens all product arrays into a single array.
- Result: ["Laptop", "Phone", "Shirt", "Phone", "Laptop", "Pants"].

filter:

- Removes duplicates using indexOf.
- Result: ["Laptop", "Phone", "Shirt", "Pants"].

Task 41: given an array of students:

```
const students = [
  { id: 1, name: "Hema", marks: 85 },
  { id: 2, name: "Mahesh", marks: 72 },
  { id: 3, name: "Sudha", marks: 60 },
  { id: 4, name: "TATA", marks: 92 },
];
```

Check if any student failed (marks < 40).
Verify if all students passed (marks >= 60).
Calculate the average marks.

Solution:

```
const anyFailed = students.some((student) => student.marks < 40); // Step 1: Check if any
failed
const allPassed = students.every((student) => student.marks >= 60); // Step 2: Check if all
passed
const averageMarks =
  students.reduce((total, student) => total + student.marks, 0) /
  students.length; // Step 3: Calculate average

console.log(anyFailed); // Output: false
console.log(allPassed); // Output: true
console.log(averageMarks); // Output: 77.25
```

Explanation:

some:

- Checks if any student failed (marks < 40).

every:

- Verifies if all students passed (marks >= 60).

reduce:

- Calculates the total marks and divides by the number of students.

Cookies, Local Storage, and Session Storage:

Task 42: to store a user's login status (true) in session storage and remove it after the session ends.

Solution:

```


// Store login status
sessionStorage.setItem("isLoggedIn", "true");

// Retrieve and check login status
const isLoggedIn = sessionStorage.getItem("isLoggedIn") === "true";
console.log(isLoggedIn); // Output: true

// Remove login status
sessionStorage.removeItem("isLoggedIn");
console.log(sessionStorage.getItem("isLoggedIn")); // Output: null
```

Explanation:

Session-Based Storage:

- sessionStorage.setItem(key, value) stores data for the current session only.

Remove Data:

- sessionStorage.removeItem(key) deletes the stored key-value pair.

Task 43: Create a cookie named "username" with the value "Mahesh", and set it to expire in 7 days.

Solution:

```
// Set a cookie
document.cookie = "username=Mahesh; max-age=" + 7 * 24 * 60 * 60;

// Read cookies
console.log(document.cookie); // Output: "username=Mahesh"
```

Explanation:

Setting a Cookie:

- Use document.cookie to set the cookie. The max-age specifies the expiration time in seconds (7 days).

Reading Cookies:

- document.cookie retrieves all cookies as a single string.

Task 44: Create a cookie consent banner that stores the user's consent in cookies and hides the banner once consent is given.

Solution:

```
// Check for consent cookie
if (!document.cookie.includes("consent=true")) {
  console.log("Show consent banner");

  // Simulate user giving consent
  document.cookie = "consent=true; max-age=" + 365 * 24 * 60 * 60; // 1 year
  console.log("Consent given");
} else {
  console.log("Consent already given");
}
```

Explanation:

Cookie Check:

- Check if the "consent=true" cookie exists.

Set Cookie:

- If not, set the cookie when consent is given.

Task 45: Store data in local storage with an expiry time. If the expiry has passed, remove the data.

Solution:

```
// Store data with expiry
const setWithExpiry = (key, value, ttl) => {
  const now = new Date();
  const item = {
    value,
    expiry: now.getTime() + ttl,
  };
  localStorage.setItem(key, JSON.stringify(item));
};

// Retrieve data with expiry check
const getWithExpiry = (key) => {
  const itemStr = localStorage.getItem(key);
  if (!itemStr) return null;

  const item = JSON.parse(itemStr);
  const now = new Date();

  if (now.getTime() > item.expiry) {
    localStorage.removeItem(key);
    return null;
  }
  return item.value;
};

// Example usage
setWithExpiry("token", "abc123", 5000); // Expires in 5 seconds
console.log(getWithExpiry("token")); // Output: "abc123"
setTimeout(() => console.log(getWithExpiry("token")), 6000); // Output: null
```

Explanation:

Set Expiry:

- Store both the value and expiry time in local storage.

Check Expiry:

- On retrieval, compare the current time with the stored expiry time.

Debounce and Throttle Function:

Task 46: Write a debounce function that delays the execution of a given function until after a specified time has passed since the last time it was called.

Solution:

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

// Example Usage
const logMessage = (message) => console.log(message);

const debouncedLog = debounce(logMessage, 1000);

debouncedLog("Hello"); // Only logs after 1 second if no new call occurs
```

Explanation:

Concept:

- Delays execution of fn until after delay milliseconds have passed since the last call.

clearTimeout:

- Resets the timer whenever the function is called again within the delay period.

Task 47: Write a throttle function that ensures a given function is executed at most once every specified interval.

Solution:

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

// Example Usage
const logScroll = () => console.log("Scroll event");

const throttledLog = throttle(logScroll, 1000);

window.addEventListener("scroll", throttledLog);
```

Explanation:

Concept:

- Ensures fn is called only once every interval milliseconds.

Date.now:

- Tracks the last execution time to control subsequent calls.

Task 48: Throttle the execution of a function that logs the scroll position to the console, ensuring it runs at most once every 200ms.

Solution:

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

const logScrollPosition = () => {
  console.log(`Scroll position: ${window.scrollY}`);
};

const throttledScroll = throttle(logScrollPosition, 200);

window.addEventListener("scroll", throttledScroll);
```

Explanation:

Throttling:

- Ensures logScrollPosition executes at most once every 200ms, even if the scroll event fires more frequently.

Task 49: Create a function that:

- **Debounces a user typing in a search bar.**
- **Throttles the API call to prevent it from being made more than once every second.**

Solution:

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

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

const fetchResults = (query) => {
  console.log(`Fetching results for: ${query}`);
};

const debouncedSearch = debounce((query) => throttledAPI(query), 300);
const throttledAPI = throttle(fetchResults, 1000);

// Simulate typing
debouncedSearch("JavaScript");
```

Explanation:

Debouncing:

- Waits until the user stops typing for 300ms before triggering the API call.

Throttling:

- Ensures the API call executes at most once every second.

Task 50: Fetch a list of users from the JSONPlaceholder API and log their names to the console.

Solution:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    data.forEach((user) => console.log(user.name));
  })
  .catch((error) => console.error("Error fetching data:", error));
```

Explanation:

1. **fetch**:
   - Makes a GET request to the API.

2. **Error Handling**:
   - Checks if the response status is OK (response.ok).
3. **Data Parsing**:
   - Converts the response to JSON format with response.json().

Task 51: Send a new post object to the JSONPlaceholder API and log the server's response.

```
const newPost = {
  title: "My New Post",
  body: "This is the content of the post.",
  userId: 1,
};
```

Solution:

```
const newPost = {
  title: "My New Post",
  body: "This is the content of the post.",
  userId: 1,
};

fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(newPost),
})
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => console.log("Server Response:", data))
  .catch((error) => console.error("Error posting data:", error));
```

Explanation

1. **Method**:
   - Specifies the HTTP method (POST).
2. **Headers**:
   - Defines the Content-Type as application/json.
3. **Body**:
   - Sends the data as a JSON string using JSON.stringify.