JavaScript Professional Practice Questions

Version: 1.0

1.What is the significance of JavaScript in modern web development, and how does it differ from Java?

- JavaScript is a high-level, dynamic programming language primarily used for client-side web development. It allows developers to create interactive and dynamic web content. Java, on the other hand, is a statically typed, compiled language often used for backend applications, Android development, and enterprise-level systems.

---

2. Write a JavaScript alert function that dynamically accepts user input and displays it with the message: "You entered: [user input]".

```javascript
const userInput = prompt("Enter something:");
alert(`You entered: ${userInput}`);
```

---

3. What are the key use cases of JavaScript in both client-side and server-side development?

- **Client-side:** DOM manipulation, event handling, form validation, animations, and dynamic content rendering.
- **Server-side:** Backend logic, API development, real-time data handling with frameworks like Node.js.

---

4. Identify and explain at least three major updates introduced in the latest version of ECMAScript.

- **Optional chaining (`?.`):** Simplifies access to deeply nested object properties.
- **Nullish coalescing (`??`):** Provides default values for `null` or `undefined`.
- **Top-level `await`:** Enables the use of `await` outside async functions in modules.

---

5. List and explain all JavaScript primitive data types, including their typical use cases.

- **String:** Textual data (`"hello"`).
- **Number:** Numeric values (`42`).
- **Boolean:** Logical values (`true` or `false`).
- **Null:** Intentional absence of value.
- **Undefined:** Uninitialized variable.
- **Symbol:** Unique, immutable property keys.
- **BigInt:** Arbitrarily large integers.

---

6. How does JavaScript handle non-integer numbers, and what are some common issues with floating-point arithmetic?

- JavaScript uses IEEE 754 for floating-point numbers, leading to rounding errors (e.g., `0.1 + 0.2 !== 0.3`). Use libraries like BigDecimal or `toFixed()` for precision.

---

7. What are the main differences between an integer in JavaScript and integers in strongly typed languages like Java?

- JavaScript does not differentiate between integers and floating-point numbers; both are `Number`. Java enforces types, and integers are distinct from floating-point numbers.

---

8. What are some scenarios where using floating-point numbers in JavaScript might cause unexpected results? Provide examples.

- Rounding errors in financial calculations:

```javascript
console.log(0.1 + 0.2); // 0.30000000000000004
```

---

9. Explain how JavaScript manages strings internally and discuss the impact of immutability on performance.

- Strings are immutable. Operations like concatenation create new strings, impacting memory and performance for large-scale manipulations.

---

10. What are the key characteristics and use cases of the Boolean data type in JavaScript?

- Represents logical `true` or `false`. Used in conditions, loops, and toggling states in applications.

---

11. Describe how 'undefined' is different from 'null' in JavaScript, with examples of when each might be encountered.

- `undefined`: Variable declared but not initialized.
- `null`: Explicitly assigned absence of value.

```javascript
let a; // undefined
let b = null; // null
```

---

12. What is the difference between the terms 'undefined' and 'undeclared' in JavaScript?

- **Undefined:** Declared variable without a value.
- **Undeclared:** Variable not declared; accessing it throws a `ReferenceError`.

---

13. How does JavaScript differentiate between 'null' and an empty object, and when should each be used?

- `null` represents no value. `{}` represents an empty but existing object.

---

14. Describe the Symbol data type and provide an example where it can be used to prevent property name collisions.

- Symbols create unique keys for object properties.

```javascript
const id = Symbol("id");
const obj = { [id]: 123 };
```

---

15. What are the limitations of the BigInt data type, and how does it integrate with other number types in JavaScript?

- Cannot mix `BigInt` with `Number`. Limited support for mathematical functions.

---

16. Write a function that demonstrates the difference between function declarations and function expressions in JavaScript.

```javascript
// Function declaration
function declared() {
    return "I'm declared";
}

// Function expression
const expressed = function () {
    return "I'm expressed";
};
```

---

17. What are the differences between parameters and arguments, and how can default parameters be utilized effectively?

- **Parameters:** Placeholders defined in the function declaration.
- **Arguments:** Actual values passed to the function when it is called.
- **Default parameters:** Used to set default values for parameters if no arguments are provided.

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

---

18. How do JavaScript functions handle varying numbers of arguments passed to them? Provide examples using the arguments object and rest parameters.

- `arguments` **object:** Array-like object for traditional functions.

```javascript
function sum() {
    let total = 0;
    for (const arg of arguments) total += arg;
    return total;
}
console.log(sum(1, 2, 3)); // 6
```

- **Rest parameters:** Modern ES6 syntax for collecting arguments.

```javascript
function sum(...args) {
    return args.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // 6
```

---

19. What would happen if you tried to use a return statement outside of a function?

- A `SyntaxError` would occur because `return` is only valid inside functions.

```javascript
return "Hello"; // SyntaxError: Illegal return statement
```

---

20. What are different ways to access HTML elements in JavaScript, and when would you use querySelector over getElementById?

- `getElementById`: Selects an element by ID (faster but limited to one element).
- `querySelector`: Selects the first matching element using a CSS selector (more versatile).

```javascript
document.getElementById("myId");
document.querySelector(".myClass");
```

---

21. Explain the purpose of the `<script>` tag and how attributes like 'defer' and 'async' affect JavaScript execution.

- `defer`: Scripts are loaded in order and executed after the HTML is parsed.
- `async`: Scripts are loaded independently and executed as soon as they're ready.

```html
<script src="script.js" defer></script>
<script src="script.js" async></script>
```

---

22. Write a JavaScript statement to display "Hello, how are you?" in the console and as part of an HTML element simultaneously.

```javascript
console.log("Hello, how are you?");
document.getElementById("output").innerText = "Hello, how are you?";
```

---

23. Describe the advantages and disadvantages of placing JavaScript code in the `<head>` vs the `<body>` of an HTML document.

- `<head>`:
  - Advantage: Allows scripts to be loaded before rendering.
  - Disadvantage: Blocks page rendering until the script loads.
- `<body>`:
  - Advantage: Improves initial page load time.
  - Disadvantage: May delay interactive features.

---

24. How do you include multiple external JavaScript files in an HTML document? Explain how loading order affects execution.

```html
<script src="file1.js"></script>
<script src="file2.js"></script>
```

- Scripts load and execute in the order they appear unless `async` or `defer` is used.

---

25. Explain the process of defining and invoking a named and an anonymous function in JavaScript.

- **Named function:**

```javascript
function greet() {
    return "Hello!";
}
greet();
```

- **Anonymous function:**

```javascript
const greet = function () {
    return "Hello!";
};
greet();
```

---

26. What is the main difference between variables declared with 'const' and 'let', beyond reassignment?

- `const` : Cannot be reassigned and must be initialized.

- `let` : Can be reassigned and declared without initialization.

---

27. Describe the block scope of a 'let' variable and how it prevents issues like variable hoisting.

- `let` is limited to the block it is declared in and avoids unintended access.

```javascript
{
    let x = 10;
}
console.log(x); // ReferenceError
```

---

28. Why does `typeof null` return 'object'? Is this behavior intentional or a bug in JavaScript?

- This is a bug in JavaScript due to legacy implementation. `null` is a primitive but `typeof null` returns `'object'`.

---

29. What are the two types of comments in JavaScript, and how do they impact code readability and execution?

- **Single-line:** `// Comment`
- **Multi-line:** `/* Comment */`
- Comments improve readability and are ignored during execution.

---

30. What is a recursive function, and how can JavaScript handle recursion efficiently without causing stack overflow?

- A function that calls itself.
- Use base cases and tail recursion for efficiency.

```javascript
function factorial(n) {
    if (n === 0) return 1;
    return n * factorial(n - 1);
}
```

---

31. What is the 'arguments' object in JavaScript, and how is it different from rest parameters in modern ES6?

- `arguments` : Array-like, available in traditional functions.
- **Rest parameters:** True array, preferred in ES6.

```javascript
function example(...args) {
    console.log(args); // [1, 2, 3]
}
```

---

32. What happens if you forget to include a 'break' statement in a JavaScript switch case? Demonstrate with an example.

- Without `break` , cases will "fall through" to subsequent cases.

```javascript
switch (1) {
    case 1:
        console.log("One");
```

```
    case 2:
        console.log("Two");
}
// Output: "One" "Two"
```

---

33. Which loops in JavaScript are best suited for iterating over objects, and why?

- `for...in`: Iterates over object keys.
- `Object.keys()` with `forEach`: More control and better performance.

---

34. How does the `push()` method behave when used on an array with a defined length?

- It appends to the end and updates the length.

---

35. What is the difference between dot notation and bracket notation when accessing object properties in JavaScript?

- Dot notation: `obj.key` (only for valid identifiers).
- Bracket notation: `obj["key"]` (works with dynamic keys).

---

36. Write a `for` loop that iterates backward through an array and prints its elements to the console.

```javascript
const arr = [1, 2, 3, 4];
for (let i = arr.length - 1; i >= 0; i--) {
    console.log(arr[i]);
}
```

---

37. How do you use the `length` property of an array to iterate over it dynamically, accounting for array mutation?

- Use a `for` loop with the `length` property to handle dynamic updates:

```javascript
const arr = [1, 2, 3];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
    arr.push(i + 4); // Demonstrates mutation
}
```

---

38. Explain the difference between the `==` and `===` operators in JavaScript, and when strict equality should be used.

- `==` compares values with type coercion.
- `===` compares values without type coercion.

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

---

39. What is the difference between the `pop()` and `splice()` methods for array manipulation in JavaScript?

- `pop()`: Removes the last element.
- `splice()`: Removes or adds elements at specific indices.

```javascript
let arr = [1, 2, 3];
arr.pop(); // [1, 2]
arr.splice(1, 1); // [1]
```

---

40. What are some pitfalls of using `parseInt()` to convert strings to numbers, and how can they be avoided?

- Stops parsing at the first invalid character.
- May produce incorrect results with leading zeros.

```javascript
parseInt('08'); // 8 in most cases, but may be 0 in older browsers.
// Avoid by specifying radix:
parseInt('08', 10); // 8
```

---

41. What is the result of `'10' - '5' + 5` in JavaScript, and why?

- Result: `10`

```javascript
console.log('10' - '5' + 5); // '10' and '5' are coerced to numbers.
```

---

42. How do you efficiently access the last element of an array without knowing its length?

```javascript
const arr = [1, 2, 3];
console.log(arr[arr.length - 1]); // 3
```

---

43. Write a function to check if a given input is an array using multiple methods in JavaScript.

```javascript
function isArray(input) {
    return Array.isArray(input);
}
console.log(isArray([1, 2, 3])); // true
```

---

44. What is the difference between the `slice()` and `splice()` methods in JavaScript arrays?

- `slice()`: Returns a shallow copy without modifying the original array.
- `splice()`: Modifies the array by removing or adding elements.

---

45. How can you find both the highest and lowest values in an array without using built-in methods?

```javascript
const arr = [3, 1, 4];
let min = arr[0], max = arr[0];
```

```
for (const num of arr) {
    if (num < min) min = num;
    if (num > max) max = num;
}
console.log({ min, max });
```

---

46. Explain how JavaScript distinguishes between `undefined` and uninitialized variables.

- `undefined`: Variable declared but not assigned.
- **Uninitialized:** Accessing undeclared variables throws a `ReferenceError`.

---

47. Demonstrate how `parseFloat()` handles strings with both numeric and non-numeric characters.

javascript

```
console.log(parseFloat('12.34px')); // 12.34
console.log(parseFloat('abc12.34')); // NaN
```

---

48. Write a program to concatenate three strings using both the '+' operator and the `concat()` method.

javascript

```
let str1 = "Hello";
let str2 = "World";
let str3 = "!";
console.log(str1 + " " + str2 + str3);
console.log(str1.concat(" ", str2, str3));
```

49. What issues can arise from using `NaN` in JavaScript calculations, and how can you check for it?

- **Issues:** Propagation of `NaN` in calculations.
- **Check:** Use `Number.isNaN()` or `isNaN()`.

```javascript
console.log(Number.isNaN(NaN)); // true
console.log(isNaN('text')); // true
```

---

50. What does accessing a non-existent property of an object return, and how can you safeguard against it?

- Returns `undefined`. Use optional chaining to safeguard:

```javascript
const obj = {};
console.log(obj?.property); // undefined
```

---

51. Demonstrate how the `in` operator and `hasOwnProperty()` differ in object property checking.

- `in`: Checks for properties in the prototype chain.
- `hasOwnProperty()`: Checks only own properties.

```javascript
const obj = { key: 1 };
console.log('key' in obj); // true
console.log(obj.hasOwnProperty('key')); // true
```

52. How does the `map()` function transform arrays in JavaScript, and what are its advantages over `forEach()`?

- `map()`: Returns a new array with transformed values.
- **Advantages:** Functional programming and immutability.

```javascript
const arr = [1, 2, 3];
const doubled = arr.map(x => x * 2);
console.log(doubled); // [2, 4, 6]
```

---

53. Explain how JavaScript manages asynchronous operations using callbacks, promises, and async/await syntax. Provide examples.

- **Callbacks:** Functions passed to handle async results.

```javascript
setTimeout(() => console.log("Done"), 1000);
```

- **Promises:** Handles chaining and errors.

```javascript
fetch(url).then(response => response.json());
```

- `async/await`: Cleaner syntax.

```javascript
const fetchData = async () => {
    const data = await fetch(url);
    return await data.json();
};
```

54. What is the difference between synchronous and asynchronous execution in JavaScript?

- **Synchronous:** Code runs sequentially; each statement waits for the previous one to finish.
- **Asynchronous:** Code executes independently; long-running tasks do not block the main thread.

```javascript
console.log("Start");
setTimeout(() => console.log("Async Task"), 1000);
console.log("End");
// Output: Start, End, Async Task
```

---

55. Write a JavaScript function that fetches data from an API using both `then` and `async/await`. Compare their readability.

- **Using `then`:**

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

- **Using `async/await`:**

```javascript
```

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

---

56. What is the purpose of the event loop in JavaScript, and how does it handle tasks like setTimeout or promises?

- The event loop ensures that the main thread processes tasks in the call stack. Tasks like `setTimeout` and promises are handled in the **task queue** or **microtask queue** and executed when the stack is clear.

---

57. What are the potential issues with nested callbacks in JavaScript, and how can they be resolved?

- **Issues:** Callback hell, hard-to-read code, and error handling difficulties.
- **Solution:** Use promises or `async/await` for better structure.

---

58. How does the `Promise.all()` method work, and what happens if one of the promises fails?

- `Promise.all()` runs all promises in parallel and resolves when all succeed. If one fails, it rejects immediately.

59. What is the purpose of the `finally` block in a promise chain, and how is it used?

- **Purpose:** Executes code after a promise resolves or rejects.

```javascript
fetch(url)
    .then(response => response.json())
    .catch(err => console.error(err))
    .finally(() => console.log("Fetch completed."));
```

---

60. Describe the difference between a microtask and a macrotask in JavaScript. Provide examples of each.

- **Microtasks:** Processed before macrotasks (e.g., `Promise.then`).
- **Macrotasks:** Scheduled for the next event loop iteration (e.g., `setTimeout`).

```javascript
Promise.resolve().then(() => console.log("Microtask"));
setTimeout(() => console.log("Macrotask"), 0);
console.log("Synchronous");
// Output: Synchronous, Microtask, Macrotask
```

---

61. What is the purpose of the `fetch` API in JavaScript, and how does it improve upon the older `XMLHttpRequest`?

- **Purpose:** Simplifies HTTP requests.
- **Advantages:** Promises-based, cleaner syntax, better error handling.

---

62. Write a JavaScript program to debounce a function, explaining how debouncing improves performance.

- **Debounce:** Delays function execution until a specified time has passed since the last call.

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

const log = debounce(() => console.log("Debounced!"), 300);
window.addEventListener("resize", log);
```

---

63. What is throttling in JavaScript, and how does it differ from debouncing? Provide use cases for each.

- **Throttling:** Ensures a function is called at most once in a given interval.
- **Use cases:**
  - **Debouncing:** Search bar input.
  - **Throttling:** Scroll event handling.

---

64. Explain how closures in JavaScript work, and provide an example of a practical use case.

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

```javascript
```

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

---

65. What is the significance of the `this` keyword in JavaScript, and how does its value change depending on the context?

- **Significance:** Refers to the object invoking the function.
- **Examples:**

```javascript
const obj = {
    value: 42,
    getValue() {
        return this.value;
    },
};
console.log(obj.getValue()); // 42
const getValue = obj.getValue;
console.log(getValue()); // undefined
```

---

66. Write a function that demonstrates how `bind()`, `call()`, and `apply()` are used in JavaScript.

```javascript
```

```
function greet(greeting, name) {
    return `${greeting}, ${name}`;
}
const boundGreet = greet.bind(null, "Hello");
console.log(boundGreet("Alice")); // Hello, Alice
console.log(greet.call(null, "Hi", "Bob")); // Hi, Bob
console.log(greet.apply(null, ["Hey", "Charlie"])); // Hey, Charlie
```

---

67. What are arrow functions, and how do they differ from regular function expressions in terms of syntax and behavior?

- **Arrow functions:**
  - Shorter syntax.
  - Lexical `this` (no own `this`).

javascript

```
const add = (a, b) => a + b;
const obj = {
    value: 10,
    getValue: () => this.value, // 'this' is inherited
};
console.log(obj.getValue()); // undefined
```

---

68. Explain the concept of hoisting in JavaScript. How does it apply to variables, functions, and classes?

- **Hoisting:** Moves declarations to the top of their scope.
  - Functions are fully hoisted.
  - `var` is hoisted without initialization.
  - `let` and `const` are hoisted but uninitialized.

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

---

69. What is the difference between global and local scope in JavaScript, and how does `var` behave differently from `let` and `const`?

- **Global scope:** Accessible everywhere.
- **Local scope:** Limited to the function or block.
- `var` has function scope, while `let` and `const` have block scope.

---

70. Explain how block scoping works in JavaScript and provide examples of its usage.

- Variables declared with `let` or `const` are scoped to the block.

```javascript
{
    let x = 10;
    console.log(x); // 10
}
console.log(x); // ReferenceError
```

---

71. What is the purpose of IIFE (Immediately Invoked Function Expressions) in JavaScript, and how are they written?

- **Purpose:** Avoid polluting the global scope by creating a temporary scope for variables.
- **Syntax:**

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

---

72. How does JavaScript handle exceptions using `try`, `catch`, `finally`, and `throw`? Provide examples.

- `try`: Executes code that might throw errors.
- `catch`: Handles errors.
- `finally`: Executes after `try` or `catch`, regardless of the result.
- `throw`: Manually throws an error.

```javascript
try {
    throw new Error("Something went wrong!");
} catch (err) {
    console.log(err.message);
} finally {
    console.log("Execution completed.");
}
```

---

73. What is the difference between type coercion and type conversion in JavaScript? Provide examples of each.

- **Type coercion:** Implicit conversion by JavaScript.

```javascript
console.log("5" + 2); // "52"
console.log("5" * 2); // 10
```

- **Type conversion:** Explicit conversion by the developer.

```javascript
console.log(Number("5") + 2); // 7
```

---

74. What is the result of the following code, and why? `console.log([] + {})` and `console.log({} + [])`

- `[] + {}`: String concatenation, `"[object Object]"`.
- `{} + []`: Treated as a block, `0` in some contexts.

---

75. How do `for...of` and `for...in` loops differ in JavaScript? Provide scenarios where each is preferable.

- `for...of`: Iterates over iterable objects like arrays.

```javascript
for (const value of [1, 2, 3]) console.log(value);
```

- `for...in`: Iterates over object keys.

```javascript
const obj = { a: 1, b: 2 };
for (const key in obj) console.log(key);
```

76. What is destructuring in JavaScript, and how can it be used with arrays and objects?

- **Destructuring:** Unpacks values from arrays or properties from objects.

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

---

77. Explain the spread operator in JavaScript and provide examples of how it is used with arrays and objects.

- **Purpose:** Expands arrays or objects.

```javascript
const arr1 = [1, 2];
const arr2 = [...arr1, 3]; // [1, 2, 3]

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

---

78. What is the rest parameter in JavaScript, and how does it differ from the spread operator?

- **Rest parameter:** Collects arguments into an array.

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

- **Spread operator:** Expands arrays/objects.

---

79. Write a JavaScript function to flatten a deeply nested array using recursion or the `flat()` method.

```javascript
function flatten(arr) {
    return arr.reduce(
        (flat, val) =>
            flat.concat(Array.isArray(val) ? flatten(val) : val),
        []
    );
}
console.log(flatten([1, [2, [3, 4]]])); // [1, 2, 3, 4]
```

---

80. What is the difference between deep and shallow copying in JavaScript? Provide examples of both.

- **Shallow copy:** Only top-level properties are copied.

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

- **Deep copy:** All levels are copied.

  ```javascript
  const deepCopy = JSON.parse(JSON.stringify(obj1));
  ```

---

81. Explain how JavaScript's `setTimeout` and `setInterval` functions work. What are their limitations?

- `setTimeout`: Delays execution of a function.
- `setInterval`: Repeatedly executes a function.
- **Limitations:** Delays may vary due to the event loop.

---

82. What is the purpose of the `clearInterval` method in JavaScript? Provide an example.

- Stops an interval.

```javascript
const intervalId = setInterval(() => console.log("Running"), 1000);
setTimeout(() => clearInterval(intervalId), 5000);
```

---

83. Write a program that uses `setTimeout` to display a series of messages with increasing delays.

```javascript
const messages = ["Hello", "How are you?", "Goodbye"];
messages.forEach((msg, i) => {
    setTimeout(() => console.log(msg), i * 1000);
});
```

---

84. What is a JavaScript generator function, and how is it different from a regular function? Provide a use case.

- **Generator function:** Can pause and resume execution.

```javascript
function* gen() {
    yield 1;
    yield 2;
}
const it = gen();
console.log(it.next().value); // 1
console.log(it.next().value); // 2
```

---

85. How does the `yield` keyword work in JavaScript generators? Provide an example with a generator that produces Fibonacci numbers.

```javascript
function* fibonacci() {
    let [prev, curr] = [0, 1];
    while (true) {
        yield curr;
        [prev, curr] = [curr, prev + curr];
    }
}
const fib = fibonacci();
console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
console.log(fib.next().value); // 2
```

---

86. What is the purpose of JavaScript's `Symbol.iterator`, and how is it implemented in custom objects?

- Enables iteration over custom objects.

```javascript
```

```
const obj = {
    *[Symbol.iterator]() {
        yield 1;
        yield 2;
    },
};
for (const val of obj) console.log(val);
```

---

87. Explain the difference between `Object.keys()`, `Object.values()`, and `Object.entries()`. Provide examples.

- `Object.keys()`: Returns an array of object keys.
- `Object.values()`: Returns an array of object values.
- `Object.entries()`: Returns an array of key-value pairs.

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

---

88. What are JavaScript's WeakMap and WeakSet, and how do they differ from regular Map and Set?

- **WeakMap:** Key-value pairs with keys as objects, keys are weakly referenced.
- **WeakSet:** Collection of unique objects, objects are weakly referenced.
- Weak references prevent memory leaks.

```javascript
let obj = { key: 1 };
const weakMap = new WeakMap();
weakMap.set(obj, "value");
obj = null; // Object is garbage collected.
```

---

89. How do you handle deep equality checks in JavaScript objects, given that `==` and `===` do not suffice?

- Use a recursive function or a library like Lodash:

```javascript
function deepEqual(obj1, obj2) {
    if (obj1 === obj2) return true;
    if (typeof obj1 !== "object" || typeof obj2 !== "object") return false;
    const keys1 = Object.keys(obj1);
    const keys2 = Object.keys(obj2);
    if (keys1.length !== keys2.length) return false;
    return keys1.every(key => deepEqual(obj1[key], obj2[key]));
}
```

---

90. Explain how the JavaScript `class` keyword works. How is it different from function constructors?

- `class` is syntactic sugar for function constructors, providing cleaner syntax and built-in inheritance mechanisms.

```javascript
class Person {
    constructor(name) {
        this.name = name;
    }
    greet() {
```

```
        return `Hello, ${this.name}`;
    }
}
const person = new Person("Alice");
console.log(person.greet()); // Hello, Alice
```

---

91. What are getter and setter methods in JavaScript classes, and how are they used?

- **Getters:** Define read-only properties.
- **Setters:** Define custom logic for setting property values.

javascript

```
class Person {
    constructor(name) {
        this._name = name;
    }
    get name() {
        return this._name.toUpperCase();
    }
    set name(newName) {
        this._name = newName;
    }
}
const person = new Person("Alice");
console.log(person.name); // ALICE
person.name = "Bob";
console.log(person.name); // BOB
```

---

92. Write a JavaScript class that implements a simple counter with increment, decrement, and reset methods.

javascript

```
class Counter {
    constructor() {
        this.count = 0;
    }
    increment() {
        this.count++;
    }
    decrement() {
        this.count--;
    }
    reset() {
        this.count = 0;
    }
}
const counter = new Counter();
counter.increment();
console.log(counter.count); // 1
```

---

93. How does inheritance work in JavaScript classes, and how does it differ from prototype-based inheritance?

- **Class inheritance:** Uses the `extends` keyword for cleaner syntax.
- **Prototype-based inheritance:** Directly manipulates the prototype.

```javascript
class Animal {
    speak() {
        return "Generic sound";
    }
}
class Dog extends Animal {
    speak() {
        return "Bark";
    }
}
```

```
const dog = new Dog();
console.log(dog.speak()); // Bark
```

---

94. What is the difference between `Object.create()` and using a `class` to create objects in JavaScript?

- `Object.create()`: Creates objects directly with specified prototypes.
- `class`: Constructs objects and provides structure.

```javascript
const proto = { greet() { return "Hello"; } };
const obj = Object.create(proto);
console.log(obj.greet()); // Hello
```

---

95. Explain the use of `super` in JavaScript. How is it used in class constructors and methods?

- `super`: Calls the parent class constructor or methods.

```javascript
class Animal {
    constructor(name) {
        this.name = name;
    }
}
class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Calls Animal's constructor
        this.breed = breed;
    }
}
```

```
const dog = new Dog("Rex", "Labrador");
console.log(dog.name); // Rex
```

---

96. What is the difference between event bubbling and event capturing in JavaScript? How can you manage them?

- **Event bubbling:** Events propagate from child to parent.
- **Event capturing:** Events propagate from parent to child.
- Use `addEventListener` with the `capture` option to manage:

```javascript
element.addEventListener("click", handler, { capture: true });
```

---

97. Explain the concept of delegation in JavaScript event handling. Why is it useful?

- **Delegation:** Attaching a single event listener to a parent element to handle events for child elements.
- **Benefits:** Improves performance and handles dynamic elements.

```javascript
document.getElementById("parent").addEventListener("click", e => {
    if (e.target.tagName === "BUTTON") {
        console.log("Button clicked!");
    }
});
```

---

98. What are custom events in JavaScript, and how can they be created and dispatched?

- Custom events are user-defined events created with the `CustomEvent` constructor.

```javascript
const event = new CustomEvent("myEvent", { detail: { message: "Hello!" } });
document.dispatchEvent(event);
document.addEventListener("myEvent", e => {
    console.log(e.detail.message); // Hello!
});
```

---

99. Write a JavaScript function that clones an object, ensuring all nested properties are copied.

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

---

100. What is a proxy in JavaScript, and how can it be used to intercept operations on objects?

- Proxies wrap objects and intercept operations like property access or assignment.

```javascript
const obj = { a: 1 };
const proxy = new Proxy(obj, {
    get(target, prop) {
        return prop in target ? target[prop] : "Property not found";
```

```
    },
});
console.log(proxy.a); // 1
console.log(proxy.b); // Property not found
```

---

101. How does the `Reflect` API complement JavaScript proxies? Provide an example.

- **Reflect API:** Provides methods to manipulate objects. Often used in proxies to forward operations to the target object.

```javascript
const obj = { a: 1 };
const proxy = new Proxy(obj, {
    get(target, prop) {
        console.log(`Getting ${prop}`);
        return Reflect.get(target, prop);
    },
});
console.log(proxy.a); // Logs: "Getting a", Output: 1
```

---

102. What is the purpose of the `Object.freeze()` method in JavaScript, and how does it compare to `Object.seal()`?

- `Object.freeze()`: Makes an object immutable (no additions, deletions, or modifications).
- `Object.seal()`: Prevents additions or deletions but allows modification of existing properties.

```javascript
```

```javascript
const obj = { a: 1 };
Object.freeze(obj);
obj.a = 2; // No effect
console.log(obj.a); // 1
```

---

103. How can you check if an object is frozen in JavaScript?

- Use `Object.isFrozen()`:

```javascript
const obj = { a: 1 };
Object.freeze(obj);
console.log(Object.isFrozen(obj)); // true
```

---

104. What is the `Intl` object in JavaScript, and how can it be used for internationalization?

- `Intl` **Object:** Provides functionality for formatting numbers, dates, and strings according to locale.

```javascript
const formatter = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' });
console.log(formatter.format(1234.56)); // $1,234.56
```

---

105. Explain how JavaScript's `Date` object can be used to manipulate and format dates and times.

- The `Date` object represents dates and times. Methods:

- `getDate()`, `getMonth()`, `getFullYear()`: Retrieve date parts.
- `setDate()`, `setMonth()`, `setFullYear()`: Modify date parts.

```javascript
const date = new Date();
console.log(date.toISOString()); // e.g., 2023-05-01T12:00:00.000Z
```

---

106. What are JavaScript modules, and how are `import` and `export` used?

- Modules organize code into reusable files. Use `export` to expose code and `import` to include it.

```javascript
// file1.js
export const greet = () => "Hello";

// file2.js
import { greet } from './file1.js';
console.log(greet()); // Hello
```

---

107. Explain the difference between named and default exports in JavaScript modules.

- **Named export:** Export multiple values by name.
- **Default export:** Export a single value without a name.

```javascript
// Named export
export const a = 1;
export const b = 2;
```

```
// Default export
export default function () {
    return "Default Export";
}
```

---

108. What is tree-shaking, and how does it optimize JavaScript module bundling?

- Tree-shaking removes unused code from bundled files. It works with ES6 modules to optimize file size.

---

109. What is the difference between mutable and immutable data in JavaScript? Provide examples.

- **Mutable:** Data that can be changed (e.g., objects, arrays).

```javascript
const arr = [1, 2];
arr.push(3); // Mutated
```

- **Immutable:** Data that cannot be changed (e.g., primitives, frozen objects).

---

110. How does JavaScript handle memory management, and what is the role of garbage collection?

- JavaScript uses automatic garbage collection to reclaim memory from unreachable objects. The engine uses algorithms like **mark-and-sweep**.

---

111. What are the advantages of using functional programming techniques in JavaScript?

- Benefits:
  - Code is modular and predictable.
  - Avoids side effects.
  - Promotes immutability.

```javascript
const double = x => x * 2;
const numbers = [1, 2, 3];
const doubled = numbers.map(double);
```

---

112. Explain the concept of pure functions in JavaScript. Why are they important in functional programming?

- **Pure function:** Produces the same output for the same input without side effects.

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

- Importance: Easier to test and debug.

---

113. Write a JavaScript program that uses higher-order functions like `filter()`, `map()`, and `reduce()`.

```javascript
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(n => n % 2 === 0);
const doubled = evens.map(n => n * 2);
```

```javascript
const sum = doubled.reduce((total, n) => total + n, 0);
console.log(sum); // 12
```

---

114. What is the purpose of JavaScript's `Object.assign()` method? How does it handle conflicts between objects?

- Merges objects and handles conflicts by using the last source object's values.

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

---

115. How does JavaScript handle immutability with objects and arrays?

- Use `Object.freeze()` or libraries like **Immutable.js** to ensure immutability.

```javascript
const obj = Object.freeze({ a: 1 });
obj.a = 2; // No effect
```

---

116. Write a JavaScript function to debounce a button click event to prevent multiple form submissions.

```javascript
function debounce(func, delay) {
    let timer;
```

```
    return function (...args) {
        clearTimeout(timer);
        timer = setTimeout(() => func(...args), delay);
    };
}
const submit = debounce(() => console.log("Form submitted"), 500);
document.getElementById("button").addEventListener("click", submit);
```

---

117. How does JavaScript handle asynchronous file reading in Node.js? Provide an example using `fs.promises`.

javascript

```
const fs = require('fs/promises');
async function readFile(filePath) {
    try {
        const data = await fs.readFile(filePath, 'utf8');
        console.log(data);
    } catch (err) {
        console.error(err);
    }
}
readFile('example.txt');
```

---

118. What are the key differences between Node.js `require` and ES6 `import`?

- `require`: CommonJS syntax, synchronous, supports dynamic imports.
- `import`: ES6 syntax, asynchronous, supports static analysis.

---

119. Explain the purpose of npm (Node Package Manager) and how it helps in JavaScript development.

- **npm:** Manages dependencies, versions, and scripts for JavaScript projects.

```bash
npm install lodash
```

---

120. How can you use WebSockets to implement real-time communication in a JavaScript application?

```javascript
const socket = new WebSocket("ws://example.com");
socket.onopen = () => console.log("Connected");
socket.onmessage = event => console.log(event.data);
socket.send("Hello, Server!");
```

---

121. What are the differences between HTTP and WebSockets in terms of communication protocols?

- **HTTP:** Request-response model; unidirectional communication.
- **WebSockets:** Full-duplex, persistent connection; bidirectional communication.

---

122. Write a JavaScript function that validates an email address using a regular expression.

```javascript
function validateEmail(email) {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(email);
}
console.log(validateEmail("test@example.com")); // true
```

---

123. What are regular expressions in JavaScript, and how are they used for pattern matching?

- **Regular expressions:** Patterns for matching text.

```javascript
const regex = /\d+/;
console.log(regex.test("123")); // true
```

---

124. How does JavaScript handle promises in a sequence (chaining)? Write an example that uses multiple `.then()` calls.

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

---

125. What is the difference between `Promise.race()` and `Promise.any()`? Provide examples.

- `Promise.race()`: Resolves/rejects with the first settled promise.
- `Promise.any()`: Resolves with the first fulfilled promise; rejects if all promises fail.

```javascript
Promise.race([fetch(url1), fetch(url2)]).then(console.log);
Promise.any([fetch(url1), fetch(url2)]).then(console.log);
```

---

126. Explain the concept of memoization in JavaScript and write a function to demonstrate it.

- **Memoization:** Caching results of expensive function calls.

```javascript
function memoize(fn) {
    const cache = {};
    return function (...args) {
        const key = JSON.stringify(args);
        if (key in cache) return cache[key];
        return (cache[key] = fn(...args));
    };
}
const fib = memoize(n => (n <= 1 ? n : fib(n - 1) + fib(n - 2)));
console.log(fib(10)); // 55
```

---

127. What is the purpose of the `reduce()` method in JavaScript arrays, and how does it differ from `map()`?

- `reduce()`: Aggregates array values into a single value.
- `map()`: Transforms array elements into a new array.

```javascript
```

```
const arr = [1, 2, 3];
console.log(arr.reduce((sum, val) => sum + val, 0)); // 6
console.log(arr.map(x => x * 2)); // [2, 4, 6]
```

---

128. How does JavaScript handle recursion when calculating factorials? Write an example program.

javascript

```
function factorial(n) {
    return n === 0 ? 1 : n * factorial(n - 1);
}
console.log(factorial(5)); // 120
```

---

129. Write a program that simulates a stopwatch using JavaScript's `setInterval` function.

javascript

```
let seconds = 0;
const intervalId = setInterval(() => {
    console.log(`Elapsed time: ${++seconds} seconds`);
    if (seconds === 10) clearInterval(intervalId);
}, 1000);
```

---

130. What is event delegation in JavaScript, and how does it improve performance in dynamic DOM manipulation?

- Event delegation attaches a listener to a parent element, handling events for child elements dynamically.

```javascript
document.getElementById("parent").addEventListener("click", e => {
    if (e.target.tagName === "BUTTON") console.log("Button clicked!");
});
```

---

131. Explain the concept of the Virtual DOM. How does it differ from the real DOM?

- **Virtual DOM:** In-memory representation of the real DOM. Changes are batched and updated efficiently.
- **Real DOM:** Directly updates the UI, which is slower.

---

132. What are JavaScript promises, and how do they solve the problem of callback hell?

- Promises handle async operations more cleanly, allowing chaining and better error handling.

```javascript
fetch(url)
    .then(res => res.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));
```

---

133. Write a JavaScript function that uses `Promise.allSettled()` to handle multiple asynchronous tasks.

```javascript
```

```
const promises = [
    Promise.resolve("Success"),
    Promise.reject("Error"),
];
Promise.allSettled(promises).then(results =>
    results.forEach(res => console.log(res))
);
```

---

134. Explain the difference between `undefined` and `not defined` in JavaScript. Provide examples.

- `undefined`: Variable declared but not initialized.
- `not defined`: Variable never declared; results in a `ReferenceError`.

```javascript
let a;
console.log(a); // undefined
console.log(b); // ReferenceError: b is not defined
```

---

135. How does JavaScript handle shadowing in block scopes with variables declared using `let` and `const`?

- Variables in inner scopes can shadow outer scope variables.

```javascript
let x = 1;
{
    let x = 2;
    console.log(x); // 2
}
console.log(x); // 1
```

136. What is the difference between the `filter()` and `find()` methods in JavaScript arrays?

- `filter()`: Returns all matching elements.
- `find()`: Returns the first matching element.

```javascript
const arr = [1, 2, 3];
console.log(arr.filter(n => n > 1)); // [2, 3]
console.log(arr.find(n => n > 1)); // 2
```

---

137. Explain how the `sort()` method works in JavaScript, and why it requires a compare function for numbers.

- ** `sort()` sorts elements lexicographically by default.
- Use a compare function for numerical sorting:

```javascript
const arr = [3, 1, 2];
arr.sort((a, b) => a - b); // [1, 2, 3]
```

---

138. Write a function to remove duplicate elements from an array using JavaScript.

```javascript
const unique = arr => [...new Set(arr)];
console.log(unique([1, 2, 2, 3])); // [1, 2, 3]
```

---

139. What is the purpose of the `Promise.prototype.finally()` method, and how is it used?

- Executes code after a promise is settled, regardless of outcome.

```javascript
fetch(url)
    .then(data => console.log(data))
    .catch(err => console.error(err))
    .finally(() => console.log("Done"));
```

---

140. What are JavaScript's optional chaining and nullish coalescing operators? Provide examples.

- **Optional chaining (`?.`)**: Safely access deeply nested properties.
- **Nullish coalescing (`??`)**: Provide a default value for `null` or `undefined`.

```javascript
const obj = { a: { b: 1 } };
console.log(obj?.a?.b); // 1
console.log(obj?.a?.c ?? "Default"); // Default
```

---

141. How does JavaScript handle immutability with `Object.freeze()`? What are its limitations?

- `Object.freeze()` makes an object immutable by preventing additions, deletions, or changes to existing properties.

- **Limitations:** Does not apply to nested objects (shallow freeze).

```javascript
const obj = Object.freeze({ a: 1, b: { c: 2 } });
obj.b.c = 3; // Allowed since `b` is not deeply frozen.
console.log(obj.b.c); // 3
```

---

142. Write a JavaScript function to merge two sorted arrays into a single sorted array.

```javascript
function mergeSortedArrays(arr1, arr2) {
    let merged = [];
    let i = 0, j = 0;
    while (i < arr1.length && j < arr2.length) {
        if (arr1[i] < arr2[j]) merged.push(arr1[i++]);
        else merged.push(arr2[j++]);
    }
    return merged.concat(arr1.slice(i), arr2.slice(j));
}
console.log(mergeSortedArrays([1, 3, 5], [2, 4, 6])); // [1, 2, 3, 4, 5, 6]
```

---

143. What is the purpose of the `instanceof` operator in JavaScript? How does it work?

- **Purpose:** Checks if an object is an instance of a specific constructor.

```javascript
class Person {}
const person = new Person();
console.log(person instanceof Person); // true
console.log(person instanceof Object); // true
```

144. Explain the difference between shallow and deep cloning in JavaScript. Write a function for deep cloning.

- **Shallow clone:** Copies only the top-level properties.
- **Deep clone:** Copies all nested properties.

```javascript
function deepClone(obj) {
    return JSON.parse(JSON.stringify(obj));
}
const obj = { a: 1, b: { c: 2 } };
const clone = deepClone(obj);
clone.b.c = 3;
console.log(obj.b.c); // 2
```

---

145. How does JavaScript handle `try...catch` with asynchronous operations? Provide examples with `async/await`.

- Use `try...catch` to handle errors in `async/await` functions:

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

---

146. What is a JavaScript promise chain, and how does it handle errors with `.catch()`?

- **Promise chain:** Sequence of `.then()` calls where each step depends on the previous one. `.catch()` handles errors in the entire chain.

```javascript
fetch(url)
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.error("Error:", err));
```

---

147. What is the purpose of the `eval()` function in JavaScript, and why is it considered unsafe?

- **Purpose:** Executes a string as JavaScript code.
- **Unsafe:** Allows execution of malicious code, hard to debug, and impacts performance.

```javascript
eval("console.log('Dangerous!')");
```

---

148. Write a JavaScript program to implement a simple calculator using HTML and JavaScript.

```html
<input id="num1" type="number" placeholder="Enter first number">
<input id="num2" type="number" placeholder="Enter second number">
<button onclick="calculate()">Add</button>
<p id="result"></p>

<script>
    function calculate() {
```

```
        const num1 = parseFloat(document.getElementById("num1").value);
        const num2 = parseFloat(document.getElementById("num2").value);
        document.getElementById("result").innerText = `Sum: ${num1 + num2}`;
    }
</script>
```

---

149. How does JavaScript handle the `this` context inside event listeners? How can it be managed explicitly?

- By default, `this` refers to the element the listener is attached to.
- Explicitly bind `this` using `.bind()` or arrow functions:

```javascript
const obj = {
    value: 42,
    showValue() {
        console.log(this.value);
    },
};
document.getElementById("btn").addEventListener("click",
obj.showValue.bind(obj));
```

---

150. What is the difference between `document.ready` and `window.onload` in JavaScript? When should each be used?

- `document.ready`: Executes when the DOM is fully loaded but before resources like images are loaded. (Often used with jQuery.)
- `window.onload`: Executes after all resources are loaded.

```javascript
document.addEventListener("DOMContentLoaded", () => {
    console.log("DOM fully loaded");
```

```
});
window.onload = () => {
    console.log("Everything loaded");
};
```
