ChatGPT

# Core JavaScript Concepts

1. **What are the different data types in JavaScript?**
   JavaScript data types are categorized as *primitive* and *non-primitive* [1]. Primitive types include `string`, `number`, `boolean`, `undefined`, `null`, `symbol`, and `bigint`. Non-primitive types are objects (including arrays and functions). For example, `"hello"` is a primitive string, while `{}` is an object.

2. **What is the difference between `==` and `===` in JavaScript?**
   The `==` operator checks for loose equality by converting ("coercing") types if needed, whereas `===` checks for strict equality of both value and type [2]. For example, `5 == "5"` is `true` (number and string are coerced), but `5 === "5"` is `false` (different types). In general, use `===` to avoid unexpected type conversions.

3. **What are `var`, `let`, and `const`, and how do they differ?**
   These keywords declare variables with different scope and mutability [3]. `var` is function-scoped (or global if outside a function), and allows re-declaration. `let` and `const` are block-scoped (limited to `{…}`) [3]. `let` variables can be reassigned but not re-declared in the same scope. `const` variables must be initialized on declaration and cannot be reassigned (though object contents can mutate) [3]. In strict mode, using `var` inside blocks still hoists to the function scope, which can lead to bugs, so `let` / `const` are preferred for block-level scoping.

4. **Explain hoisting in JavaScript.**
   Hoisting is JavaScript's behavior of moving **declarations** to the top of their scope at compile time. In practice, this means you can reference a `var`-declared variable before its declaration, but its value will be `undefined` until the actual assignment [4]. Function declarations are also hoisted entirely, so you can call a function before its declaration. However, `let` and `const` are not initialized until their declaration is evaluated; referencing them before declaration causes a `ReferenceError` (the *Temporal Dead Zone*) [5]. For example:

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

   behaves like `var a; console.log(a); a = 5;` [6].

5. **What is the difference between `null` and `undefined` in JavaScript?**
   `null` is an assignment value that represents an *intentional* absence of value; it must be explicitly set. `undefined` means a variable has been declared but not assigned a value [7]. For example, if you write `let x;`, then `x === undefined`. If you write `let y = null;`, then `y` is deliberately set to `null`. In code:

   ```
   let a = null;      // explicit "no value"
   let b;             // declared but unassigned
   ```

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

6. **What are closures in JavaScript?**
   A **closure** is a function that retains access to its lexical scope even after the outer function has finished execution [8]. In other words, an inner function "remembers" the variables from the context in which it was created. For example:

```
function outer() {
  let x = 10;
  return function inner() {
    console.log(x);
  };
}
const closureFunc = outer();
closureFunc(); // prints 10
```

   Here, `inner` is a closure: it still sees `x` from `outer`'s scope even after `outer` has returned [8]. Closures are useful for data privacy and callback functions.

7. **How does the `this` keyword work in JavaScript?**
   In JavaScript, `this` refers to the object that is the current context of execution [9]. Its value depends on how a function is called:

8. **Global context:** In non-strict mode, `this` is the global object (`window` in browsers).
9. **Object method:** `this` is the object the method belongs to.
10. **Constructor:** When using `new`, `this` is the newly created instance.
11. **Arrow function:** Arrow functions *do not* have their own `this`; they inherit `this` from the surrounding lexical context [10].
    For example:

```
const obj = { value: 42, getValue: function() { return this.value; } };
console.log(obj.getValue()); // 42 (this refers to obj)
const f = obj.getValue;
console.log(f()); // undefined in strict mode (this is global or
undefined)
```

    Using `call` / `apply` / `bind`, you can explicitly set `this` for a function (see next question).

12. **Explain `call()`, `apply()`, and `bind()` methods.**
    These methods allow you to control the value of `this` when invoking functions:

13. `call(thisArg, arg1, arg2, ...)` invokes the function immediately, setting `this` to `thisArg` and passing subsequent arguments one by one [11].

14. `apply(thisArg, [argsArray])` invokes the function immediately, setting `this` to `thisArg` and passing arguments as an array [12]. The primary difference between `call` and `apply` is how arguments are provided (list vs array).

15. `bind(thisArg, arg1, arg2, ...)` does not call the function immediately. Instead, it returns a *new function* with `this` bound to `thisArg` and optionally preset initial arguments [13]. Calling that new function will use the bound `this`.
    For example:

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

16. **What is prototypal inheritance in JavaScript?**
    Every object in JavaScript has an internal link (`[[Prototype]]`) to another object, its *prototype*. When you access a property on an object, JavaScript looks for that property on the object itself; if not found, it follows the prototype link chain until it finds it or reaches an object with `null` prototype [14]. This chain of prototypes implements inheritance: child objects "inherit" properties from their prototype. Classes in JS are mostly syntactic sugar on top of this mechanism [14]. For example:

    ```
    const parent = { hello() { console.log("Hello"); } };
    const child = Object.create(parent);
    child.hello(); // finds hello() on parent via prototype chain
    ```

    Here, `child` does not have its own `hello()`, but its prototype `parent` does, so `child.hello()` works. The chain ends when it reaches an object whose prototype is `null`.

17. **What are immediately-invoked function expressions (IIFEs) and why are they used?**
    An IIFE is a function that is defined and invoked at the same time. For example:

    ```
    (function() {
      console.log("IIFE runs immediately");
    })();
    ```

    The parentheses around the function turn it into an expression, and the trailing `()` invokes it. IIFEs are used to create a new scope without polluting the global namespace. Variables declared inside an IIFE are not visible outside it, avoiding conflicts. Before `let/const` and modules, IIFEs were a common pattern to emulate block scope.

18. **What is the event loop in JavaScript?**
    JavaScript is single-threaded, so it uses an **event loop** to handle asynchronous callbacks without blocking. The event loop continuously checks the call stack and the task queues (callbacks from events, timers, promises, etc.). When the call stack is empty, the event loop takes the next task

from the callback (or microtask) queue and pushes it onto the stack for execution [15]. This mechanism allows JavaScript to perform non-blocking operations; for example, a `setTimeout` callback waits in a queue until its delay expires and the stack is clear. As GfG notes, the event loop "moves tasks from the callback queue to the call stack when the stack is empty" [15].

# ES6 and Modern JavaScript

1. **What are arrow functions and how do they differ from regular functions?**
   Arrow functions are a concise syntax introduced in ES6. For example, `(x, y) => x + y` is an arrow function equivalent to `function(x, y) { return x + y; }`. Key differences are:

   - **Lexical** `this`: Arrow functions do *not* have their own `this`. Instead, `this` is inherited from the surrounding scope [10]. This makes arrow functions useful in callbacks where you want `this` to stay consistent.
   - **No** `arguments` **object**: Arrow functions do not have their own `arguments`; to access all args you'd use rest parameters.
   - **Not constructible**: You cannot use `new` with an arrow function.
   - **Expression body**: If the arrow function body is an expression (no braces), it implicitly returns that value. With braces, you must use `return`.
     Example:

     ```
     // Regular function
     function add(a, b) { return a + b; }
     // Arrow function
     const addArrow = (a, b) => a + b;
     ```

     In object methods, arrow functions should generally be avoided because they don't bind their own `this` [10].

2. **What are template literals and how are they used?**
   Template literals (backtick-quoted strings) allow embedded expressions and multi-line text [16]. They use backticks (`` ` ``) instead of quotes. Inside a template literal, `${...}` is used to interpolate expressions. Example:

   ```
   const name = "Alice", item = "book";
   console.log(`Hello ${name}, you bought a ${item}.`);
   // Output: Hello Alice, you bought a book.
   ```

   Template literals also naturally span multiple lines:

   ```
   console.log(`Line1
   Line2`);
   // Prints:
   // Line1
   // Line2
   ```

According to MDN, template literals “allow for multi-line strings, string interpolation with embedded expressions…” [16].

3. **What is destructuring in JavaScript?**
   Destructuring allows unpacking values from arrays or objects into distinct variables. For example, array destructuring:

```
const arr = [1, 2, 3];
const [a, b, c] = arr; // a=1, b=2, c=3
```

   Object destructuring:

```
const obj = { x: 10, y: 20 };
const { x, y } = obj; // x=10, y=20
```

   You can also use default values and the rest operator. Destructuring makes it concise to extract multiple properties or elements in one statement.

4. **What are default parameters in functions?**
   Default parameters allow functions to have parameters with default values if none or `undefined` is passed [17]. For example:

```
function multiply(a, b = 1) {
  return a * b;
}
console.log(multiply(5, 2)); // 10
console.log(multiply(5));    // 5 (b defaults to 1)
```

   In this example, `b` defaults to 1 if not provided. MDN notes that default parameters “allow named parameters to be initialized with default values if no value or `undefined` is passed” [17].

5. **What is the spread operator (`...`) and the rest operator?**
   The `...` syntax serves two roles: as a **spread operator** it expands an iterable into individual elements, and as **rest parameters** it collects multiple elements. For example:

   - **Spread**:

```
const nums = [1, 2, 3];
const copy = [...nums];    // spreads into a new array [1,2,3]
const joined = [...nums, 4, 5]; // [1,2,3,4,5]
```

   - **Rest**:
     In function definitions or destructuring, `...` can capture remaining items into an array or object. For example:

```
function sum(...args) {
  return args.reduce((a, b) => a + b, 0);
}
console.log(sum(1,2,3)); // 6, args = [1,2,3]
```

Or with array destructuring:

```
const [first, ...rest] = [10, 20, 30, 40];
console.log(first); // 10
console.log(rest);  // [20, 30, 40]
```

The distinction: spread "unpacks" elements, whereas rest "packs" the remaining elements.

6. **What are modules in JavaScript (ES6 modules)?**
   ES6 modules allow you to export and import code between files. You use `export` to expose functions/variables from one file, and `import` to use them in another. For example:

```
// math.js
export function add(x, y) { return x + y; }
// main.js
import { add } from './math.js';
console.log(add(2,3)); // 5
```

   Modules run in strict mode by default and have their own scope (top-level `let/const` don't pollute the global scope). They differ from older `require()` (CommonJS) in that ES6 modules are statically analyzed (imports are resolved at compile time) and use browser-supportable syntax.

7. **What are classes in JavaScript, and how do they relate to prototypes?**
   Classes in JavaScript (introduced in ES6) are mostly syntactic sugar over the existing prototypal inheritance [18]. The `class` keyword provides a cleaner syntax to define constructor functions and methods. For example:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}
const p = new Person("Bob");
p.greet(); // Hello, Bob
```

   Internally, JavaScript classes still work via prototypes. As MDN explains, "Classes are a template for creating objects... Classes in JS are built on prototypes" [18]. The `constructor` function and methods are actually placed on the prototype chain.

# Asynchronous JavaScript

1. **What is a callback function?**
   A callback is a function passed as an argument to another function, to be "called back" later after some operation completes. Callbacks are fundamental for handling asynchronous behavior in JavaScript (like I/O or timers) [19]. For example:

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}
function goodbye() {
  console.log("Goodbye!");
}
greet("Anjali", goodbye);
// Output:
// Hello Anjali
// Goodbye!
```

   Here, `goodbye` is a callback passed to `greet`, and it executes after logging the greeting. Callbacks can lead to nested code (callback "hell"), which is why Promises and async/await were introduced.

2. **What are promises in JavaScript and how do they work?**
   A **Promise** is an object representing the eventual result (or failure) of an asynchronous operation [20]. It has three states: `pending` (initial), `fulfilled` (completed successfully), or `rejected` (failed) [20]. You create a promise via `new Promise((resolve, reject) => { … })`, and call `resolve(value)` or `reject(error)` to settle it. Consumers use `.then()` to handle success and `.catch()` for errors. For example:

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;
  if (success) resolve("Done");
  else reject("Error");
});
myPromise.then(result => console.log(result))  // "Done"
         .catch(error => console.error(error));
```

   Promises allow chaining and avoid deeply nested callbacks. They always return immediately with a promise object, and the callback passed to `.then()` runs once the promise resolves.

3. **How does `async/await` work in JavaScript?**
   The `async` / `await` syntax (ES2017) makes working with promises more straightforward. An `async` function always returns a promise [21]. Inside it, you can use `await` to pause execution until a promise settles, making asynchronous code appear synchronous. For example:

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

In this code, `await fetch()` pauses until the network request completes. The benefit is "avoiding the need to explicitly configure promise chains" and writing cleaner code [22]. Under the hood, each `await` builds on `.then()` callbacks. Note that `await` can only be used in `async` functions, and errors can be caught with `try/catch`.

4. **What is `Promise.all` and `Promise.race`?**
   `Promise.all([...promises])` returns a new promise that fulfills when *all* input promises fulfill (or rejects if any reject). It yields an array of results.

```
Promise.all([p1, p2]).then(results => {
  // results is [resultOfP1, resultOfP2]
});
```

`Promise.race([...promises])` returns a promise that settles as soon as *any* input promise settles (resolves or rejects) – it "races" them. The resulting promise resolves/rejects with the value/reason of the first-settled promise. These are useful for running tasks in parallel and aggregating results or timeouts.

5. **What is the difference between synchronous and asynchronous code in JS?**
   Synchronous code runs in sequence, blocking further execution until each step finishes. Asynchronous code (like I/O, timers, or event callbacks) does not block the main thread. Instead, such tasks are handed off (e.g., to the browser APIs or Node's libuv), and a callback or promise is used to handle the result later. The event loop then processes these callbacks when the main execution stack is clear [15]. In practice, synchronous code might be computations or blocking loops, whereas AJAX calls, `setTimeout`, or file reads are asynchronous, allowing the main thread to continue.

6. **How are errors handled in promises vs async/await?**
   With promises, you handle errors via `.catch()`. For example: `fetch(...).then(resp => resp.json()).catch(err => console.error(err));`. With `async/await`, you typically use `try/catch` around the await calls:

```
async function f() {
  try {
    const resp = await fetch('/api');
    const data = await resp.json();
  } catch (err) {
```

```
    console.error("Error:", err);
  }
}
```

This makes error handling look more synchronous. Both patterns propagate rejections as exceptions; an unhandled rejection in a promise will throw an error in async functions.

7. **What is the call stack and how does it relate to the event loop?**
   The call stack is the mechanism JavaScript uses to keep track of function calls. Each time a function is invoked, it's pushed onto the stack; when it returns, it's popped off. If the stack is empty, the engine can process queued tasks (from timers, I/O, or promise microtasks) via the event loop [15]. This means asynchronous callbacks only run when the stack is empty, ensuring one task at a time. If the stack never empties (e.g., an infinite loop), then callbacks (even resolved promises) never get a chance to run, because the event loop never proceeds.

# DOM, BOM, and Browser Behavior

1. **What is the DOM (Document Object Model)?**
   The DOM is a programming interface for HTML/XML documents. It represents the page structure as a tree of objects (nodes) – elements, text, attributes, etc. Through the DOM API (like `document.getElementById()` or `querySelector()`), JavaScript can read and manipulate the page content and structure dynamically. For example, `document.body.innerHTML = "Hello"` changes the page's body content. Think of the DOM as a live object representation of the web page that JavaScript can interact with.

2. **How do you select and manipulate DOM elements?**
   You can select elements using methods like `document.getElementById("id")`, `document.querySelector(".class")`, `document.getElementsByTagName("div")`, etc. Once selected, you can read/modify their properties. For example:

   ```
   const elem = document.getElementById("title");
   elem.textContent = "New Title"; // change text
   elem.style.color = "red";       // change CSS
   ```

   Creating elements is done with `document.createElement()`, and appending with methods like `appendChild` or `append`. Removing an element can be done with `elem.remove()`. Always ensure scripts run after DOM loads (e.g., put `<script>` at end of body or use `DOMContentLoaded`).

3. **What are event listeners and how does event propagation work?**
   You attach event handlers using `addEventListener`, e.g. `button.addEventListener("click", handler)`. When an event occurs, it triggers event listeners on that element and, by default, bubbles up the DOM tree (propagation) to ancestors. You can also capture from the top down. For example:

```
document.querySelector("div").addEventListener("click", () =>
console.log("div"));
document.body.addEventListener("click", () => console.log("body"));
```

Clicking the div will log “div” then “body” (bubbling). You can stop propagation with `event.stopPropagation()`, and prevent default browser actions (like form submission) with `event.preventDefault()`.

4. **What is event delegation?**
   Event delegation is a pattern where you set a single event listener on a parent element to handle events on its children, instead of adding separate listeners to each child. This works because of event bubbling. Inside the handler, you can check `event.target` to identify which child triggered the event. For example, if you have many list items:

```
ul.addEventListener("click", function(event) {
  if (event.target.tagName === "LI") {
    console.log("Clicked item:", event.target.textContent);
  }
});
```

   This way, you don't need a separate listener for each `<li>`, which is more efficient.

5. **What are cookies, `localStorage`, and `sessionStorage`, and how do they differ?**

   - **Cookies** store small pieces of data (up to ~4KB) that are sent to the server with each HTTP request (unless marked `HttpOnly`). They have expiration dates and are domain-specific. Cookies can be persistent or session-only.
   - `localStorage` and `sessionStorage` are Web Storage APIs that store key-value pairs on the client side. Both are limited (typically 5-10MB) and stored only on the client. The difference is persistence: `localStorage` persists data indefinitely (until cleared), scoped by origin. `sessionStorage` persists data only for the duration of the page session (tab or window) [23]. Once the tab is closed, `sessionStorage` is cleared. As MDN notes, `sessionStorage` is partitioned by origin *and* tab, and data lasts only for the page session [23]. `localStorage` is shared across tabs of the same origin and remains until manually cleared or expired by code. Neither storage is automatically sent to the server (unlike cookies).

6. **What is Cross-Origin Resource Sharing (CORS)?**
   CORS is a browser security feature that restricts web pages from making requests to a different domain than the one that served the page (the *same-origin policy*). CORS allows servers to explicitly permit cross-origin requests by sending special headers (`Access-Control-Allow-Origin` and others). For example, if a page at `example.com` wants to fetch data from `api.other.com`, `api.other.com` must include a header like `Access-Control-Allow-Origin: https://example.com` (or `*`) in its response. Otherwise, the browser blocks the request.

7. **What is the difference between `window.onload` and `DOMContentLoaded`?**
   `DOMContentLoaded` fires when the HTML has been fully parsed and the DOM tree is built, but

before external resources like images or stylesheets may have loaded. You can use `document.addEventListener("DOMContentLoaded", ...)` to run code as soon as the DOM is ready. In contrast, `window.onload` fires only after *everything* on the page loads, including images and CSS. Thus, for most dynamic script initialization (that only needs the DOM), `DOMContentLoaded` is sufficient and faster.

8. **Explain the browser rendering process briefly.**
   When a page loads, the browser downloads HTML and builds the DOM, then downloads CSS and constructs the CSS Object Model (CSSOM). It combines DOM and CSSOM into a *render tree*. It then performs layout (calculates positions/sizes) and paints pixels on the screen. JavaScript that manipulates the DOM or CSS can cause reflows (recalculations of layout) and repaints, which are expensive. This is why modifying DOM in batches or using techniques like `requestAnimationFrame`, or virtual DOM libraries, can improve performance.

# JavaScript Problem-Solving and Coding

1. **How do you reverse a string in JavaScript?**
   One common approach is to split the string into an array of characters, reverse the array, then join it back:

```
function reverseString(str) {
  return str.split('').reverse().join('');
}
console.log(reverseString("hello")); // "olleh"
```

   Note that this works for simple strings; handling grapheme clusters (like emojis) can be more complex. For basic interview purposes, this solution suffices.

2. **How would you check if a string is a palindrome?**
   A palindrome reads the same forwards and backwards. One way is to reverse the string (as above) and compare to the original:

```
function isPalindrome(s) {
  const cleaned = s.replace(/[^a-z0-9]/gi, '').toLowerCase();
  return cleaned === cleaned.split('').reverse().join('');
}
console.log(isPalindrome("A man, a plan, a canal: Panama")); // true
```

   This example ignores non-alphanumeric characters and case. The core idea is comparing `s` to its reverse.

3. **Implement FizzBuzz in JavaScript.**
   Print numbers 1 to N with “Fizz” for multiples of 3, “Buzz” for multiples of 5, and “FizzBuzz” for multiples of both:

```
function fizzBuzz(n) {
  for (let i = 1; i <= n; i++) {
    let out = "";
    if (i % 3 === 0) out += "Fizz";
    if (i % 5 === 0) out += "Buzz";
    console.log(out || i);
  }
}
fizzBuzz(15);
// Output: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13,
14, FizzBuzz
```

4. **Write a function to flatten a nested array (to one level).**
   ES2019 introduced `Array.prototype.flat()`, which does this. For example, `[1, [2, 3]].flat()` yields `[1,2,3]`. According to MDN, `flat()` "creates a new array with all sub-array elements concatenated into it" up to a given depth [24]. Example:

```
const nested = [0, 1, [2, 3], [4, [5]]];
console.log(nested.flat());    // [0, 1, 2, 3, 4, [5]]
console.log(nested.flat(2));   // [0, 1, 2, 3, 4, 5]
```

   If you need to support older environments, you can use `reduce` and recursion:

```
function flatten(arr) {
  return arr.reduce((acc, val) =>
    Array.isArray(val) ? acc.concat(flatten(val)) : acc.concat(val),
[]);
}
```

5. **How can you remove duplicates from an array?**
   A concise way is to use a `Set`, which only stores unique values. For example:

```
const arr = [1, 2, 3, 2, 4, 1];
const uniqueArr = [...new Set(arr)];
console.log(uniqueArr); // [1, 2, 3, 4]
```

   The `Set` object "lets you store unique values of any type" [25]. By spreading the set back into an array, you get the unique elements. Alternatively, you can use `filter`:

```
const unique = arr.filter((v, i, a) => a.indexOf(v) === i);
```

6. **Explain debouncing and throttling, and give a use case.**
   These are optimization techniques for frequently-called functions (like on scroll or keypress).

   - **Debouncing** delays a function call until after a specified idle time. If the event keeps firing, the function doesn't run until the user stops triggering for the given delay [26]. E.g., waiting 300ms after the last keystroke before performing a search.
   - **Throttling** ensures a function is called at most once in a given time interval, regardless of how many times the event fires [27]. For instance, handling window resize or scroll events at most once every 100ms to reduce load.
     In code, a debounce implementation:

     ```
     function debounce(func, delay) {
       let timerId;
       return function(...args) {
         clearTimeout(timerId);
         timerId = setTimeout(() => func.apply(this, args), delay);
       };
     }
     // Usage
     window.addEventListener('resize', debounce(() => {
       console.log('Resized!');
     }, 200));
     ```

     This runs the callback only after resizing has stopped for 200ms [26]. Throttle is implemented differently (calling the function immediately then blocking further calls for the interval).

7. **How do you implement a simple event emitter (pub/sub) in JavaScript?**
   A basic pattern: maintain a map of event names to listener arrays, and provide `on`, `off`, and `emit` methods. For example:

   ```
   class EventEmitter {
     constructor() { this.events = {}; }
     on(event, listener) {
       (this.events[event] || (this.events[event] = [])).push(listener);
     }
     off(event, listener) {
       this.events[event] = this.events[event].filter(l => l !== listener);
     }
     emit(event, ...args) {
       (this.events[event] || []).forEach(l => l(...args));
     }
   }
   // Usage:
   const emitter = new EventEmitter();
   emitter.on('greet', name => console.log(`Hello, ${name}!`));
   emitter.emit('greet', 'Alice'); // "Hello, Alice!"
   ```

   This pattern underlies Node.js's EventEmitter.

8. **How would you merge two arrays or objects?**
   For arrays, you can use `concat` or spread:

```
const a = [1,2], b = [3,4];
const merged = [...a, ...b]; // [1,2,3,4]
```

   For objects, use `Object.assign()` or spread in ES2018+:

```
const o1 = {x:1, y:2}, o2 = {z:3};
const mergedObj = {...o1, ...o2}; // {x:1, y:2, z:3}
```

   If keys overlap, later values overwrite earlier ones.

9. **How do you deep clone an object in JavaScript?**
   A common (but limited) way is using `JSON`: `const clone = JSON.parse(JSON.stringify(obj))`. This breaks if the object has functions, dates, `undefined`, or non-JSON data. In modern browsers, `structuredClone(obj)` also performs deep clone for many types. For custom classes or edge cases, you might write a recursive clone function. For example (simple version):

```
function deepClone(obj) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (Array.isArray(obj)) return obj.map(deepClone);
  const copy = {};
  for (const key in obj) {
    copy[key] = deepClone(obj[key]);
  }
  return copy;
}
```

10. **What is the difference between `==` and `===`?**
    *(This was already covered in question 2.)*

11. **How do you check if a variable is an array or an object?**
    Use `Array.isArray(value)` to check for arrays. For objects, one common check is `typeof value === "object" && value !== null && !Array.isArray(value)`. Be cautious: in JS, `typeof null` is `"object"`. For more specific checks, you can use `instanceof`: e.g. `value instanceof Array` or `value instanceof Object`, but this can fail across iframes or window contexts. MDN's recommendation for arrays is `Array.isArray`.

12. **Explain the difference between `forEach`, `map`, and `filter`.**

    - `forEach(callback)`: Iterates over an array and calls `callback(item, index, array)` for each element. It returns `undefined`; you typically use it for side effects (e.g., logging or modifying elements in place).

- `map(callback)`: Creates a new array by applying `callback(item)` to each element of the original array. The resulting array has the same length. Use `map` when you want to transform each element.
- `filter(callback)`: Creates a new array containing only the elements for which `callback(item)` returns `true`. The output array may be shorter. Use `filter` to select a subset of elements.
  Example:

```
const nums = [1, 2, 3, 4];
nums.forEach(x => console.log(x * 2)); // logs 2,4,6,8; returns
undefined
const doubled = nums.map(x => x * 2); // [2,4,6,8]
const evens = nums.filter(x => x % 2 === 0); // [2,4]
```

13. **How would you debounce a function in code? Provide an example.**
*(See question 39 explanation above for concept and code.)* A full example:

```
function debounce(func, delay) {
  let timerId;
  return function(...args) {
    clearTimeout(timerId);
    timerId = setTimeout(() => func.apply(this, args), delay);
  };
}
// Usage:
window.addEventListener('resize', debounce(() => {
  console.log('Resized!');
}, 300));
```

Here, the resize handler runs only after 300ms of no resize events, preventing it from firing continuously.

14. **What does `typeof null` return and why?**
`typeof null` returns `"object"`. This is a well-known quirk of JavaScript: `null` is considered a primitive, but the original implementation erroneously classified it as an object type. It's a historical bug, and to check for null you must explicitly compare (`value === null`).

15. **How do you handle JSON data in JavaScript?**
You can convert objects to JSON strings using `JSON.stringify(obj)` and parse JSON strings to objects with `JSON.parse(jsonString)`. For example:

```
const obj = {x: 5, y: 10};
const str = JSON.stringify(obj); // '{"x":5,"y":10}'
const newObj = JSON.parse(str);   // {x:5, y:10}
```

This is commonly used for data interchange (e.g., sending data to/from a server).

16. **Explain function currying.**
    Currying is transforming a function of multiple arguments into a sequence of functions each taking a single argument. For example, a function `f(a, b)` could be curried into `f(a)(b)`. It allows partial application of functions. Example:

```
function add(a) {
  return function(b) {
    return a + b;
  };
}
const addFive = add(5);
console.log(addFive(10)); // 15
```

    Currying can make functions more reusable and is common in functional programming.

17. **Describe the difference between synchronous and asynchronous loops (e.g., `for` vs `for await`).**
    Standard loops (`for`, `forEach`) run synchronously. An `async` function inside a `for` loop can use `await`, but the loop won't wait unless you explicitly handle it (e.g., using `await` inside). ES2020 introduced the `for await (... of ...)` syntax to iterate over async iterators. This allows using `await` on each item in a streaming fashion. For example:

```
async function* getData() {
  yield await fetchData(1);
  yield await fetchData(2);
}
(async () => {
  for await (const item of getData()) {
    console.log(item);
  }
})();
```

    In general, asynchronous operations in loops need care: you might use `Promise.all` or `async/await` inside a standard loop to control execution order.

18. **What happens if you `await` a non-promise value?**
    If you `await` a non-promise, it's converted to a resolved promise of that value. For example, `await 5` will immediately resolve to `5`. The `await` keyword always returns a promise, so `await value` is like `Promise.resolve(value)`. This means you can safely `await` any expression, but only inside an `async` function.

19. **How do you perform error handling in `async/await` versus callbacks?**
    With callbacks, errors are often passed as the first argument to the callback (Node-style), or you check for error conditions manually. With promises and `async/await`, you use `.catch()` or `try/catch`. In `async/await`, wrap your code in `try { ... } catch(err) { ... }`, as shown in question 21. This allows handling both synchronous and asynchronous errors in one place.

20. **How can you convert a callback-based function to return a promise?**
    You can wrap it in a `new Promise`. For example, converting `setTimeout`:

```
function wait(ms) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("Done waiting");
    }, ms);
  });
}
wait(1000).then(msg => console.log(msg)); // Logs after 1 sec
```

    For Node-style callbacks (`function(err, data)`), you can use `util.promisify` (in Node.js) or manually wrap:

```
function callbackFunc(arg, callback) {
  // ...
}
function promiseFunc(arg) {
  return new Promise((resolve, reject) => {
    callbackFunc(arg, (err, result) => {
      if (err) reject(err);
      else resolve(result);
    });
  });
}
```

21. **What is `Promise.reject()` and `Promise.resolve()`?**
    These are static methods to create resolved/rejected promises immediately. `Promise.resolve(value)` returns a promise resolved with `value`; if `value` is a promise, it returns that promise (unless inside an `async` function where it still wraps). `Promise.reject(err)` returns a promise that immediately rejects with `err`. They are useful for returning promises from functions.

22. **How do you use the Fetch API for AJAX calls?**
    The Fetch API provides `fetch(url, options)` that returns a promise resolving to a response. Basic usage:

```
fetch('/api/data')
  .then(response => {
    if (!response.ok) throw new Error("HTTP error " + response.status);
    return response.json(); // parse JSON body
  })
  .then(data => console.log(data))
  .catch(err => console.error("Fetch error:", err));
```

    With `async/await`, you can write:

```
async function getData() {
  try {
    const res = await fetch('/api/data');
    if (!res.ok) throw new Error(`HTTP error ${res.status}`);
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error("Fetch error:", err);
  }
}
getData();
```

By default, `fetch` does not reject on HTTP errors (only on network failure), so you should check `response.ok`. The Fetch API replaces older `XMLHttpRequest`.

23. **How do you throttle a scroll event handler to improve performance?**
Use a throttling technique. For example, limit calls to once every 100ms:

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  }
}
window.addEventListener('scroll', throttle(() => {
  console.log('Scrolling...');
}, 100));
```

This ensures the scroll handler runs at most once per 100ms, reducing load during continuous scrolling.

24. **Write code to find the maximum element in an array.**
You can use `Math.max` with spread:

```
const arr = [5, 1, 9, 3];
const max = Math.max(...arr); // 9
```

Or loop:

```
function arrayMax(a) {
  let max = a[0];
  for (let v of a) {
    if (v > max) max = v;
```

```
  }
  return max;
}
```

25. **Explain how event handlers work with `addEventListener` and remove them.**
    When you use `element.addEventListener(event, handler)`, the handler is registered to be called when the event occurs. To remove it, use `removeEventListener` with the same function reference:

    ```
    function handleClick() { ... }
    btn.addEventListener("click", handleClick);
    // Later:
    btn.removeEventListener("click", handleClick);
    ```

    Removing listeners prevents memory leaks and unwanted behavior. Note: anonymous functions cannot be removed since you need the original reference.

26. **How do `setTimeout` and `setInterval` differ?**

    - `setTimeout(func, ms)` schedules `func` to run *once* after `ms` milliseconds.
    - `setInterval(func, ms)` schedules `func` to run *repeatedly* every `ms` milliseconds.
      Both return an ID that can be used to cancel them: `clearTimeout(id)` or `clearInterval(id)`. Internally, both rely on the event loop: the callback runs only after the timeout elapses and the call stack is clear.

27. **What are some ways JavaScript code can cause memory leaks?**
    Common sources of leaks include:

    - **Global variables**: Undeclared assignments (`x = 5;`) create globals that persist.
    - **Forgotten timers/callbacks**: An active `setInterval` or DOM event listener holding references to closures can prevent garbage collection. Always clear timers and remove listeners when no longer needed.
    - **Closures holding large structures**: If a closure (e.g., a callback) captures variables that reference big objects, those objects stay in memory as long as the closure exists.
    - **Detached DOM nodes**: If you remove a DOM element but still have a reference to it in JS, it won't be garbage-collected.
      Good practice (using tools or linting, limiting scopes, clearing resources) can mitigate leaks.

*Sources:* Explanations and examples are drawn from JavaScript documentation and expert resources [1] [8] [10] [24] [25] [26] [16].

---

[1] [2] [3] [7] [8] [9] [15] [19] [20] JavaScript Interview Questions and Answers | GeeksforGeeks
https://www.geeksforgeeks.org/javascript-interview-questions/

[4] [5] [6] Grammar and types - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types

10 Arrow function expressions - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

11 12 13 Explain call(), apply(), and bind() methods in JavaScript | GeeksforGeeks
https://www.geeksforgeeks.org/explain-call-apply-and-bind-methods-in-javascript/

14 Inheritance and the prototype chain - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain

16 Template literals (Template strings) - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

17 Default parameters - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

18 Classes - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

21 22 async function - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

23 Window: sessionStorage property - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage

24 Array.prototype.flat() - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flat

25 Set - JavaScript | MDN
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

26 27 Difference between Debouncing and Throttling | GeeksforGeeks
https://www.geeksforgeeks.org/difference-between-debouncing-and-throttling/