25 Advanced Vanilla JavaScript Coding Questions

JavaScript has evolved dramatically over the years. Beyond the basics lie powerful, nuanced topics that are essential for developing efficient, scalable, and robust applications. In this post, we explore 25 advanced JavaScript coding questions, each accompanied by an in-depth explanation and code examples to illustrate key concepts.

1. How Does the JavaScript Event Loop Handle Microtasks vs. Macrotasks?

Answer:
The event loop in JavaScript manages asynchronous tasks using two queues: the microtask queue (for promises, MutationObserver callbacks, etc.) and the macrotask queue (for setTimeout, setInterval, I/O events, etc.). After executing the current stack, the event loop first processes all microtasks before moving to the next macrotask. This prioritization ensures that promise callbacks run as soon as possible.

Example:

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

setTimeout(() => console.log('Macrotask'), 0);

Promise.resolve().then(() => console.log('Microtask'));

console.log('End');

// Output order: Start, End, Microtask, Macrotask
```

2. What Are Memory Leaks in JavaScript and How Can You Prevent Them?

Answer:
Memory leaks occur when unused memory isn't released. Common causes include global variables, forgotten timers or intervals, detached DOM nodes, and unintended closures that keep references alive. Prevent leaks by cleaning up timers, removing event listeners when elements are removed, and using tools (like Chrome DevTools) to profile memory usage.

3. Explain Prototypal Inheritance in Depth

Answer:
JavaScript's inheritance is based on prototypes. Each object has a hidden internal property (often accessible via `__proto__`) that points to its prototype. When a property or method is accessed, JavaScript looks up the prototype chain until it finds the property or reaches the end. `Object.create()` creates a new object with a specified prototype, making prototypal inheritance explicit.

Example:

```
const parent = {

  greet() {

    return `Hello from ${this.name}`;

  }

};

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

```
child.name = 'Child';

console.log(child.greet()); // "Hello from Child"
```

4. What Are the Differences Between `call`, `apply`, and `bind`?

Answer:

- **`call`:** Invokes a function immediately with a specified `this` and arguments passed individually.
- **`apply`:** Similar to `call` but accepts arguments as an array.
- **`bind`:** Returns a new function with a bound `this` and optionally preset arguments, without immediately invoking it.

Example:

```
function showDetails(age, city) {

  return `${this.name} is ${age} years old and lives in ${city}.`;

}

const person = { name: "Alice" };

console.log(showDetails.call(person, 30, "New York"));

console.log(showDetails.apply(person, [30, "New York"]));

const boundShowDetails = showDetails.bind(person, 30);

console.log(boundShowDetails("Los Angeles"));
```

5. How Does JavaScript's Garbage Collection Work?

Answer:
JavaScript uses a mark-and-sweep algorithm for garbage collection. The engine marks all objects that are reachable from the root (global scope, current execution context, etc.) and then sweeps away objects that aren't marked, freeing memory. Understanding object references and scope is crucial to avoiding memory leaks.

6. What Are the Nuances of the `this` Keyword and Arrow Functions?

Answer:
The `this` keyword refers to the execution context. Arrow functions don't have their own `this`; they inherit it lexically from the surrounding scope. This behavior is useful in callback functions where maintaining the original context is necessary.

Example:

```
const obj = {

  name: "Arrow Function Demo",

  regularFunction: function() {

    console.log(this.name);

  },

  arrowFunction: () => {

    // 'this' here is inherited from the global scope

    console.log(this.name);

  }

};

obj.regularFunction(); // "Arrow Function Demo"

obj.arrowFunction();   // undefined (or global name if defined)
```

7. What Is the Temporal Dead Zone (TDZ) in JavaScript?

Answer:
TDZ refers to the period between entering a block and the variable's declaration when using `let` or `const`. Accessing the variable in this zone results in a `ReferenceError`. This behavior enforces proper declaration before usage.

Example:

```
{
  // console.log(temp); // ReferenceError: Cannot access 'temp' before
initialization
  let temp = "TDZ";
  console.log(temp); // "TDZ"
}
```

8. How Do Synchronous and Asynchronous Execution Differ in JavaScript?

Answer:
Synchronous code executes sequentially, blocking subsequent operations until completion. Asynchronous code, on the other hand, allows other operations to run while waiting for tasks (like API calls or timers) to complete. This is crucial for non-blocking I/O and smooth user experiences in web applications.

9. What Are Symbols and How Do They Enhance Object Properties?

Answer:
Symbols are unique and immutable primitive values used as identifiers for object properties. They help prevent naming collisions, especially in large codebases or when integrating multiple libraries, by providing a unique key for each property.

Example:

```
const sym = Symbol('unique');
const obj = { [sym]: 'value' };
console.log(obj[sym]); // "value"
```

10. What Is the Reflect API and How Do Proxies Work in JavaScript?

Answer:
The Reflect API provides methods for interceptable JavaScript operations (like getting, setting,

or deleting properties) that mirror the behavior of internal methods. Proxies allow you to intercept and redefine fundamental operations for an object. They're used for logging, validation, or creating custom behaviors.

Example:

```
const target = { name: "John" };

const handler = {

  get(target, property) {

    console.log(`Accessing ${property}`);

    return target[property];

  }

};

const proxy = new Proxy(target, handler);

console.log(proxy.name); // Logs: "Accessing name", then "John"
```

11. How Would You Implement a Custom Event Emitter in Vanilla JavaScript?

Answer:
A custom event emitter allows objects to subscribe to and emit events. You can implement this using closures to maintain a registry of events and their listeners.

Example:

```
function EventEmitter() {

  const events = {};

  this.on = function(event, listener) {

    if (!events[event]) {

      events[event] = [];

    }
    events[event].push(listener);

  };

  this.emit = function(event, ...args) {

    if (events[event]) {

      events[event].forEach(listener => listener(...args));

    }

  };

}

// Usage:

const emitter = new EventEmitter();

emitter.on("data", data => console.log("Received:", data));

emitter.emit("data", "Hello World!");
```

12. What Is the Module Pattern and the Revealing Module Pattern?

Answer:
The Module Pattern encapsulates related functions, variables, and state into a single unit with a public API, avoiding global scope pollution. The Revealing Module Pattern improves this by returning an object that exposes only the methods you choose, keeping internal details private.

Example:

```
const myModule = (function() {

  let privateVar = "I am private";

  function privateMethod() {

    console.log(privateVar);

  }
  function publicMethod() {

    privateMethod();

  }

  return {

    publicMethod

  };

})();

myModule.publicMethod(); // "I am private"
```

13. How Can You Implement Dependency Injection in JavaScript?

Answer:
Dependency injection involves passing required dependencies into a function or module instead of hardcoding them. This approach promotes loose coupling and easier testing.

Example:

```
function createUserService(httpClient) {

  return {

    getUser(id) {

      return httpClient.get(`/users/${id}`);

    }

  };

}

// Inject a mock or real HTTP client:

const userService = createUserService(fetch);

userService.getUser(1);
```

14. What Are WeakMap and WeakSet and When Should You Use Them?

Answer:
WeakMap and WeakSet allow you to store weakly referenced objects, meaning they do not prevent garbage collection if there are no other references. Use them for caching or tracking metadata where you do not want to interfere with garbage collection.

Example:

```
let wm = new WeakMap();

let obj = {};

wm.set(obj, "metadata");

obj = null; // Now eligible for garbage collection
```

15. Explain Function Currying and Partial Application in Detail

Answer:
Currying transforms a function with multiple arguments into a sequence of functions, each taking a single argument. Partial application involves fixing a few arguments of a function, producing another function of smaller arity. Both techniques enable reusability and modularity.

Example (Currying):

```
function curriedSum(a) {

  return function(b) {

    return a + b;

  }

}

const addFive = curriedSum(5);

console.log(addFive(10)); // 15
```

16. What Is Tail Call Optimization and Does JavaScript Support It?

Answer:
Tail call optimization (TCO) is a technique where the last function call in a recursive function is optimized to reuse the current stack frame. While ES6 introduced proper tail calls, support in JavaScript engines remains inconsistent, so reliance on TCO is not widespread.

Example:

```
function factorial(n, acc = 1) {

  if (n <= 1) return acc;

  return factorial(n - 1, n * acc); // Tail call

}

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

17. What Are Async Iterators and Generators?

Answer:
Generators are functions that can pause execution (`yield`) and resume later. Async iterators extend this concept to asynchronous operations using `for await...of` loops, allowing you to handle data streams or asynchronous sequences elegantly.

Example (Generator):

```
function* countGenerator() {

  yield 1;

  yield 2;

  yield 3;

}

for (let value of countGenerator()) {

  console.log(value);
}
```

18. How Would You Implement a Custom Promise from Scratch?

Answer:
Building a Promise involves managing states (pending, fulfilled, rejected), storing callbacks, and ensuring proper asynchronous execution. While simplified versions exist for educational purposes, creating a production-level promise requires rigorous handling of edge cases.

Example (Simplified):

```
function MyPromise(executor) {

  let onResolve, onReject;

  let fulfilled = false, rejected = false, value;

  function resolve(val) {

    fulfilled = true;

    value = val;

    if (typeof onResolve === 'function') {

      onResolve(value);

    }

  }

  function reject(reason) {

    rejected = true;

    value = reason;

    if (typeof onReject === 'function') {

      onReject(value);

    }
  }

  this.then = function(callback) {

    onResolve = callback;

    if (fulfilled) onResolve(value);

    return this;

  };

  this.catch = function(callback) {

    onReject = callback;

    if (rejected) onReject(value);

    return this;

  };

  executor(resolve, reject);

}

// Usage:

const promise = new MyPromise((resolve, reject) => {

  setTimeout(() => resolve("Done"), 1000);

});

promise.then(result => console.log(result));
```

19. How Do Microtasks and Macrotasks Differ in Node.js (e.g., `process.nextTick`)?

Answer:
In Node.js, `process.nextTick` queues a microtask that executes before the event loop continues, even before promise callbacks. Macrotasks (like `setTimeout`) are queued for the

next cycle. Understanding these differences is key for performance tuning and avoiding starvation of I/O tasks.

20. What Are Some Methods to Deep Clone Objects Beyond `JSON.parse/stringify`?

Answer:
Besides the JSON approach, you can use recursive functions, the structured clone algorithm (if available via `structuredClone()`), or libraries like Lodash's `cloneDeep` that handle functions, dates, maps, and more complex types.

21. How Would You Advancedly Implement Debounce and Throttle Functions?

Answer:
Building robust debounce and throttle functions involves careful handling of closures, timers, and ensuring proper context (`this`) preservation. Advanced implementations allow configuration for immediate execution, trailing invocation, and cancellation of pending executions.

Example (Advanced Debounce):

```
function debounce(fn, delay, immediate = false) {

  let timeout;

  return function(...args) {

    const context = this;

    const later = () => {

      timeout = null;

      if (!immediate) fn.apply(context, args);

    };

    const callNow = immediate && !timeout;

    clearTimeout(timeout);
    timeout = setTimeout(later, delay);

    if (callNow) fn.apply(context, args);

  };

}
```

22. How Do You Implement the Observer Pattern in JavaScript?

Answer:
The Observer pattern involves subjects maintaining a list of observers, notifying them when changes occur. This pattern is useful for creating reactive systems.

Example:

```
function Subject() {

  this.observers = [];

}

Subject.prototype.subscribe = function(observer) {

  this.observers.push(observer);

};

Subject.prototype.notify = function(message) {

  this.observers.forEach(observer => observer.update(message));

};

function Observer(name) {

  this.name = name;

}

Observer.prototype.update = function(message) {

  console.log(`${this.name} received: ${message}`);
};

// Usage:

const subject = new Subject();

const obs1 = new Observer("Observer1");

const obs2 = new Observer("Observer2");

subject.subscribe(obs1);

subject.subscribe(obs2);

subject.notify("Hello Observers!");
```

23. How Can You Create a Simple Virtual DOM Implementation?

Answer:
A simple virtual DOM involves representing DOM elements as plain JavaScript objects, diffing these objects to identify changes, and then updating the real DOM accordingly. While real-world virtual DOMs (like React's) are complex, a basic version demonstrates key concepts.

Example (Conceptual):

```
function createElement(tag, props, ...children) {

  return { tag, props: props || {}, children };

}

function render(vdom) {

  const el = document.createElement(vdom.tag);

  Object.entries(vdom.props).forEach(([key, value]) =>
el.setAttribute(key, value));

  vdom.children.forEach(child => {

    const childEl = typeof child === "object" ? render(child) :
document.createTextNode(child);
```

```
    el.appendChild(childEl);

  });

  return el;

}
```

24. What Are Advanced Uses of Destructuring and the Rest/Spread Operators?

Answer:
Advanced destructuring allows you to extract values from nested objects or arrays, rename variables, and set defaults. The rest and spread operators can merge or split arrays and objects in deep copies, making code more concise and expressive.

Example (Nested Destructuring):

```
const user = {

  name: "Eve",

  address: {

    city: "Wonderland",

    zip: 12345

  }

};

const { name, address: { city, zip } } = user;

console.log(name, city, zip);
```

25. How Do You Handle Advanced Error Handling with Custom Error Classes?

Answer:
Creating custom error classes by extending the built-in `Error` class provides more descriptive

error messages and consistent error handling. Custom errors are useful for differentiating between various failure modes in an application.

Example:

```
class ValidationError extends Error {

  constructor(message) {

    super(message);

    this.name = "ValidationError";

  }

}

function validateUser(user) {

  if (!user.name) {

    throw new ValidationError("User must have a name");

  }

}

try {

  validateUser({});

} catch (error) {

  if (error instanceof ValidationError) {

    console.error("Validation error:", error.message);

  } else {

    console.error("Unknown error:", error);

  }

}
```

These 25 advanced questions cover a wide range of topics—from asynchronous patterns and memory management to deep object manipulation and design patterns in vanilla JavaScript. Understanding these concepts will not only improve your code quality but also equip you with the tools to build more sophisticated applications.

JavaScript is one of the most popular programming languages today, powering everything from simple web page interactions to complex web applications. Whether you're just starting out or have been coding for years, understanding key JavaScript concepts is essential. In this post, we'll walk through 25 common JavaScript coding questions, complete with detailed explanations and examples to deepen your understanding.

1. What is the Difference Between `var`, `let`, and `const`?

Answer:

- **`var`** is function-scoped and can be re-declared. It is hoisted, meaning its declaration is moved to the top of its scope, though not its initialization.
- **`let`** is block-scoped (inside `{ }`), cannot be re-declared in the same scope, and is not hoisted in the same way as `var` (it exists in a "temporal dead zone" until its declaration).
- **`const`** is also block-scoped but, unlike `let`, its value cannot be reassigned once set. However, the contents of objects or arrays declared with `const` can be modified.

Example:

```
function example() {
  var a = 10;
  let b = 20;
  const c = 30;
  if (true) {
    var a = 40; // redeclares and overwrites the outer 'a'
    let b = 50; // new variable within block scope
    // c = 60; // Error: Assignment to constant variable.
  }
  console.log(a); // 40
  console.log(b); // 20
}
example();
```

2. How Does JavaScript Hoisting Work?

Answer:
Hoisting is JavaScript's default behavior of moving declarations to the top of their containing scope during compilation. However, only declarations are hoisted, not initializations.

- Variables declared with `var` are hoisted but initialized to `undefined` until their assignment.
- Functions declared with the function declaration syntax are hoisted entirely, meaning they can be called before they appear in the code.

Example:

```
console.log(hoistedVar); // undefined
var hoistedVar = 'I am hoisted';
hoistedFunc(); // "I am a hoisted function"
function hoistedFunc() {
  console.log("I am a hoisted function");
}
```

3. What are Closures in JavaScript?

Answer:
A closure is a function that remembers its lexical scope even when the function is executed outside that scope. It allows a function to access variables from an enclosing scope or function even after the outer function has finished executing.

Example:

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

Explanation: The inner function retains access to the `count` variable defined in `makeCounter` even after `makeCounter` has completed.

4. What is an IIFE (Immediately Invoked Function Expression)?

Answer:
An IIFE is a function that is executed right after it is defined. It's used to create a new scope and avoid polluting the global namespace.

Example:

```
(function() {
  let message = "Hello, IIFE!";
  console.log(message);
})();
```

Explanation: The function is defined and then immediately invoked, keeping the variable `message` scoped within the function.

5. What is the Event Loop in JavaScript?

Answer:
The event loop is a core mechanism that allows JavaScript to perform non-blocking operations, despite being single-threaded. It continuously checks the call stack and the task queue, executing tasks asynchronously when the call stack is empty.

Explanation:
When asynchronous operations (like API calls or timers) complete, their callbacks are pushed to the task queue. The event loop then moves them to the call stack when it's empty.

6. Explain Promises in JavaScript and How to Use Them

Answer:
Promises represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They allow you to write asynchronous code in a more manageable way.

Example:

```
const promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Operation succeeded!");
  } else {
    reject("Operation failed!");
  }
});
promise
```

```
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Explanation: A promise takes an executor function that runs immediately and either resolves with a value or rejects with an error. `.then()` handles success, while `.catch()` handles errors.

7. What is `async/await` in JavaScript?

Answer:
`async/await` is syntax that simplifies working with promises. An `async` function always returns a promise, and the `await` keyword pauses the function execution until the promise resolves.

Example:

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

Explanation: `async` functions allow you to write asynchronous code that looks synchronous, improving readability and error handling.

8. How Does the `this` Keyword Work in JavaScript?

Answer:
`this` refers to the object that is executing the current function. Its value depends on the context:

- In a method, `this` refers to the object the method belongs to.
- In a function, in non-strict mode, `this` defaults to the global object; in strict mode, it remains `undefined`.
- In event handlers, `this` refers to the element that received the event.

- Arrow functions do not have their own `this`; they inherit it from the parent scope.

Example:

```
const obj = {
  name: "JavaScript",
  getName: function() {
    return this.name;
  }
};
console.log(obj.getName()); // "JavaScript"
```

9. What is the Difference Between Function Declaration and Function Expression?

Answer:

- **Function Declaration:**
  Declares a function with a given name and is hoisted, so you can call it before it is defined in the code.

- **Function Expression:**
  Defines a function as part of a larger expression and is not hoisted. It can be anonymous or named.

Example:

```
// Function Declaration
function greet() {
  return "Hello!";
}
console.log(greet()); // "Hello!"
// Function Expression
const sayHi = function() {
  return "Hi!";
};
console.log(sayHi()); // "Hi!"
```

10. How Do Arrow Functions Differ from Traditional Functions?

Answer:
Arrow functions provide a shorter syntax and have lexical scoping for `this`, meaning they don't bind their own `this` value. They are best used for non-method functions.

Example:

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

Explanation: The arrow function simplifies the syntax and inherits `this` from the surrounding code, making them useful in many scenarios.

11. What is Prototypal Inheritance in JavaScript?

Answer:
JavaScript objects inherit properties and methods from a prototype. This means that objects can share methods and properties through their prototype chain, enabling code reuse.

Example:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  return `Hello, my name is ${this.name}`;
};
const alice = new Person("Alice");
console.log(alice.greet()); // "Hello, my name is Alice"
```

Explanation: The `greet` method is defined on the prototype, so all instances of `Person` share the same function.

12. What is the Purpose of the `bind` Method?

Answer:
The `bind` method creates a new function with a specified `this` context and, optionally, initial

arguments. It is useful for ensuring a function retains its intended context, regardless of how it is called.

Example:

```
const person = {
  name: "Bob",
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
};
const greet = person.greet;
const boundGreet = greet.bind(person);
boundGreet(); // "Hello, my name is Bob"
```

13. Explain the Concept of Callback Functions

Answer:
A callback is a function passed as an argument to another function, which is then invoked inside the outer function to complete some action or after an asynchronous operation.

Example:

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data fetched!");
  }, 1000);
}
fetchData(message => {
  console.log(message); // "Data fetched!" after 1 second
});
```

Explanation: Callbacks help manage asynchronous operations and keep the code non-blocking.

14. How Do You Perform Deep Cloning of Objects in JavaScript?

Answer:
Deep cloning creates a completely new object that is a copy of the original, including nested

objects. One common method is using `JSON.parse(JSON.stringify(object))`, though it has limitations (e.g., loss of functions, handling of special types).

Example:

```
const original = { a: 1, b: { c: 2 } };
const clone = JSON.parse(JSON.stringify(original));
clone.b.c = 42;
console.log(original.b.c); // 2 (original remains unchanged)
```

Explanation: Deep cloning ensures that changes to the cloned object do not affect the original.

15. What Are Modules in JavaScript? Explain Using `import/export`

Answer:
Modules allow you to break your code into reusable pieces with their own scope. The `export` keyword is used to expose variables or functions, while `import` is used to bring them into another file.

Example (module file - `math.js`):

```
export function add(a, b) {
  return a + b;
}
```

Example (importing file):

```
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

Explanation: Modules help in organizing code, making it easier to maintain and reuse.

16. What is Event Delegation in JavaScript?

Answer:
Event delegation is a technique where a single event listener is added to a parent element to manage events for its child elements. It leverages event bubbling to capture events at a higher level.

Example:

```
document.querySelector("#parent").addEventListener("click", event => {
  if (event.target.matches(".child")) {
    console.log("Child element clicked!");
  }
});
```

Explanation: This method improves performance by reducing the number of event listeners attached to elements.

17. What is the Difference Between `null` and `undefined`?

Answer:

- **`undefined`:** Indicates that a variable has been declared but has not been assigned a value.
- **`null`:** Represents the intentional absence of any object value.

Example:

```
let a;
console.log(a); // undefined
let b = null;
console.log(b); // null
```

Explanation: While both denote an absence of value, `undefined` is the default state, and `null` is explicitly assigned.

18. How Does Type Coercion Work in JavaScript?

Answer:
Type coercion is JavaScript's automatic or implicit conversion of values from one type to another (e.g., string to number). It often happens when using operators like + (which concatenates if one operand is a string) or when comparing values with ==.

Example:

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

Explanation: JavaScript dynamically converts types to make operations work, which can sometimes lead to unexpected results.

19. What Are the Differences Between == and === Operators?

Answer:

- **== (Equality Operator):** Checks for value equality after performing type coercion if necessary.
- **=== (Strict Equality Operator):** Checks for both value and type equality, without coercion.

Example:

```
console.log(5 == '5');  // true (coerced comparison)
console.log(5 === '5'); // false (different types)
```

Explanation: Using === is generally recommended to avoid unexpected type coercion bugs.

20. Explain the Concept of Currying in JavaScript

Answer:
Currying is a functional programming technique where a function is transformed into a sequence of functions, each taking a single argument. It helps create more specialized functions from a general function.

Example:

```
function multiply(a) {
  return function(b) {
    return a * b;
  }
}
const double = multiply(2);
console.log(double(5)); // 10
```

Explanation: Currying allows partial application of functions, making it easier to reuse and compose functions.

21. What is Memoization and How Can You Implement It?

Answer:
Memoization is an optimization technique that caches the results of expensive function calls and returns the cached result when the same inputs occur again. It improves performance for repeated computations.

Example:

```
function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache[key]) {
      return cache[key];
    }
    const result = fn(...args);
    cache[key] = result;
    return result;
  }
}
const factorial = memoize(function(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
});
console.log(factorial(5)); // 120
```

Explanation: By caching the results, subsequent calls with the same arguments bypass the computation, saving time.

22. What Are JavaScript Template Literals and How to Use Them?

Answer:
Template literals allow you to embed expressions in strings, spanning multiple lines easily. They use backticks (`) instead of quotes.

Example:

```
const name = "Alice";
const greeting = `Hello, ${name}!
Welcome to our website.`;
console.log(greeting);
```

Explanation: Template literals enhance readability and simplify string interpolation.

23. How Does Destructuring Work in JavaScript?

Answer:
Destructuring lets you extract properties from arrays or objects into individual variables in a concise way.

Example (Array Destructuring):

```
const numbers = [1, 2, 3];
const [one, two, three] = numbers;
console.log(one, two, three); // 1 2 3
```

Example (Object Destructuring):

```
const person = { name: "Bob", age: 25 };
const { name, age } = person;
console.log(name, age); // "Bob" 25
```

Explanation: Destructuring makes code cleaner and reduces redundancy when accessing multiple properties.

24. What Are the Spread and Rest Operators in JavaScript?

Answer:

- **Spread Operator (`...`):** Expands elements of an iterable (e.g., array) into individual elements.
- **Rest Operator (`...`):** Collects multiple elements into a single array parameter in a function definition.

Example (Spread):

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];
console.log(arr2); // [1, 2, 3, 4, 5]
```

Example (Rest):

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

Explanation: These operators offer elegant ways to work with collections and function arguments.

25. How Do You Debounce and Throttle Functions in JavaScript?

Answer:
Both debounce and throttle are techniques to control how often a function executes:

- **Debouncing:** Delays the function execution until after a specified wait time has elapsed since the last invocation. Useful for events that fire frequently, like resizing or keypress events.
- **Throttling:** Ensures that a function is called at most once every specified time interval, regardless of how many times an event fires.

Example (Debounce):

```
function debounce(fn, delay) {
  let timeoutID;
  return function(...args) {
    clearTimeout(timeoutID);
    timeoutID = setTimeout(() => fn.apply(this, args), delay);
  }
}
window.addEventListener('resize', debounce(() => {
  console.log('Resize event handled!');
}, 300));
```

Example (Throttle):

```
function throttle(fn, interval) {
  let lastTime = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastTime >= interval) {
      lastTime = now;
      fn.apply(this, args);
    }
  }
}
window.addEventListener('scroll', throttle(() => {
  console.log('Scroll event handled!');
}, 200));
```

Explanation:
Debouncing helps prevent functions from being called too rapidly, while throttling guarantees a regular rate of execution regardless of how many events occur.

JavaScript is full of powerful concepts that can greatly improve the efficiency and readability of your code. Understanding these 25 questions and their detailed answers not only strengthens your grasp on JavaScript but also empowers you to write cleaner, more optimized code.