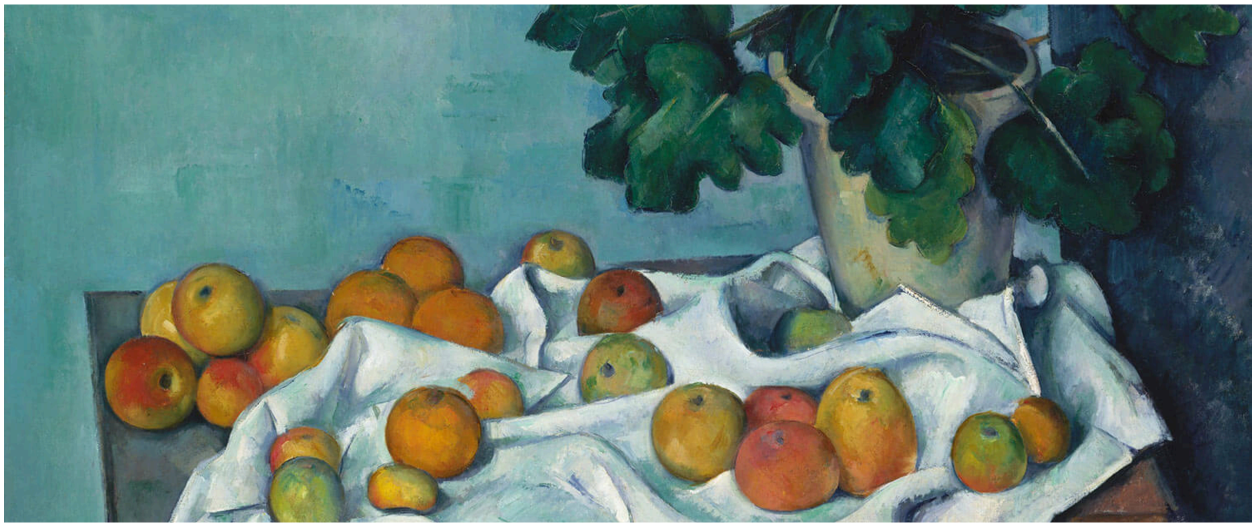

JavaScript (Day13-Day18) - Interview Questions

Compiled by | Ⓚ Kumar Nayan

❓ Q #1 (🟢 Basic)

Tags: 🧩 General Interview , Destructuring , Arrays

🧠 **Question:**

What is array destructuring and how does it simplify variable assignment?

✅ **Answer:**

Array destructuring is a JavaScript feature that allows you to "unpack" values from an array into distinct variables in a single statement. It simplifies code by avoiding the need to access each element by its index individually.

```javascript
// Old way
let fruits = ["apple", "banana", "cherry"];
let first = fruits[0];
let second = fruits[1];

// With destructuring
let [firstFruit, secondFruit] = fruits;
console.log(firstFruit);  // 'apple'
console.log(secondFruit); // 'banana'
```

❓ Q #2 (🟢 Basic)

Tags: 🧩 `General Interview`, `Destructuring`, `Objects`

🧠 Question:

How does object destructuring work, and how do you provide default values?

✅ Answer:

Object destructuring unpacks properties from an object into variables with the same name as the keys. You can provide a default value using the `=` operator, which is used only if the property is `undefined` in the object.

```
javascriptlet person = { name: "Alice" };  // age is missing
let { name, age = 25 } = person;

console.log(name);   // 'Alice'
console.log(age);   // 25 (default value is used)
```

❓ Q #3 (🟡 Intermediate)

Tags: 🚀 `MAANG-level`, `Destructuring`, `Objects`

🔥 EXTREMELY IMPORTANT

🧠 Question:

Explain how to destructure nested objects and rename variables at the same time.

✅ Answer:

You can destructure nested objects by mirroring the object's structure. To rename a variable, you use a colon (`:`) followed by the new variable name. This is useful for avoiding naming conflicts or creating more descriptive variable names.

```
javascriptlet user = {
 id: 1,
 info: {
   name: "Bob",
   address: { city: "New York" },
 },
};

// Destructure `name` and `city`, renaming `name` to `userName`
let { info: { name: userName, address: { city } } } = user;
```

```
console.log(userName);   // 'Bob'
console.log(city);       // 'New York'
```

❓ Q #4 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , `Spread Operator` , `Rest Parameters`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is the difference between the spread syntax (`...`) and the rest parameter syntax (`...`)?

✅ **Answer:**

Although they use the same `...` syntax, they serve opposite purposes:

- **Spread Syntax** "expands" an iterable (like an array or object) into individual elements. It's used for making copies or combining data.
- **Rest Parameter** "gathers" the remaining elements or properties into a new array or object. It's used in function arguments or during destructuring to collect leftover values.

```
javascript  // Spread: expands [1, 2] into individual elements
let arr1 = [1, 2];
let arr2 = [...arr1, 3, 4];  // Result: [1, 2, 3, 4]// Rest: gathers remaining elements [2, 3, 4] into an array
let [first, ...rest] = [1, 2, 3, 4];
console.log(rest);  // Result: [2, 3, 4]
```

❓ Q #5 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , `Immutability` , `Shallow Copy` , `Spread Operator`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

When using the spread operator to copy an object, does it create a shallow or deep copy? Explain the risks associated with this.

✅ **Answer:**

The spread operator performs a **shallow copy**. It copies the top-level properties of the object into a new object. However, if any of those properties are nested objects or arrays, it only copies the *reference* (the memory address) to them, not the nested objects themselves.

The risk is that modifying a nested object in the copied version will also mutate the original object, leading to unintended side effects and bugs.

```
javascriptlet original = { name: "Dave", details: { age: 40 } };
let shallowCopy = { ...original };

// Modifying the nested object in the copy
shallowCopy.details.age = 41;

// The original object is also changed, which is often a bug!
console.log(original.details.age);  // 41
```

❓ Q #6 (🟡 Intermediate)

Tags: 🧩 General Interview , Functions , Destructuring , Rest Parameters

🧠 **Question:**

Explain how you can use destructuring and rest parameters in a function's arguments to handle complex inputs.

✅ **Answer:**

You can destructure an object directly in the function's parameter list to pull out specific properties into variables. The rest parameter (`...`) can then be used to collect all other properties into a separate object. This makes the function signature clean and flexible.

```
javascript  // This function expects an object with fullName and age, and gathers all other properties.
function processUser({ fullName, age = 18, ...otherDetails }, ...hobbies) {
  return `${fullName} is ${age} and likes ${hobbies.join(", ")}. Extra info: ${JSON.stringify(otherDetails)}`;
}

let output = processUser(
  { fullName: "Nayan", age: 26, job: "Developer", city: "Bengaluru" },
  "Coding", "Movies"
);

// output: "Nayan is 26 and likes Coding, Movies. Extra info: {"job":"Developer","city":"Bengaluru"}"
```

❓ Q #7 (🟢 Basic)

Tags: 🧩 General Interview , Modern ES Features , Error Handling

🧠 **Question:**

What problem does Optional Chaining (`?.`) solve in JavaScript?

✅ **Answer:**

Optional Chaining (`?.`) solves the problem of trying to access properties on `null` or `undefined` values, which would normally throw a `TypeError`. It allows you to safely read nested properties without crashing your code. If any part of the chain is `null` or `undefined`, the expression short-circuits and returns `undefined` instead of throwing an error.

```
javascriptconst user = {
  // name: { first: "John" } // name property is missing
};

// Without optional chaining, this would crash// const firstName = user.name.first; // TypeError: Cannot read properties of undefined// With optional chaining, it safely returns undefined
const firstName = user?.name?.first;
console.log(firstName);  // undefined
```

❓ Q #8 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , `Modern ES Features` , `Logical Operators`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is the key difference between the Nullish Coalescing operator (`??`) and the Logical OR operator (`||`)?

✅ **Answer:**

The key difference is how they treat "falsy" values.

- `||` **(Logical OR)** returns the right-hand side if the left-hand side is any *falsy* value (`false` , `0` , `''` , `null` , `undefined` , `NaN`).
- `??` **(Nullish Coalescing)** only returns the right-hand side if the left-hand side is *nullish* (`null` or `undefined`). It treats other falsy values like `0` and `''` as valid.

This makes `??` safer for setting defaults when `0` or an empty string are valid inputs.

```
javascriptlet valueFromAPI = 0;

```

`// Using ||, 0 is falsy, so it incorrectly uses the default.`
`let setting1 = valueFromAPI || 10;` `// setting1 is 10// Using ??, 0 is not nullish, so it's considered a valid value.`
`let setting2 = valueFromAPI ?? 10;` `// setting2 is 0`

❓ Q #9 (`🟡 Intermediate`)

Tags: 🧩 `General Interview` , `Modern ES Features` , `Assignment Operators`

🧠 **Question:**

Explain what a logical assignment operator like `??=` does and provide a use case.

✅ **Answer:**

Logical assignment operators combine a logical operation with an assignment. The `??=` operator, or "nullish assignment operator," assigns the value on the right to the variable on theleft *only if* the left-hand variable is `null` or `undefined` .

It's a concise way to set a default value for a variable that might not have been initialized yet.

`javascript` `// Instead of writing this:// if (user.name === null || user.name === undefined) {//   user.name = "Guest";// }`

`let user = { age: 30 };` `// user.name is undefined// Use ??= for a clean, single-line default assignment`
`user.name ??= "Guest";`

`console.log(user.name);` `// "Guest"`

❓ Q #10 (`🟡 Intermediate`)

Tags: 🧩 `General Interview` , `Compatibility` , `Polyfills`

🧠 **Question:**

How would legacy browsers react to modern JavaScript syntax like optional chaining, and what is the purpose of a polyfill?

✅ **Answer:**

Legacy browsers that predate a modern feature (like optional chaining `?.`) do not understand the syntax. When they encounter it, they will throw an immediate `SyntaxError` , crashing the script.

A **polyfill** is a piece of code (like the `core-js` library) that you include in your project to provide a functional implementation of a missing modern feature in an older environment. It allows you to write modern code that can still run on older browsers without crashing.

❓ Q #11 (🟢 Basic)

Tags: 🧩 `General Interview`, `Error Handling`

🧠 **Question:**

What is the purpose of the `try`, `catch`, and `finally` blocks in JavaScript?

✅ **Answer:**

These blocks are used for handling errors in synchronous code:

- `try`: You place code that might throw an error inside this block.
- `catch`: If an error is thrown in the `try` block, execution jumps to the `catch` block, where you can handle the error (e.g., log it or show a user message).
- `finally`: This block runs *always*, regardless of whether an error was thrown or not. It's typically used for cleanup tasks, like closing a file or a database connection.

❓ Q #12 (🔴 Advanced)

Tags: 🚀 `MAANG-level`, `Error Handling`, `Call Stack`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Explain the concept of error propagation, or "bubbling," in JavaScript. How does an uncaught error travel up the call stack?

✅ **Answer:**

When an error is `throw`n, JavaScript looks for the nearest `try...catch` block within the current function's scope to handle it. If it doesn't find one, the error isn't contained. Instead, it **propagates** (or "bubbles up") to the calling function.

This process repeats up the entire call stack. The error travels from the inner function to the outer function, and so on, until it's caught by a `try...catch` block at a higher level or reaches the global scope, which crashes the program.

```
javascriptfunction innerFunction() {
  throw new Error("Problem inside!");   // 1. Error is thrown here.
}

function outerFunction() {
  innerFunction();   // 2. Error bubbles up from innerFunction.
}

try {
  outerFunction();   // 3. Error bubbles up from outerFunction and is caught here.
} catch (error) {
  console.log("Caught at the highest level: " + error.message);
}
```

❓ Q #13 (🟡 Intermediate)

Tags: 🧩 General Interview , Error Handling

🧠 **Question:**

What is the benefit of creating custom Error classes in JavaScript?

✅ **Answer:**

Creating custom Error classes by extending the built-in `Error` class allows you to create more specific and descriptive error types. The benefits include:

1. **Clarity**: You can give errors meaningful names (e.g., `NetworkError` , `ValidationError`).
2. **Extra Information**: You can add custom properties like an error code, status, or metadata.
3. **Specific Handling**: It allows you to use `instanceof` in a `catch` block to handle different types of errors differently.

```
javascriptclass ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
    this.code = 400;   // Add custom property
  }
}

try {
  throw new ValidationError("Email is invalid.");
} catch (error) {
  if (error instanceof ValidationError) {
    console.log(`Validation failed (Code ${error.code}): ${error.message}`);
  }
}
```

❓ Q #14 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , `Asynchronous` , `Promises` , `Error Handling`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

How does error handling in asynchronous code with Promises differ from synchronous `try...catch`?

✅ **Answer:**

Synchronous errors are caught by a `try...catch` block that wraps the code. Asynchronous errors in Promises cannot be caught this way because the promise may reject long after the `try...catch` block has finished executing.

Instead, you handle promise rejections by chaining a `.catch()` method to the promise chain. For `async/await` syntax, you can use a traditional `try...catch` block because `await` pauses the function and makes the asynchronous code behave in a more synchronous-looking manner.

```javascript
// Promise .catch() method
fetch("bad-url")
  .then(response => response.json())
  .catch(error => console.log("Caught with .catch():", error));

// async/await with try...catch
async function fetchData() {
  try {
    let data = await fetch('bad-url');
  } catch (error) {
    console.log("Caught in async function:", error);
  }
}
```

❓ Q #15 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Best Practices` , `Defensive Coding`

🧠 **Question:**

What is a "guard clause" and how does it contribute to writing defensive code?

✅ **Answer:**

A guard clause is a check at the beginning of a function to validate inputs or conditions. If the condition is not met, the function exits early (e.g., by returning).

This pattern contributes to defensive coding by preventing invalid data from running through the main logic, reducing potential errors. It also improves readability by handling edge cases at the top and avoiding deeply nested `if` statements.

```
javascriptfunction getUsername(user) {
  // Guard clause: exit early if user is invalid
 if (!user || !user.name) {
   return "Guest";
 }

  // Main logic proceeds only if data is valid
 return user.name;
}
```

❓ Q #16 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Error Handling` , `Best Practices`

🧠 **Question:**

Explain the concept of re-throwing an error and why it's a useful pattern.

✅ **Answer:**

Re-throwing an error involves catching an error at one level, performing a partial action (like logging), and then throwing it again so it can be handled by a higher-level error handler.

This pattern is useful for creating layered error-handling strategies. A low-level function might log technical details about an error, while a high-level function (e.g., in the UI layer) might catch the re-thrown error to display a user-friendly message. It allows different parts of the application to handle the aspects of an error that are relevant to them.

```
javascriptfunction processData() {
 try {
    // ... some operation fails
```

```
  throw new Error("Initial error");
 } catch (error) {
  console.log("Low-level log:", error.message);  // 1. Log details
  throw error;  // 2. Re-throw for higher-level handling
 }
}

try {
 processData();
} catch (error) {
  // 3. Catch the re-thrown error to show a UI message
 console.log("High-level handle: Informing the user.");
}
```

❓ Q #17 (🟢 Basic)

Tags: 🚀 `MAANG-level`, `Fundamentals`, `Immutability`

🔥 EXTREMELY IMPORTANT

🧠 Question:

Explain the difference between pass-by-value and pass-by-reference in JavaScript.

✅ Answer:

This determines how data is passed to functions or assigned to variables:

- **Pass-by-Value**: Applies to **primitives** (`String`, `Number`, `Boolean`, `null`, `undefined`, `Symbol`). When a primitive is assigned or passed, a **copy of its value** is created. Modifying the copy does not affect the original.
- **Pass-by-Reference**: Applies to **objects** (including arrays and functions). When an object is assigned or passed, a **reference** (or pointer) to the object's location in memory is copied, not the object itself. Modifying the object through one reference will affect all other references to it.

```
javascript  // Pass-by-value (primitives)
let a = 10;
let b = a;  // b gets a copy of the value 10
b = 20;
console.log(a);  // 10 (original is unaffected)// Pass-by-reference (objects)
let obj1 = { name: "Alice" };
let obj2 = obj1;  // obj2 gets a copy of the reference to the same object
obj2.name = "Bob";
console.log(obj1.name);  // "Bob" (original is affected)
```

❓ Q #18 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Immutability` , `Shallow Copy` , `Deep Copy`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Compare `Object.assign()` , the spread syntax (`...`), and `structuredClone()` for copying objects.

✅ **Answer:**

- `Object.assign()` : A function that copies enumerable own properties from source objects to a target object. It performs a **shallow copy**. It can mutate the target object if it's not an empty object `{}` . It is an older ES6 feature.
- **Spread Syntax (`...`)**: Modern syntax for creating a new object or array. It is more concise and generally preferred for creating shallow copies or merging objects. Like `Object.assign()` , it performs a **shallow copy**.
- `structuredClone()` : A modern, built-in function that performs a **deep copy**. It creates a completely independent clone of the original object, including nested structures. It can handle complex data types like `Date` , `Map` , and `Set` , but it cannot clone functions or DOM nodes.

```
javascriptconst original = { a: 1, nested: { b: 2 } };

// Shallow copy with spread
const spreadCopy = { ...original };
spreadCopy.nested.b = 99;  // Mutates original.nested// Deep copy with structuredClone
const deepCopy = structuredClone(original);
deepCopy.nested.b = 55;  // Does NOT mutate original.nested

console.log(original.nested.b);  // 99 (mutated by the shallow copy)
```

❓ Q #19 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Immutability` , `Deep Copy`

🧠 **Question:**

What are the limitations of `structuredClone()` ?

✅ **Answer:**

While powerful for deep copying, `structuredClone()` has several limitations. It will throw a `DataCloneError` if the object contains:

1. **Functions**: Functions are not cloneable as they contain behavior, not just data.
2. **DOM Nodes**: DOM elements cannot be cloned with this method.
3. **Prototypes**: The prototype chain is not preserved; the clone will be a plain object.
4. Other non-serializable objects.

It is designed for cloning structured data, like what you would send over a network as JSON, but with support for more types.

```
javascriptconst objWithFunc = { func: () => console.log("hello") };

try {
  structuredClone(objWithFunc);
} catch (e) {
    // This will throw a DOMException (DataCloneError)
  console.log("Error: Functions are not cloneable.");
}
```

❓ Q #20 (🟡 Intermediate)

Tags: 🧩 General Interview , Immutability , Arrays

🧠 **Question:**

How can you immutably add an element to an array or update an object's property?

✅ **Answer:**

To maintain immutability, you should create a new array or object instead of modifying the original one. The spread syntax is perfect for this.

- **For Arrays**: Spread the original array into a new array literal and add the new element.
- **For Objects**: Spread the original object into a new object literal and add or overwrite the desired property.

```
javascript // Immutable Array Add
const originalArray = [1, 2];
const newArray = [...originalArray, 3]; // [1, 2, 3]// Immutable Object Update
const originalObject = { name: "John", age: 30 };
const updatedObject = { ...originalObject, age: 31 };  // { name: "John", age: 31 }
```

❓ Q #21 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Iterators`

🧠 **Question:**

What makes an object "iterable" in JavaScript, and what is the iterator protocol?

✅ **Answer:**

An object is **iterable** if it implements the iterable protocol, which means it has a property with the key `Symbol.iterator`. This property must be a function that returns an **iterator** object.

The **iterator protocol** defines how an iterator object should behave. It must have a `next()` method that, when called, returns an object with two properties:

- `value`: The next value in the sequence.
- `done`: A boolean that is `false` if there are more values, and `true` when the sequence is complete.

```
javascriptconst fruits = ["apple", "banana"];
const iterator = fruits[Symbol.iterator]();  // Get the iterator

console.log(iterator.next());  // { value: 'apple', done: false }
console.log(iterator.next());  // { value: 'banana', done: false }
console.log(iterator.next());  // { value: undefined, done: true }
```

❓ Q #22 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Iterators` , `Generators`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is a generator function, and how is it different from a regular function?

✅ **Answer:**

A generator function, defined with `function*`, is a special type of function that can be paused and resumed, allowing it to produce a sequence of values over time instead of a single value.

Key differences from a regular function:

1. **Execution**: Regular functions run to completion. Generators can be paused using the `yield` keyword and resumed later.
2. **Return Value**: A regular function returns a single value. A generator function returns an **iterator** (a generator object) which you use to control execution and retrieve values.
3. **`yield` Keyword**: Generators use `yield` to produce a value and pause. Regular functions use `return` to exit and provide a final value.

```
javascriptfunction* countToTwo() {
  yield 1;   // Pause and return 1
  yield 2;   // Pause and return 2
}

const gen = countToTwo(); // Returns a generator object, doesn't execute code yet

console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

❓ Q #23 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Generators` , `Performance`

🧠 **Question:**

Explain the concept of "lazy evaluation" and how generators facilitate it.

✅ **Answer:**

Lazy evaluation is an evaluation strategy where an expression is not evaluated until its result is actually needed. This contrasts with "eager" evaluation, where expressions are computed immediately.

Generators facilitate lazy evaluation because they only compute the next value in a sequence when the `.next()` method is called. The `yield` keyword produces a value and then pauses the function's execution. This is extremely memory-efficient for handling very large or even infinite data sequences, as you only generate the values you currently need, not all of them upfront.

```
javascript // This generator can produce an infinite sequence of numbers
function* infiniteNumbers() {
 let num = 1;
 while (true) {
   yield num++;
 }
}

const nums = infiniteNumbers();
 // The program doesn't crash because numbers are generated lazily
console.log(nums.next().value);  // 1
console.log(nums.next().value);  // 2
```

❓ Q #24 (🔴 Advanced)

Tags: 🚀 MAANG-level , Asynchronous , Generators

🧠 **Question:**

In an async generator, what is the distinct purpose of `await` versus `yield`?

✅ **Answer:**

In an `async function*`, `await` and `yield` serve two distinct pausing purposes:

- `await`: Pauses the generator's execution to wait for an asynchronous operation (like a Promise) to resolve. It's for waiting on external, time-consuming tasks (e.g., a network request).
- `yield`: Pauses the generator's execution to hand a value to the consumer. It's for producing the next item in the sequence and waiting for the consumer to call `.next()` again.

In short: `await` waits for a value to become available, while `yield` waits for the consumer to ask for the next value.

```
 javascriptasync function* fetchData() {
   // await waits for the fetch to complete
 const response = await fetch('some-api-url');
 const data = await response.json();
   // yield gives the processed data to the consumer and pauses
 yield data;
}
```

❓ Q #25 (🟡 Intermediate)

Tags: 🧩 General Interview , Asynchronous , Iterators

🧠 **Question:**

How do you consume values from an async generator?

✅ **Answer:**

You consume values from an async generator using the `for await...of` loop. This loop is specifically designed for async iterables. It automatically handles calling `.next()` on the async iterator and `await`ing the promise that each call returns, making the consumption code clean and readable.

```
javascriptasync function* asyncCounter() {
 yield Promise.resolve(1);
 yield Promise.resolve(2);
 yield Promise.resolve(3);
}

async function consume() {
   // The 'for await...of' loop handles the async iteration
 for await (const num of asyncCounter()) {
   console.log(num);   // Logs 1, then 2, then 3
 }
}
consume();
```

❓ Q #26 (🟡 Intermediate)

Tags: 🧩 `General Interview`, `Memory Management`

🧠 **Question:**

Based on your notes, what is a memory leak in JavaScript, and what are some common causes?

✅ **Answer:**

A memory leak occurs when a program allocates memory but fails to release it when it's no longer needed. Over time, this consumes available memory, potentially slowing down or crashing the application.

Common causes mentioned in the notes include:

- **Closures**: A closure can unintentionally keep references to variables in an outer scope, preventing them from being garbage collected.
- **DOM References**: Keeping references to DOM elements that have been removed from the page.

- **Event Listeners**: Failing to remove event listeners from objects that are no longer in use. The listener holds a reference to the object.
- **Timers**: Un-cleared `setInterval` or `setTimeout` calls can keep references alive.
- **Caches**: If a cache grows indefinitely without a mechanism to clear old entries.

❓ Q #27 (🟣 Expert)

Tags: 🚀 `MAANG-level` , `Memory Management` , `Garbage Collection`

🧠 **Question:**

Your notes mention `WeakSet` and `WeakMap`. How do weak references differ from strong references, and how do they help prevent memory leaks?

✅ **Answer:**

A **strong reference** is a standard reference that keeps an object "alive" in memory. As long as a strong reference to an object exists, the garbage collector cannot reclaim its memory.

A **weak reference**, used by `WeakSet` and `WeakMap`, does *not* prevent an object from being garbage collected. If an object is only held by weak references, the garbage collector is free to destroy it and remove its memory.

This is crucial for preventing memory leaks in scenarios like caching or managing metadata for objects. If you store an object in a `WeakMap`, and all other references to that object are removed, the garbage collector will automatically remove it from the `WeakMap`, preventing the map from holding onto obsolete objects.

❓ Q #28 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Immutability` , `Objects`

🧠 **Question:**

What is the difference between `Object.freeze()` and `Object.seal()`?

✅ **Answer:**

Both methods provide a level of immutability, but they differ in strictness:

- `Object.seal()`: Prevents adding or removing properties from an object. You can still **change the value** of existing properties.
- `Object.freeze()`: Is stricter. It prevents adding, removing, AND **changing** properties. It effectively makes the object's top-level properties read-only.

Both are shallow, meaning they only apply to the object's immediate properties, not to nested objects.

```
javascriptconst sealedObj = { a: 1 };
Object.seal(sealedObj);
sealedObj.a = 2;   // This is allowed
delete sealedObj.a;   // This is NOT allowed

const frozenObj = { b: 1 };
Object.freeze(frozenObj);
frozenObj.b = 2;   // This is NOT allowed
```

❓ Q #29 (🟡 Intermediate)

Tags: 🧩 `General Interview`, `Immutability`, `Arrays`

🧠 **Question:**

The array methods `.sort()` and `.reverse()` mutate the original array. How would you perform these operations immutably?

✅ **Answer:**

To perform a sort or reverse operation immutably, you must first create a copy of the original array and then apply the mutating method to the copy. The easiest way to create the copy is with the spread syntax (`...`). This leaves the original array untouched.

```
javascriptconst original = [3, 1, 2];

// The original array is mutated// original.sort(); // original becomes [1, 2, 3]// Immutable sort: create a copy first
const sortedCopy = [...original].sort();

console.log(original);    // [3, 1, 2] (unchanged)
console.log(sortedCopy);   // [1, 2, 3]
```

❓ Q #30 (🟡 Intermediate)

Tags: 🚀 `MAANG-level`, `Immutability`, `Objects`, `Destructuring`

🧠 **Question:**

How can you immutably remove a property from an object?

✅ **Answer:**

The `delete` keyword mutates an object, which should be avoided in immutable patterns. The modern, immutable way to remove a property is to use destructuring combined with the rest syntax. You extract the property you want to remove into a variable and collect the remaining properties into a new object.

```
javascriptconst user = { id: 123, name: "Admin", email: "admin@test.com" };

// Destructure 'email' out, and collect the rest into 'newUser'
const { email, ...newUser } = user;

console.log(user);     // { id: 123, name: 'Admin', email: 'admin@test.com' } (original is unchanged)
console.log(newUser);  // { id: 123, name: 'Admin' } (new object without email)
```

❓ Q #31 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Promises` , `Error Handling` , `Best Practices`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is the risk of "swallowing" an error in a promise chain?

✅ **Answer:**

"Swallowing" an error means catching a rejection in a promise chain but failing to handle it or re-throw it properly. This is dangerous because it can make the application appear to succeed when it has actually failed silently.

The risks are severe:

- **Silent Failures**: The application continues in an unpredictable or broken state without any indication that something went wrong.
- **Debugging Difficulty**: Bugs become extremely hard to trace because there are no error logs or signals pointing to the root cause.
- **Inconsistent State**: The application's state can become corrupted, leading to further, more confusing errors down the line.

You should always have a `.catch()` at the end of a promise chain to log or handle any potential rejections.

❓ Q #32 (🟢 Basic)

Tags: 🧩 `General Interview` , `Defensive Coding` , `Debugging`

🧠 **Question:**

What does `console.assert()` do and how can it be used for defensive coding?

✅ **Answer:**

The `console.assert(assertion, message)` method is a defensive tool that writes a message to the console *only if* the `assertion` (the first argument) evaluates to `false`. If the assertion is `true`, it does nothing.

It's used to check for assumptions or "invariants" in your code—conditions that you expect to always be true. If an assumption is ever violated during development, the assertion will fail and log a message, immediately alerting you to the problem.

```
javascriptfunction calculateBonus(salary, rating) {
  // Defensive check: Assert that the rating is within the expected range.
  console.assert(rating >= 1 && rating <= 5, "Rating must be between 1 and 5");

  // ... main logic
  return salary * (rating / 10);
}

calculateBonus(50000, 7); // Logs: "Assertion failed: Rating must be between 1 and 5"
```

❓ Q #33 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Spread Operator` , `Objects`

🧠 **Question:**

According to your notes, the spread operator was initially for iterables only. What changed in ES2018 regarding its use with objects?

✅ **Answer:**

Before ES2018, the spread syntax (`...`) was only valid for iterables like Arrays, Strings, Maps, and Sets. Attempting to use it on a plain object would cause a

`TypeError` because objects are not inherently iterable.

The ES2018 (ES9) update extended the spread syntax to work with objects for the purpose of cloning and merging. Even though objects are still not iterable (you can't use them in a `for...of` loop directly), the spread syntax was adapted to copy an object's own enumerable properties into a new object.

```
javascriptconst obj1 = { a: 1, b: 2 };

// This is now valid since ES2018
const obj2 = { ...obj1, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 }
```

❓ Q #34 (🟢 Basic)

Tags: 🧩 `General Interview` , `Modern ES Features` , `Arrays`

🧠 **Question:**

What is the purpose of the `Array.prototype.findLast()` method introduced in ES2023?

✅ **Answer:**

The `findLast()` method works just like `find()`, but it iterates the array from the last element to the first. It returns the value of the **first element** that satisfies the testing function, starting from the end of the array.

This is useful for performance when you expect the element you're looking for to be near the end of the array, as it can find the item more quickly without having to search from the beginning.

❓ Q #35 (🟣 Expert)

Tags: 🚀 `MAANG-level` , `Modern ES Features` , `Promises` , `Asynchronous`

🧠 **Question:**

What problem does `Promise.withResolvers()` from ES2024 solve?

✅ **Answer:**

`Promise.withResolvers()` is a factory function that creates a promise and gives you access to its `resolve` and `reject` functions externally.

Normally, the `resolve` and `reject` functions are only accessible inside the promise's constructor callback. This new feature is useful for advanced asynchronous patterns where you need to create a promise but control its state from an outside scope, such as wrapping event-based APIs or managing complex async flows.

```
javascript // This is a conceptual example of its use
function getDelayedValue() {
  const { promise, resolve, reject } = Promise.withResolvers();

  // The resolve function can now be passed around or called later
  setTimeout(() => {
    resolve("Data has arrived");
  }, 2000);

  return promise;
}

const myPromise = getDelayedValue();
myPromise.then(console.log); // Logs "Data has arrived" after 2 seconds
```

❓ Q #36 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , `Immutability` , `Deep Copy`

🧠 **Question:**

When is a deep clone necessary over a shallow copy?

✅ **Answer:**

A deep clone is necessary whenever you have a nested data structure (e.g., an object containing other objects or arrays) and you need to ensure that the copy is completely independent from the original.

You must use a deep clone if you intend to modify any part of the nested structure in the copy without causing side effects in the original object. Shallow copies are only safe for flat data structures or when you intentionally want to share nested references.

Common use cases include state management in frameworks like React, undo/redo functionality, and safely manipulating data fetched from an API.

❓ Q #37 (🟢 Basic)

Tags: 🧩 `General Interview` , `Fundamentals` , `Immutability`

🧠 **Question:**

When does modifying one object unintentionally affect another?

✅ **Answer:**

Modifying one object unintentionally affects another when both variables are pointing to the **same reference** in memory. This happens when you assign an object to another variable or when you create a shallow copy of an object that contains nested objects.

Because both variables hold a reference to the same underlying object, a change made through one variable is visible through the other.

```
javascriptlet obj1 = { details: { status: 'active' } };

// obj2 gets a reference to the same object as obj1
let obj2 = obj1;

// Modifying the object via obj2
obj2.details.status = 'inactive';

// The change is reflected in obj1 because they point to the same object
console.log(obj1.details.status); // 'inactive'
```

❓ Q #38 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , `Iterators`

🧠 **Question:**

How do you make a custom object iterable so it can be used in a `for...of` loop?

✅ **Answer:**

To make a custom object iterable, you must implement the iterable protocol by adding a method with the key `Symbol.iterator` to the object. This method must return an iterator object, which itself must have a `next()` method.

The `next()` method should return an object containing a `value` and a `done` property on each call.

```
javascriptconst myCustomObject = {
 data: ['a', 'b', 'c'],
 [Symbol.iterator]() {
   let index = 0;
   return {
     next: () ⇒ {
      if (index < this.data.length) {
        return { value: this.data[index++], done: false };
      } else {
        return { value: undefined, done: true };
      }
    }
  };
 }
};

// Now the object is iterable
for (const item of myCustomObject) {
  console.log(item); // a, b, c
}
```

❓ Q #39 (🟢 Basic)

Tags: 🧩 `General Interview`, `Destructuring`

🧠 **Question:**

Is it possible to rename variables when destructuring an array? Explain why or why not.

✅ **Answer:**

No, it is not possible to rename variables during array destructuring in the same way you can with object destructuring.

The reason is that array destructuring is based on position (index), not keys. The syntax `let [a, b] = arr` assigns values based on their order. Object destructuring, `let {name: newName} = obj`, works because it maps a named key (`name`) to a new variable (`newName`). Since arrays use numerical indices, there is no named key to rename from.

❓ Q #40 (🟢 Basic)

Tags: 🧩 `General Interview`, `Destructuring`, `Arrays`

🧠 **Question:**

What happens if you try to destructure an array with more or fewer variables than there are elements in the array?

✅ **Answer:**

- **Fewer Variables**: If you provide fewer variables than elements, the remaining elements in the array are simply ignored.
- **More Variables**: If you provide more variables than elements, the extra variables that do not have a corresponding element in the array are assigned the value `undefined`.

```
javascriptconst items = [10, 20];

// Fewer variables: 'c' is ignored
let [a] = [10, 20, 30];
console.log(a);   // 10// More variables: 'c' becomes undefined
let [x, y, z] = items;
console.log(z);   // undefined
```

❓ Q #41 (🟢 Basic)

Tags: 🧩 `General Interview` , `Modern ES Features` , `Strings`

🧠 **Question:**

What is the advantage of `String.prototype.replaceAll()` over `String.prototype.replace()` ?

✅ **Answer:**

The `String.prototype.replace()` method, when used with a string as the first argument, only replaces the *first* occurrence of that string. To replace all occurrences, you would need to use a regular expression with the global flag (`/g`).

`String.prototype.replaceAll()` , introduced in ES2021, simplifies this by replacing *all* occurrences of a substring without needing a regular expression.

❓ Q #42 (🟢 Basic)

Tags: 🧩 `General Interview` , `Modern ES Features` , `Readability`

🧠 **Question:**

What is the purpose of numeric separators (`_`) in JavaScript?

✅ **Answer:**

Numeric separators (`_`) were introduced in ES2021 to improve the readability of long numeric literals. You can place underscores between digits to act as a visual separator, making large numbers easier to read at a glance. The JavaScript engine completely ignores these underscores.

```javascript
// Instead of this:
const hardToRead = 1000000000;

// You can write this:
const easyToRead = 1_000_000_000;

console.log(hardToRead === easyToRead); // true
```

❓ Q #43 (🟡 Intermediate)

Tags: 🧩 `General Interview`, `Fundamentals`, `Immutability`

🧠 **Question:**

Why does `===` evaluate to `false` in JavaScript?Day13-22Aug.md

✅ **Answer:**

This evaluates to `false` because the strict equality operator (`===`) compares objects (including arrays) by **reference**, not by value.

When you write ``, you are creating a new array object in memory. The expression `===` creates two separate array objects in memory, each with its own unique reference. Since the two references are different, the comparison returns `false`, even though their contents are identical.Day13-22Aug.md

---

❓ Q #44 (🔴 Advanced)

Tags: 🚀 `MAANG-level`, `Immutability`, `Performance`

🧠 **Question:**

What are the performance tradeoffs between mutable and immutable operations?

✅ **Answer:**

- **Mutable Operations**: Are generally faster and use less memory for a single operation because they modify data "in-place" without creating new copies. However, they are risky in complex applications because they can lead to unpredictable side effects and bugs when state is shared.

- **Immutable Operations**: Prioritize safety and predictability. They create copies of data, which can consume more memory and be slightly slower (O(n) time/space for a copy). However, they prevent side effects, simplify state management, and make change detection trivial (a simple reference check `===` is O(1) and tells you if anything has changed).

In large applications, the safety and simplicity of immutability often outweigh the micro-performance cost of creating copies.

❓ Q #45 (🟢 Basic)

Tags: 🧩 `General Interview`, `Immutability`, `Arrays`

🧠 **Question:**

How can you use the `.filter()` method to immutably remove elements from an array?

✅ **Answer:**

The `Array.prototype.filter()` method is inherently immutable. It iterates over an array and returns a **new array** containing only the elements that pass the condition specified in the callback function. The original array is left unchanged. This makes it an ideal tool for immutably removing elements based on a condition.

```
javascriptconst numbers = [1, 2, 3, 4, 5];

// Remove all numbers less than 3
const filteredNumbers = numbers.filter(num => num >= 3);

console.log(numbers);         // [1, 2, 3, 4, 5] (original is untouched)
console.log(filteredNumbers); // [3, 4, 5] (new array is returned)
```

❓ Q #46 (🟡 Intermediate)

Tags: 🧩 `General Interview`, `Iterators`, `Generators`

🧠 **Question:**

What does a generator function return, and how does that relate to an iterator?

✅ **Answer:**

A generator function (`function*`) does not execute its body immediately. When called, it returns a special object called a **generator object**. This generator object is an **iterator**.

This means you can call `.next()` on the returned generator object to control its execution, and it will produce values according to the iterator protocol (`{ value, done }`). Because generators are also iterables, you can use them directly in `for...of` loops.

❓ Q #47 (🟢 Basic)

Tags: 🧩 `General Interview`, `Iterators`, `Loops`

🧠 **Question:**

What kind of objects can you use with a `for...of` loop?

✅ **Answer:**

You can only use a `for...of` loop with objects that are **iterable**. An object is iterable if it has a `[Symbol.iterator]` method.

By default, built-in types like **Arrays, Strings, Maps, and Sets** are iterable. Plain objects (`{}`) are not iterable by default.

❓ Q #48 (🟢 Basic)

Tags: 🧩 `General Interview`, `Error Handling`

🧠 **Question:**

In a `try...catch...finally` block, when does the `finally` block execute?

✅ **Answer:**

The `finally` block executes **always**, after the `try` and, if applicable, the `catch` blocks have completed. It runs regardless of the outcome:

1. It runs if the `try` block completes successfully.
2. It runs if an error is thrown in the `try` block and caught by `catch`.
3. It even runs if an error is thrown and *not* caught, or if a `return` statement is executed in the `try` or `catch` block.

It is primarily used for cleanup code that must execute no matter what.

❓ Q #49 (🟢 Basic)

Tags: 🧩 `General Interview`, `Error Handling`

🧠 **Question:**

What is the purpose of the `throw` keyword?

✅ **Answer:**

The `throw` keyword is used to manually create and throw a user-defined error. When `throw` is executed, it immediately stops the current execution flow and begins the process of error propagation up the call stack. Execution will then jump to the nearest `catch` block that can handle the error. It's used to signal that an exceptional condition or an unrecoverable error has occurred.

```
javascriptfunction getAge(year) {
 if (year > new Date().getFullYear()) {
   throw new Error("Birth year cannot be in the future.");
 }
 return new Date().getFullYear() - year;
}
```

❓ Q #50 (🟡 Intermediate)

Tags: 🧩 `General Interview`, `Modern ES Features`, `Logical Operators`

🧠 **Question:**

Explain the `||=` (logical OR assignment) operator.

✅ **Answer:**

The `||=` operator is a logical assignment operator. It assigns the value of the right-hand operand to the left-hand operand *only if* the left-hand operand is **falsy** (`false`, `0`, `""`, `null`, `undefined`, `NaN`). If the left-hand operand is truthy, nothing happens. It's a shortcut for `x = x || y`.

```
javascriptlet options = { duration: 0, speed: null };

options.speed ||= 100;     // speed is null (falsy), so it becomes 100
options.duration ||= 10;   // duration is 0 (falsy), so it becomes 10

console.log(options);   // { duration: 10, speed: 100 }
```

❓ Q #51 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Modern ES Features` , `Logical Operators`

🧠 **Question:**

Explain the `&&=` (logical AND assignment) operator.

✅ **Answer:**

The `&&=` operator is a logical assignment operator. It assigns the value of the right-hand operand to the left-hand operand *only if* the left-hand operand is **truthy**. If the left-hand operand is falsy, nothing happens. It's a shortcut for `x = x && y`.

```
javascriptlet user = { name: "Alice", isAdmin: true };
let anotherUser = { name: "Bob" };  // isAdmin is undefined (falsy)// user.isAdmin is truthy, so its value is updated
user.isAdmin &&= false;
console.log(user.isAdmin);  // false// anotherUser.isAdmin is falsy, so no assignment happens
anotherUser.isAdmin &&= true;
console.log(anotherUser.isAdmin);  // undefined
```

❓ Q #52 (🟡 Intermediate)

Tags: 🧩 `General Interview` , `Iterators`

🧠 **Question:**

How would you write a simple factory function that creates a custom iterator for an array?

✅ **Answer:**

You can write a factory function that takes an array as input and returns an iterator object. This object will manage the state (the current index) internally and expose a `next()` method that returns the next value according to the iterator protocol.

```
javascriptfunction makeIterator(arr) {
 let index = 0;
 return {
   next() {
     if (index < arr.length) {
       return { value: arr[index++], done: false };
     } else {
       return { value: undefined, done: true };
     }
   },
 };
}

const iter = makeIterator(["cat", "dog"]);
console.log(iter.next());   // { value: 'cat', done: false }
console.log(iter.next());   // { value: 'dog', done: false }
console.log(iter.next());   // { value: undefined, done: true }
```