



# JavaScript (Day7-Day12) - Interview Questions

Compiled by  Kumar Nayan

## ? Q #1 ( Basic )





Tags:  General Interview ,  Objects ,  Object Creation

 Question:

What are the four primary ways to create an object in JavaScript as described in your notes?

 Answer:

Based on the notes, there are four main ways to create an object:

1. **Object Literal:** The simplest method, using curly braces .
2. **Class Constructor:** Using the  `class` keyword and a  `constructor` method to define a blueprint.
3. **`Object.create()`:** Creates a new object, using an existing object as the prototype of the newly created object.
4. **`new Object()` Constructor:** Using the built-in  `Object` constructor function.

```
javascript // 1. Object Literal
const obj1 = { key: "value" };

// 2. Class Constructor
class Animal {
  constructor(type) { this.type = type; }
}
const cat = new Animal("feline");

// 3. Object.create
const proto = { greet: function() { return "Hi"; } };
const obj3 = Object.create(proto);

// 4. new Object()
const obj4 = new Object();
```

---

## ? Q #2 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Objects , Object Immutability

🧠 Question:

Explain the difference between `Object.freeze()` and `Object.seal()` . What can you still do to a sealed object that you cannot do to a frozen one?

✅ Answer:

Both methods restrict object modifications, but `Object.freeze()` is stricter.

- `Object.freeze(obj)` makes all properties non-writable and non-configurable. You cannot add, delete, or change any properties.
- `Object.seal(obj)` only prevents additions or deletions of properties by making them non-configurable. However, you can still modify the values of existing properties because `writable` remains `true` .

In short, you can change the value of an existing property on a sealed object, but not on a frozen one.

```
javascriptconst sealedObj = { prop: 1 };
Object.seal(sealedObj);
sealedObj.prop = 2; // This works!
sealedObj.newProp = "add"; // This fails.

const frozenObj = { prop: 1 };
Object.freeze(frozenObj);
frozenObj.prop = 2; // This fails.
```

---

## ? Q #3 ( 🟡 Intermediate )

Tags: 🚀 MAANG-level , Objects , Cloning , Shallow Copy , Deep Copy

🔥 EXTREMELY IMPORTANT

🧠 Question:

**What is the difference between a shallow copy and a deep copy of an object? Explain the risk of using a shallow copy.**

✅ Answer:

- **Shallow Copy** copies only the top-level properties of an object. If a property's value is a reference to another object (like a nested object or array), only the reference is copied, not the object itself. You can create one using the spread syntax `{ ...obj }` or `Object.assign()`.
- **Deep Copy** recursively copies all properties at every level, creating a completely independent clone of the original object and all its nested objects. A common method is `JSON.parse(JSON.stringify(obj))`, though it has limitations (e.g., it loses functions).

The main risk of a shallow copy is that modifying a nested object in the copy will also mutate the nested object in the original, as both share the same reference.

```
javascriptconst original = { name: 'A', nested: { value: 1 } };
const copy = { ...original }; // Shallow copy

copy.nested.value = "changed";

// The original object is also affected because the nested object was shared.
console.log(original.nested.value); // "changed"
```

## ? Q #4 ( 🔴 Advanced )

Tags: 🚀 MAANG-level , Objects , Property Descriptors

🧠 Question:

**What are property descriptors in JavaScript? Describe the roles of `writable`, `enumerable`, and `configurable`.**

✅ Answer:

Property descriptors are metadata that define how an object property behaves. Every property has an associated descriptor with these key attributes:

- **writable** : If `true` , the property's value can be changed. If `false` , it's read-only.
- **enumerable** : If `true` , the property will appear in `for...in` loops, `Object.keys()` , and `JSON.stringify()` . If `false` , it is hidden from these operations but can still be accessed directly.
- **configurable** : If `true` , you can delete the property and change its descriptor attributes (except switching `configurable` itself to `true` again). If `false` , the property is locked down, and you cannot delete it or modify its descriptor (though you can still change its value if `writable` is `true` ).

## ? Q #5 ( Advanced )

Tags:  MAANG-level , Objects , Symbols

 Question:

How can you use a `Symbol` as an object property key, and what is the primary advantage of doing so over using a string?

 Answer:

You use a `Symbol` as a property key by creating a symbol with `Symbol()` and using it inside square brackets `[]` when defining the object property.

The primary advantage is that symbols are unique and immutable. This prevents accidental property name collisions. Even if another developer adds a property with the same string description, it won't overwrite your symbol-keyed property because each symbol is guaranteed to be unique. This is useful for adding metadata or "hidden" properties to objects without interfering with other keys.

```
javascriptconst id = Symbol("id"); // The description "id" is just for debugging.
const user = {
  name: "Bob",
  [id]: 123 // Using the symbol as a key
};

console.log(user[id]); // 123
console.log(user['id']); // undefined - No collision with a string key 'id'
console.log(Object.keys(user)); // ['name'] - Symbols are non-enumerable by default.
```

## ? Q #6 ( 🟡 Intermediate )

Tags: 🚀 MAANG-level , Execution Context , Call Stack , Event Loop

🔥 EXTREMELY IMPORTANT

🧠 Question:

**Briefly explain the roles of the Execution Context and the Call Stack in how JavaScript runs code.**

✅ Answer:

- **Execution Context:** This is the environment where JavaScript code is evaluated and executed. It contains the currently running code, variables, functions, and scope chain. A Global Execution Context (GEC) is created by default, and a new Function Execution Context (FEC) is created for every function call.
- **Call Stack:** This is a LIFO (Last-In, First-Out) data structure that tracks which function is currently being run. When a function is called, its execution context is pushed onto the stack. When the function finishes, its context is popped off the stack. Synchronous code is managed entirely by the Call Stack.

---

## ? Q #7 ( 🔴 Advanced )

Tags: 🚀 MAANG-level , Event Loop , Microtasks , Macrotasks

🔥 EXTREMELY IMPORTANT

🧠 Question:

**What is the difference between the microtask queue and the macrotask queue in the JavaScript Event Loop?**

✅ Answer:

The key difference is priority. The Event Loop prioritizes the microtask queue over the macrotask queue.

- **Microtasks** are high-priority tasks that must be completed before the browser can continue with other work, like rendering. Examples include `Promise.then()` , `.catch()` , `.finally()` , and `queueMicrotask()` . The event loop will execute *all* tasks in the microtask queue until it's empty before moving on.

- **Macrotasks** (or tasks) are lower-priority tasks that are handled one at a time per event loop cycle. Examples include `setTimeout`, `setInterval`, I/O operations, and UI events.

After the call stack is empty, the event loop processes all microtasks, then processes just *one* macrotask, then goes back to check for more microtasks.

## ? Q #8 ( Advanced )

Tags:  MAANG-level, Event Loop, Microtasks, Macrotasks, Promises

 EXTREMELY IMPORTANT

 Question:

**Predict the output of the following code and explain the execution order based on the Event Loop model.**

```
javascriptconsole.log("Start");
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

 Answer:

The output will be:

Start

End

Promise

Timeout

**Explanation:**

1. `console.log("Start")` is synchronous and runs immediately. It's pushed to the call stack and executed.
2. `setTimeout(...)` is a macrotask. Its callback is sent to the Web API and moved to the macrotask queue after 0ms.
3. `Promise.resolve().then(...)` is a microtask. Its `.then()` callback is placed in the microtask queue.
4. `console.log("End")` is synchronous and runs immediately.

5. The call stack is now empty. The Event Loop checks the microtask queue first. It finds the "Promise" callback, pushes it to the call stack, and executes it.
  6. The microtask queue is now empty. The Event Loop checks the macrotask queue, finds the "Timeout" callback, and executes it.
- 

### ? Q #9 ( Advanced )

Tags:  MAANG-level , Event Loop , Microtasks

 Question:


**What is "starvation" in the context of the Event Loop, and how does it relate to microtasks?**

 Answer:

Starvation occurs when the macrotask queue is perpetually blocked because the microtask queue is never empty. Since the Event Loop must clear the *entire* microtask queue before it can process a single macrotask, a continuous stream of new microtasks (e.g., in a recursive `Promise.then` loop) can prevent any macrotasks like `setTimeout` or UI rendering from ever running. This effectively "starves" the macrotask queue.

---

### ? Q #10 ( Intermediate )

Tags:  General Interview , Asynchronous , Callbacks , Promises

 Question:

**What is "Callback Hell," and how do Promises offer a solution?**

 Answer:

"Callback Hell" (also known as the "Pyramid of Doom") describes a situation where multiple nested callbacks are chained together to handle sequential asynchronous operations. This creates deeply indented, unreadable, and hard-to-maintain code.


Promises solve this by allowing you to chain asynchronous operations in a flat, linear way using `.then()`. Each `.then()` returns a new promise, allowing you to

sequence actions one after another without nesting, which makes the code much cleaner and easier to reason about.

```
javascript // Callback Hell
getData(1, () => {
  getData(2, () => {
    getData(3);
  });
});

// Promise Solution
getData(1)
  .then(() => getData(2))
  .then(() => getData(3));
```

## ? Q #11 ( Advanced )

Tags:  MAANG-level , Asynchronous , Promises , Promise Chaining

 EXTREMELY IMPORTANT

 Question:

In a Promise chain, what is the consequence of forgetting to `return` a value or another Promise from a `.then()` block?

 Answer:

If you forget to `return` anything from a `.then()` handler, it implicitly returns `undefined`. The next `.then()` in the chain will receive `undefined` as its input, which can break the flow of data.

More critically, if the handler starts an asynchronous operation (like another Promise) but doesn't `return` it, the chain will **not** wait for that operation to complete. It will move on to the next `.then()` immediately, leading to race conditions and incorrect sequencing.

```
javascriptPromise.resolve(2)
  .then((num) => {
    num = num * 3;
    // Forgot to return num!
  })
  .then((result) => {
    console.log(result); // This will print 'undefined'
  });
```



## ? Q #12 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Asynchronous , async/await

🧠 Question:

What does the `async` keyword do to a function, and what is the role of the `await` keyword within it?

✅ Answer:

The `async` keyword automatically transforms a function to ensure it always returns a `Promise`. If the function explicitly returns a value, that value will be wrapped in a resolved `Promise`.

The `await` keyword can only be used inside an `async` function. It pauses the execution of the function at that line, waits for the `Promise` it is "awaiting" to settle (either resolve or reject), and then resumes execution. While paused, it doesn't block the main thread, allowing other code to run.

## ? Q #13 ( 🔴 Advanced )

Tags: 🚀 MAANG-level , Asynchronous , async/await , Promises

🧠 Question:

Explain how you would run multiple asynchronous tasks sequentially versus in parallel using `async/await`.

✅ Answer:

- **Sequential Execution:** To run tasks one after another, you simply `await` each promise in sequence. The second task will not start until the first one has completed. The total time is the sum of all task durations.
- **Parallel Execution:** To run tasks concurrently, you initiate all the asynchronous operations without awaiting them immediately. This starts them all at roughly the same time. Then, you use `Promise.all()` to wait for all of them to complete. The total time is determined by the duration of the longest task.

```
javascript // Sequential Execution
async function sequential() {
  let result1 = await doJob(1); // Waits for this to finish
  let result2 = await doJob(2); // Then starts and waits for this
  return result1 + result2;
}
```

```

}

// Parallel Execution
async function parallel() {
  let promise1 = doJob(1); // Starts task 1
  let promise2 = doJob(2); // Starts task 2// Waits for both to complete concurrently
  let [result1, result2] = await Promise.all([promise1, promise2]);
  return result1 + result2;
}

```

## ? Q #14 ( Basic )

Tags:  General Interview , Arrays , Array Creation

 Question:

What is the difference in behavior between `new Array(7)` and ``?

 Answer:

- `new Array(7)` creates a **sparse array** with a `length` of 7 but no actual elements in it. It contains 7 "empty slots" or "holes".
- `` (or `Array.of(7)`) creates an array with a `length` of 1, containing a single element: the number `7`.

```

javascriptconst sparseArray = new Array(7);
console.log(sparseArray.length); // 7
console.log(sparseArray); // undefined (but it's a hole)


```

```

const literalArray = ;
console.log(literalArray.length); // 1
console.log(literalArray); // 7

```

## ? Q #15 ( Intermediate )

Tags:  General Interview , Arrays , Iteration

 EXTREMELY IMPORTANT

 Question:

What are the key differences between the array methods `.forEach()` , `.map()` , and `.reduce()` ?

 Answer:

The main differences are their return value and primary use case:

- `.forEach()` : Executes a function once for each array element. It does **not** create a new array and always returns `undefined` . Its purpose is to cause a side effect for each element (e.g., logging it to the console).
- `.map()` : Executes a function on every element and **returns a new array** containing the results. The new array has the same length as the original. It's used for transforming data.
- `.reduce()` : Executes a reducer function on each element to produce a **single, accumulated output value**. It can return anything (a number, string, object, etc.). It's used for aggregating data from an array into one value.

## ? Q #16 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Arrays , Destructuring , Spread Operator

🧠 Question:

How do the spread operator ( `...` ) and array destructuring help you work with arrays? Provide an example of each.

✅ Answer:

Both are non-mutative ways to handle array elements:

- **Spread Operator ( `...` )**: "Unpacks" the elements of an array into another array or a function call. It's commonly used to create a shallow copy of an array or to combine arrays.
- **Destructuring**: "Unpacks" elements from an array into distinct variables. It provides a concise syntax for accessing array elements by their position.

```
javascript // Spread Operator for copying
const original = [1, 2, 3];
const copy = [...original, 4]; // [1, 2, 3, 4]// Destructuring for variable assignment
const toys = ["ball", "doll", "car"];
const [toy1, toy2] = toys;
console.log(toy1); // 'ball'
console.log(toy2); // 'doll'
```

## ? Q #17 ( 🟡 Intermediate )

Tags: 🚀 MAANG-level , Arrays , Array-likes , Iterables

### Question:

What is an "array-like" object, and what is the modern way to convert it into a true array?

### Answer:

An "array-like" object is an object that has a `length` property and indexed elements (e.g., `0`, `1`, `2`), but does not have built-in array methods like `.map()` or `.forEach()`. A common example is the `NodeList` returned by `document.querySelectorAll()`.

The modern and most robust way to convert an array-like object into a true array is by using `Array.from()`.

```
javascript // Assuming `elements` is a NodeList from document.querySelectorAll('div')// const elements = document.querySelectorAll('div');// This is an array-like object, not a true array.// elements.map(...) would fail.// Convert it to a true array// const trueArray = Array.from(elements);// trueArray.map(...); // This now works!
```

## ? Q #18 ( Advanced )

Tags:  MAANG-level , Arrays , Sparse Arrays

### Question:

What is a sparse array? Explain the difference between a "hole" in an array and a value of `undefined`, and how array methods like `.map()` treat them.

### Answer:

A sparse array is an array that has gaps or "holes" where no element has been defined, often created using `new Array(size)` or with trailing commas in an array literal.

- A **hole** is a truly empty slot. The property does not exist on the object. You can test for it with `index in array`, which would return `false`.
- An `undefined` value is a consciously set value. The property exists, and its value is `undefined`. `index in array` would return `true`.

Most iterative array methods, like `.map()` and `.filter()`, **skip over holes** completely but will execute the callback for elements with an `undefined` value.

```
javascriptconst withHole = [1, , 3]; // Hole at index 1
const withUndefined = [1, undefined, 3];

// .map() skips the hole but processes 'undefined'
const mappedHole = withHole.map(x => x * 2); // [2, <1 empty item>, 6]
const mappedUndefined = withUndefined.map(x => x * 2); // [2, NaN, 6]
```

---

## ? Q #19 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Data Structures , Map , Object

🔥 EXTREMELY IMPORTANT

🧠 Question:

When should you use a `Map` instead of an `Object` in JavaScript?

✅ Answer:

A `Map` should be used instead of an `Object` when you need more flexibility with key types or require features that objects lack. The main advantages of a `Map` are:

1. **Any Key Type:** `Map` keys can be any data type (including objects, functions, or primitives), whereas `Object` keys are limited to strings and symbols.
2. **Guaranteed Order:** `Map` elements are iterated in insertion order. `Object` property order is not guaranteed (though it has become more predictable in modern JS).
3. **Size Property:** `Map` has a `.size` property to easily get the number of entries, which is more direct than `Object.keys(obj).length`.
4. **Performance:** Maps are often more performant for frequent additions and removals of key-value pairs.
5. **No Prototype Clashes:** A `Map` is a clean dictionary and doesn't have prototype properties that could clash with your keys (like 'toString').

---

## ? Q #20 ( 🟢 Basic )

Tags: 🌱 General Interview , Data Structures , Set

🧠 Question:

What is a `Set` in JavaScript, and what is its main characteristic?

✅ Answer:

A `Set` is a collection of values, similar to an array, but its main characteristic is that it only allows **unique** values. If you try to add a value that already exists in the `Set`,

the duplicate will be ignored. It's useful for quickly filtering out duplicates from a list or checking for the presence of an item.

```
javascriptlet mySet = new Set();  
mySet.add(1);  
mySet.add(2);  
mySet.add(1); // This is ignored
```

```
console.log(mySet.size); // Outputs: 2  
console.log(mySet.has(1)); // Outputs: true
```

---

## ? Q #21 ( Advanced )

Tags:  MAANG-level , Data Structures , WeakMap , Garbage Collection

 EXTREMELY IMPORTANT

 Question:

What is a **WeakMap** , and how does its "weak" reference to keys affect memory management and garbage collection?


 Answer:

A **WeakMap** is a specialized **Map** where keys must be objects and are held "weakly." This means the **WeakMap** does not prevent its keys from being garbage collected. If an object used as a key is no longer referenced anywhere else in the program, the garbage collector can remove it from memory, and its corresponding entry in the **WeakMap** will also be automatically removed.

This behavior is its main advantage: it prevents memory leaks by allowing temporary metadata associated with an object to be cleaned up automatically when the object itself is no longer needed.

---

## ? Q #22 ( Advanced )

Tags:  MAANG-level , Data Structures , WeakMap , WeakSet

 Question:

Why can't you iterate over a **WeakMap** or **WeakSet** , or get their size?

 Answer:

You cannot iterate over weak collections or get their `.size` because their contents are unpredictable and can change at any moment due to the behavior of the garbage collector. Since entries can be removed automatically and non-deterministically whenever a key/value object is no longer referenced, providing a list of keys or a size count would be unreliable and could be outdated as soon as it's returned. They are designed to be non-enumerable for this reason.

---

## ? Q #23 ( ● Intermediate )

Tags: ✳ General Interview , Strings , Immutability

 Question:

**Strings are described as immutable in JavaScript. What does this mean, and what happens when you call a method like `.toUpperCase()` on a string?**

 Answer:

String immutability means that once a string is created, its contents cannot be changed. When you call a method like `.toUpperCase()` or `.slice()` on a string, you are not modifying the original string. Instead, the method returns a **new string** with the changes applied. The original string remains untouched. This is also why method chaining works on strings.

```
javascriptlet greeting = "hello";  
let loudGreeting = greeting.toUpperCase();
```

```
console.log(loudGreeting); // "HELLO"  
console.log(greeting);    // "hello" (The original string is unchanged)
```

---

## ? Q #24 ( ● Basic )

Tags: ✳ General Interview , Strings , Template Literals

 Question:

**What are template literals, and how do they differ from traditional string concatenation?**

 Answer:

Template literals are strings created with backticks ( ``` ) instead of single or double quotes. They offer two main advantages over traditional string concatenation with

the `+` operator:

1. **Easier Interpolation:** You can embed variables and expressions directly into the string using the `${expression}` syntax, which is much cleaner and more readable than breaking the string to add variables.
2. **Multiline Strings:** They allow you to create strings that span multiple lines without needing to use newline characters (`\n`).

```
javascriptconst name = "Alice";  
// Traditional concatenation  
const oldGreeting = "Hello, " + name + "!";  
  
// Template literal with interpolation  
const newGreeting = `Hello, ${name}!`;
```

---

## ? Q #25 ( ● Advanced )

Tags: 🚀 MAANG-level , Strings , Template Literals

🧠 Question:

What is a "tagged template," and what is its primary use case? Provide a simple example.

✅ Answer:

A tagged template is an advanced form of a template literal where a function (the "tag") is placed before the literal. This function receives the static parts of the string and the interpolated values as separate arguments, allowing you to parse and process the template literal in a custom way before the final string is constructed.

A primary use case is for custom formatting, sanitizing input to prevent injection attacks (like in SQL queries or HTML), or localization.

```
javascript // The 'tag' function receives the static strings and interpolated values  
function bold(strings, ...values) {  
  // Here, strings is ['This is ', '!'], and values is ['important']  
  return strings[0] + values.toUpperCase() + strings[1];  
}  
  
let adjective = "important";  
let message = bold`This is ${adjective}!`;  
  
console.log(message); // "This is IMPORTANT!"
```

---



## ? Q #26 ( Basic )

Tags:  General Interview ,  Objects ,  Object Creation

 Question:

**What are computed properties in an object literal, and what is their syntax?**



 Answer:

Computed properties allow you to use an expression or a variable as a property key when creating an object. The expression is evaluated, and its result is used as the property name. You define them by wrapping the expression in square brackets `[]` inside the object literal.

```
javascriptconst prefix = "user_";
const user = {
  [prefix + "id"]: 123,
  [prefix + "name"]: "Alice"
};

console.log(user.user_id); // 123
console.log(user); // { user_id: 123, user_name: "Alice" }
```

## ? Q #27 ( Intermediate )

Tags:  General Interview ,  Objects ,  Deep Copy

 Question:

**Your notes mention using `JSON.parse(JSON.stringify(obj))` for a deep copy. What is a key limitation of this method?**

 Answer:

The main limitation of the `JSON.parse(JSON.stringify(obj))` method is that it cannot handle all JavaScript data types. It will lose any non-JSON-safe values, such as functions, `undefined`, and `Symbol` s, during the stringification process. This makes it a simple but potentially unsafe method for deep copying complex objects.

```
javascriptconst original = {
  name: "Alice",
  id: Symbol("id"),
  greet: () => "hello",
  timestamp: undefined
};

const deepCopy = JSON.parse(JSON.stringify(original));
```

```
// The function, symbol, and undefined are lost.  
console.log(deepCopy); // { name: "Alice" }
```

## ? Q #28 ( Basic )

Tags:  General Interview , Objects , Object Literal Shorthand

 Question:

**What is object literal shorthand syntax and how does it reduce repetition?**

 Answer:

Object literal shorthand is a concise syntax used when you want to create an object from variables whose names are the same as the desired property keys. Instead of writing `{ id: id, name: name }`, you can simply write `{ id, name }`. JavaScript automatically maps the variable name to the key and assigns its value.

```
javascriptconst id = 1;  
const name = "Item";  
  
// Without shorthand  
const productOld = { id: id, name: name };  
  
// With shorthand  
const productNew = { id, name };  
  
console.log(productNew); // { id: 1, name: "Item" }
```

## ? Q #29 ( Basic )

Tags:  General Interview , Execution Context

 Question:

**What are the two main types of execution contexts that your notes describe?**

 Answer:

The notes describe two types of execution contexts:

1. **Global Execution Context (GEC):** This is the default context created when a JavaScript program starts. It handles all the code that is not inside any function.

2. **Function Execution Context (FEC) or "Specific EC"**: A new, separate execution context is created every time a function is called. It handles the code inside that specific function.
- 

## ? Q #30 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Execution Context

🧠 Question:

**Within an execution context, what are the roles of the "Memory" and "Code" blocks as mentioned in your notes?**

✅ Answer:

According to the notes, an execution context consists of two main components:

1. **Memory Block (Variable Environment)**: In this phase, the JavaScript engine scans the code for variable and function declarations. It allocates memory, storing variable declarations with a placeholder value of `undefined` and function declarations with their entire function body.
  2. **Code Block (Thread of Execution)**: In this phase, the engine executes the code line by line, assigning values to variables and invoking functions as it goes.
- 

## ? Q #31 ( 🟢 Basic )

Tags: 🌱 General Interview , Asynchronous , Callbacks

🧠 Question:

**Based on your notes, how would you define a "callback function"?**

✅ Answer:

A callback function is simply a function that is passed as an argument (a parameter) into another function. The outer function can then call this "callback" function at a later time, usually after some task has been completed. This is a fundamental pattern for handling asynchronous operations in JavaScript.

```
javascriptfunction sum(a, b) {  
  console.log(a + b); // The callback's logic  
}
```

```
// `sum` is passed as a callback to `callerFunction`  
function callerFunction(first, second, callback) {  
  callback(first, second);  
}
```

```
callerFunction(2, 4, sum); // Outputs: 6
```

## ? Q #32 ( Advanced )

Tags:  MAANG-level , Asynchronous , Promises

 EXTREMELY IMPORTANT

 Question:

In a Promise chain, distinguish between returning a value and performing a side effect inside a `.then()` handler.

 Answer:

- **Returning a Value:** This is the correct way to pass data down a Promise chain. When you `return` a value or another Promise from a `.then()`, that result becomes the input for the next `.then()` in the chain. This creates a predictable and testable flow.
- **Performing a Side Effect:** This is when you perform an action (like `console.log()` or updating an external variable) inside a `.then()` but do not `return` a value. The `.then()` implicitly returns `undefined`, which can break the chain or lead to unexpected behavior if subsequent links expect data.

```
javascript // Good practice: Returning a value  
Promise.resolve(5)  
  .then(num => num * 2) // Returns 10  
  .then(result => console.log(result)); // Logs 10// Side effect: breaks the data flow  
Promise.resolve(5)  
  .then(num => { console.log(num * 2); }) // Logs 10, but returns undefined  
  .then(result => console.log(result)); // Logs undefined
```

## ? Q #33 ( Intermediate )

Tags:  General Interview , Asynchronous , async/await

 Question:

What happens if you `await` a value that is not a Promise?

### ✓ Answer:

If you use `await` on a non-Promise value (e.g., a number, string, or object), JavaScript will wrap that value in an immediately resolved Promise and then "await" it. The expression will then resolve to that value, and the function will continue execution on the next line. It effectively treats the value as if it were the result of a synchronous operation.

```
javascriptasync function testAwait() {  
  console.log("Start");  
  const value = await 42; // Not a Promise  
  console.log(value); // This will still work and log 42  
  console.log("End");  
}  
testAwait(); // Logs "Start", then 42, then "End" in order
```

## ? Q #34 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Arrays

### 🧠 Question:

In what sense is a JavaScript array a "specialized object"?

### ✓ Answer:

An array is a specialized object because, at its core, it is an object that uses numeric indices as keys (e.g., `'0'`, `'1'`, `'2'`). However, it's specialized with extra features that plain objects don't have, such as:

1. A `length` property that automatically updates when you add or remove elements.
2. A prototype ( `Array.prototype` ) that provides a rich set of built-in methods for traversal and mutation (e.g., `.map()`, `.push()`, `.filter()` ).

## ? Q #35 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Arrays , Array-likes

### 🧠 Question:

Your notes list three ways to convert an array-like object to a true array. What are they?

### ✓ Answer:

The notes describe three common methods for converting an array-like object (like a `NodeList` or `arguments` object) into a true array so that array methods can be used on it:

1. `Array.from(arrayLike)` : The most modern and recommended approach.
  2. **Spread Syntax** `[...arrayLike]` : A concise and popular ES6 alternative.
  3. `Array.prototype.slice.call(arrayLike)` : The traditional, pre-ES6 method.
- 

### ? Q #36 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Arrays , Spread Operator , Rest Parameters

#### 🧠 Question:

Your notes cover the spread operator. How does the rest parameter syntax differ from the spread operator, even though they both use `...` ?

### ✓ Answer:

Although they use the same `...` syntax, their purpose is opposite:

- **Spread Operator:** *Unpacks* elements from an iterable (like an array) into individual elements. It's used in places like array literals ( `[...arr]` ) or function calls ( `myFunc(...arr)` ).
- **Rest Parameter:** *Packs* remaining function arguments into a single array. It must be the last parameter in a function definition and gathers all "rest" of the arguments.

```
javascript // Spread: Unpacks the array into arguments
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // same as Math.max(1, 2, 3)
// Rest: Packs arguments into an array
function sum(...args) { // args becomes [1, 2, 3, 4]
  return args.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

---

### ? Q #37 ( 🟡 Intermediate )

Tags: 🚀 MAANG-level , Data Structures , Map , Object

## 🔥 EXTREMELY IMPORTANT

### 🧠 Question:

What is the key difference regarding key types between a `Map` and a traditional `Object` ?

### ✅ Answer:

The fundamental difference is that `Map` keys can be of **any data type**, including objects, functions, and primitives (like numbers or booleans). In contrast, a traditional `Object` can only have keys that are **strings** or **symbols**. If you use a non-string value (like a number) as an object key, JavaScript implicitly converts it to a string.

```
javascriptlet myMap = new Map();
let myObj = {};
```

```
const keyObject = { id: 1 };
const keyFunc = () => {};
```

```
// Map can use any type as a key
myMap.set(keyObject, "Value for object key");
myMap.set(keyFunc, "Value for function key");
```

```
// Object converts keys to strings
myObj[keyObject] = "Value for object key"; // key becomes "[object Object]"
console.log(myObj); // { '[object Object]': 'Value for object key' }
```

## ? Q #38 ( 🟡 Advanced )

Tags: 🚀 MAANG-level , Data Structures , WeakMap , Use Cases

### 🧠 Question:

Based on your notes, what is a practical, real-world use case for a `WeakMap` ?

### ✅ Answer:

A practical use case for a `WeakMap` is to associate temporary metadata or cached data with an object without causing memory leaks. For example, in a web application, you could store extra, non-essential information about a DOM element. If that element is later removed from the page, the `WeakMap`'s weak reference allows the element and its associated metadata to be garbage collected automatically, preventing memory from being wasted.

## ? Q #39 ( Advanced )

Tags:  MAANG-level , Data Structures , WeakSet , Use Cases

 Question:

What is a practical use case for a **WeakSet** , as described in your notes?

 Answer:

A **WeakSet** is useful for tracking a collection of unique objects that might disappear over time. A good example is keeping a list of objects that have already been processed or "seen" in an application. If an object from that list is no longer needed and gets garbage collected, it is automatically removed from the **WeakSet** , ensuring the tracking list doesn't hold onto dead references and cause memory leaks.

---

## ? Q #40 ( Intermediate )

Tags:  General Interview , Data Structures , WeakMap , WeakSet

 Question:

What happens if you try to add a primitive value like a number or a string to a **WeakSet** or use one as a key in a **WeakMap** ?

 Answer:

It will throw a **TypeError** . Both **WeakMap** and **WeakSet** are designed to hold objects only. **WeakMap** keys must be objects, and **WeakSet** values must be objects. Attempting to use a primitive value (like a string, number, or boolean) will result in an error because their lifecycle is not managed by the garbage collector in the same way objects are, which defeats the purpose of "weak" references.

```
javascriptconst myWeakSet = new WeakSet();  
// myWeakSet.add("hello"); // Throws TypeError: Invalid value used in weak set
```

```
const myWeakMap = new WeakMap();  
// myWeakMap.set("key1", "value"); // Throws TypeError: Invalid value used as weak map key
```

---

## ? Q #41 ( Basic )

Tags:  General Interview , Data Structures , Map , Object



### Question:

How does getting the number of entries in a `Map` compare to getting the number of properties in an `Object` ?

### Answer:

Getting the size of a `Map` is more direct. A `Map` has a built-in `.size` property that instantly returns the number of key-value pairs. For an `Object`, there is no direct `.size` or `.length` property. You must first get an array of its keys, values, or entries and then get the length of that array, for example, by using `Object.keys(obj).length`.

```
javascriptlet myMap = new Map([['a', 1], ['b', 2]]);
console.log(myMap.size); // 2
```

```
let myObj = { a: 1, b: 2 };
console.log(Object.keys(myObj).length); // 2
```

## ? Q #42 ( Basic )

Tags:  General Interview , Data Structures , Arrays , Set

### Question:

How can you use a `Set` to easily remove duplicate values from an array?

### Answer:

Because a `Set` only stores unique values, you can remove duplicates from an array by converting the array into a `Set` and then converting it back into an array. This is a very concise way to get a new array with only unique elements.

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

// Convert array to Set to remove duplicates, then spread back into a new array
const uniqueNumbers = [...new Set(numbersWithDuplicates)];

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

## ? Q #43 ( Basic )

Tags:  General Interview , Numbers , Numeric Separators

### Question:

**What is the purpose of numeric separators ( `_` ) in JavaScript, and how does the engine treat them?**

✅ **Answer:**

The purpose of numeric separators ( `_` ) is purely for readability, to make large numbers easier to parse visually by grouping digits. The JavaScript engine completely ignores these underscores when it reads the number. It's syntactic sugar that has no effect on the actual value of the number.

```
javascriptconst oneMillion = 1_000_000;  
const oneBillion = 1_000_000_000;  
  
console.log(oneMillion); // 1000000  
console.log(oneBillion === 1000000000); // true
```

---

## ? Q #44 ( 🟡 Intermediate )

**Tags:** 🌱 General Interview , Strings , Method Chaining

🧠 **Question:**

**When would method chaining on a string value fail?**

✅ **Answer:**

Method chaining on a string fails if any method in the chain returns a value that is not a string (or does not have string methods on its prototype). For example, a method like `.split()` returns an array, and `.indexOf()` returns a number. You cannot call another string method like `.toUpperCase()` on an array or a number, so the chain would break and throw an error.

```
javascriptconst text = "hello world";  
  
// This chain breaks because .split(' ') returns an array.// text.split(' ').toUpperCase(); // Throws TypeError:  
text.split(...).toUpperCase is not a function// This chain breaks because .indexOf('w') returns a number.//  
text.indexOf('w').slice(0); // Throws TypeError: text.indexOf(...).slice is not a function
```

---

## ? Q #45 ( 🔴 Advanced )

**Tags:** 🚀 MAANG-level , Objects , Object Immutability

🧠 **Question:**

Your notes state that `Object.freeze()` is shallow. What does this mean for an object with nested objects, and how would you demonstrate it?

✓ Answer:

The "shallow" nature of `Object.freeze()` means it only makes the top-level properties of an object immutable. If one of those properties is a reference to another object (like a nested object or array), the properties of that *nested* object remain mutable. You can still change, add, or delete properties within the nested object.

```
javascriptconst user = {
  name: "Alice",
  details: { age: 30 }
};

Object.freeze(user);

// Attempting to change a top-level property fails.
user.name = "Bob"; // Fails silently.// But changing a property of the nested object succeeds.
user.details.age = 31;

console.log(user.name); // "Alice"
console.log(user.details.age); // 31
```

## ? Q #46 ( 🟡 Intermediate )

Tags: 🌱 General Interview , Asynchronous , async/await , Promises

🧠 Question:

How does `async/await` improve upon the readability of `.then()` promise chains?

✓ Answer:

`async/await` significantly improves readability by allowing you to write asynchronous code that looks and behaves like synchronous code. Instead of chaining `.then()` callbacks and passing data through function arguments, you can `await` a promise and assign its resolved value directly to a variable. This avoids the nesting of callbacks and creates a flat, linear, top-to-bottom code structure that is easier to read and debug.

## ? Q #47 ( 🟢 Basic )

Tags: 🌱 General Interview , Data Structures , Map

### Question:

Demonstrate the three basic operations for managing data in a `Map` : adding, retrieving, and checking for a key.


### Answer:

The three basic operations for a `Map` are:

- `set(key, value)` : Adds or updates a key-value pair.
- `get(key)` : Retrieves the value associated with a key.
- `has(key)` : Checks if a key exists, returning `true` or `false` .

```
javascriptlet myMap = new Map();  
  
// 1. Add data  
myMap.set("name", "Alice");  
myMap.set("age", 30);  
  
// 2. Retrieve data  
console.log(myMap.get("name")); // "Alice"  
// 3. Check for a key's existence  
console.log(myMap.has("age")); // true  
console.log(myMap.has("city")); // false
```

## ? Q #48 ( Basic )

Tags:  General Interview , Data Structures , Set

### Question:

Demonstrate the three basic operations for managing data in a `Set` : adding, checking for, and deleting a value.

### Answer:

The three basic operations for a `Set` are:

- `add(value)` : Adds a value to the set. Duplicates are ignored.
- `has(value)` : Checks if a value exists, returning `true` or `false` .
- `delete(value)` : Removes a value from the set.

```
javascriptlet mySet = new Set();  
  
// 1. Add values  
mySet.add(10);  
mySet.add(20);
```

```
mySet.add(10); // Ignored// 2. Check for a value
console.log(mySet.has(20)); // true
console.log(mySet.has(30)); // false// 3. Delete a value
mySet.delete(10);
console.log(mySet.has(10)); // false
console.log(mySet.size); // 1
```

---

## ? Q #49 ( Intermediate )

Tags:  General Interview ,  Objects ,  Data Structures

 Question:

**According to your notes, what happens if you use a number as a key in a standard object literal?**

 Answer:



If you use a number as a key in a standard object literal, JavaScript will implicitly convert that number into a string. All object keys are either strings or symbols, so a numeric key like `5` is stored as the string `"5"`.

```
javascriptconst obj = {
  5: "abc"
};

// The key is stored as a string.
console.log(Object.keys(obj)[0]); // "5" (a string)// You can access it with either the number or the string.
console.log(obj[5]); // "abc"
console.log(obj["5"]); // "abc"
```

---

## ? Q #50 ( Intermediate )

Tags:  MAANG-level ,  Event Loop

 Question:

**What does it mean to "block the event loop," and how can this happen with synchronous code?**

 Answer:

Blocking the event loop means executing a long-running synchronous task on the main thread. Because JavaScript is single-threaded, while this task is running, the browser cannot do anything else—it cannot handle user input (clicks), update the UI, or process any asynchronous tasks from the macrotask or microtask queues.

This leads to a frozen or unresponsive webpage. A common example is a complex, long-running `for` loop.

---

## ? Q #51 ( Advanced )

Tags:  MAANG-level , `Event Loop`

 Question:

Between `Promise.resolve().then(callback)` and `queueMicrotask(callback)` , which is the more direct way to schedule a microtask, and why might you choose one over the other?

 Answer:

`queueMicrotask(callback)` is the more direct and modern way to schedule a microtask. It explicitly tells the engine to place the callback in the microtask queue without any extra overhead.

`Promise.resolve().then(callback)` also schedules a microtask, but it involves the creation and resolution of a Promise, which adds a small amount of overhead. You might use the Promise-based approach if you are already working within a Promise-based API or need to maintain compatibility with older environments that don't support `queueMicrotask` . However, for simply queueing a function to run as a microtask, `queueMicrotask` is the cleaner and more performant choice.