JavaScript-Marathon Interview Questions Series

Q1) What is "undefined" in JavaScript?

`undefined` is a primitive value in JavaScript. It represents the **absence of a value** or when a variable has been **declared but not yet assigned** a value.

Example:

```
let x;
console.log(x); // Output: undefined
```

Here, `x` is declared but not initialized, so JavaScript assigns it `undefined` by default.

Also:

- Functions that **don't return anything** explicitly return `undefined`.
- Accessing an **object property that doesn't exist** gives `undefined`.

Q2) What will be the output of `undefined == null` and `undefined === null` ? Why?

```
undefined == null    // true
undefined === null   // false
```

Why?

- `==` **(loose equality)** checks for value **after type coercion**. JavaScript considers `undefined` and `null` to be **loosely equal**.
- `===` **(strict equality)** checks for value **and type**. Since:
  - `typeof undefined` is `"undefined"`
  - `typeof null` is `"object"`

  They are **not the same type**, so the result is `false`.

Q3) Can you explicitly assign "undefined" to a variable? (`let i = undefined`)

✅ **Yes, you can!**

```
let i = undefined;
console.log(i); // Output: undefined
```

But note: While it works, it's **usually better to use** `null` to indicate "intentional absence of value" — and let JavaScript assign `undefined` by default.

Q4) What is hoisting in JavaScript?

Hoisting is JavaScript's behavior of **moving declarations to the top of their scope (memory phase)** before code execution.

- Only **declarations** are hoisted — **not initializations**.
- Works differently for `var`, `let`, `const`, and functions.

Example with `var`:

```
console.log(x); // undefined (not error)
var x = 5;
```

JavaScript interprets this as:

```
var x;
console.log(x); // undefined
x = 5;
```

Example with `let` or `const`:

```
console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 10;
```

Because `let` and `const` are hoisted to the top **but stay in a temporal dead zone** until they are declared. This means that although the variable exists, it cannot be accessed until the line where it's declared and initialized.

Q5) How does block scope work?

Block scope means variables declared inside a block (`{}`) with `let` or `const` **exist only within that block**.

- `var` is **function-scoped**, not block-scoped.
- `let` and `const` are **block-scoped**.

Example:

```
{
  let a = 10;
  const b = 20;
  var c = 30;
}
console.log(c); // ✅ 30
console.log(a); // ❌ ReferenceError
console.log(b); // ❌ ReferenceError
```

Q6) What is the scope of a variable?

The **scope** of a variable is the part of the code **where the variable is accessible**.

There are several types of scope in JavaScript:

1. **Global Scope** – declared outside any function/block.
2. **Function Scope** – declared with `var` inside a function.
3. **Block Scope** – declared with `let` or `const` inside `{}`.
4. **Lexical Scope (Closure)** – inner functions have access to outer variables.

Example:

```
let globalVar = "I'm global";

function test() {
   let localVar = "I'm local";
   console.log(globalVar); // ✅ Accessible
   console.log(localVar);  // ✅ Accessible
}
test();
console.log(localVar); // ❌ ReferenceError
```

Q7) Should you terminate all lines with a `;`?

✅ **Best Practice: Yes**, but...

JavaScript uses **Automatic Semicolon Insertion (ASI)** to add semicolons where it *thinks* they should be — but it can **get it wrong** in tricky cases (like the next question 👇).

So, while **not always required**, **explicitly adding semicolons** is **recommended** to avoid **weird bugs**, especially when:

- Returning values
- Using IIFEs
- Chaining statements

Q8) Why is this code returning `undefined` despite returning an object?

```
function test(){
  return
  {
    a: 5
  }
}
const obj = test();
console.log(obj); // undefined
```

🔍 What's going on?

This is a classic ASI trap.

JavaScript inserts a semicolon **right after** `return`, like this:

```
function test(){
  return; // ASI inserts semicolon here 😬
  {
    a: 5
  }
}
```

So the function returns `undefined`, and the object is just ignored.

✅ How to fix it:

```
function test() {
  return {
    a: 5
  };
}
```

OR: Put the object on the **same line** as `return`.

Q9) Can `'use strict'` or strict mode change the behavior of ASI?

🚫 No, `'use strict'` does NOT change how ASI works.

ASI works the same in both strict and non-strict mode. However, strict mode:

- Disallows certain bad practices (like using undeclared variables).
- Throws more errors instead of silently failing.
- Enforces cleaner code.

But it **doesn't change** how semicolons are inserted automatically.

✅ TL;DR Summary:

Question	Answer
Should you use `;` always?	Yes, to avoid ASI pitfalls.

Why is `return { a: 5 }` returning `undefined`?	ASI inserts semicolon after `return`.
Does `'use strict'` affect ASI?	No, ASI behavior stays the same.

✅ Q10) Can we use the `arguments` object in arrow functions?

No, arrow functions do NOT have their own `arguments` object.

- If you use `arguments` inside an arrow function, it will reference the `arguments` of the **outer non-arrow function**, or throw an error if none exists.

❌ Invalid:

```
const arrow = () => {
  console.log(arguments); // ReferenceError
}
arrow(1, 2);
```

✅ Correct alternative: Use rest parameters

```
const arrow = (...args) => {
  console.log(args); // [1, 2]
}
arrow(1, 2);
```

✅ Q11) What is the best way to create new arrays with assignment?

It depends on your use case. Here are the **best and cleanest options**:

Use Case	Best Method
Create from values	`let arr = [1, 2, 3];`
Clone another array	`let copy = [...original];`
Generate pattern	`Array.from({length: n}, (_, i) => i)`
Fill array	`Array(5).fill(0);`

Examples:

```
let arr1 = [1, 2, 3]; // Direct assignment
let arr2 = [...arr1]; // Clone
let arr3 = Array.from({length: 5}, (_, i) => i * 2); // [0, 2, 4, 6, 8]
let arr4 = Array(3).fill("JS"); // ["JS", "JS", "JS"]
```

✅ Q12) How can you handle `n` number of parameters in a function?

Use the **rest parameter (`...args`)** to collect all arguments into an array.

Function to return sum:

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

Function to return max:

```
function max(...numbers) {
  return Math.max(...numbers);
}
console.log(max(5, 8, 2)); // 8
```

Works in arrow functions too:

```
const product = (...nums) => nums.reduce((a, b) => a * b, 1);
console.log(product(2, 3, 4)); // 24
```

✅ Q13) Can the rest operator be placed anywhere in the function parameter list?

No, it must be the last parameter.

❌ Invalid code:

```
function test(...a, b) {
  // SyntaxError
}
```

❗ Reason:

The JavaScript engine wouldn't know **how many arguments** to assign to the `...a` rest parameter if there's another parameter (`b`) after it.

✅ Valid:

```
function test(a, b, ...rest) {
  console.log(rest); // Captures remaining arguments
}
```

🔚 Summary:

Question	Answer
Use `arguments` in arrow functions?	❌ No. Use `...args` instead.
Best array creation?	`[]`, `[...copy]`, `Array.from()`, `Array(n).fill()`
Handle any number of args?	✅ Use `...args`
Can rest be in middle of params?	❌ No. Must be last.

✅ Q.14) How will you put a validation for positive or negative Infinity?

To validate if a value is **positive or negative Infinity**, you can use:

➤ Option 1: Use `Number.isFinite()`

```
Number.isFinite(value) // returns false if value is Infinity or -Infinity or NaN
```

➤ Option 2: Direct comparison

```
if (value === Infinity) {
  console.log("Positive Infinity");
} else if (value === -Infinity) {
```

```
  console.log("Negative Infinity");
}
```

➤ Option 3: Check using `isFinite()` (global function)

```
if (!isFinite(value)) {
  console.log("It's Infinity or NaN");
}
```

> ✅ Number.isFinite() is more accurate because it doesn't coerce values like the global isFinite() does.

✅ Q.15) What will be the output of this code?

```
console.log(1 / 0);
```

🔁 Output:

```
Infinity
```

❓ Why?

In JavaScript:

- Dividing any **positive number by 0** returns `Infinity`
- Dividing a **negative number by 0** returns `Infinity`
- Dividing `0 / 0` returns `NaN` (Not a Number)

So,

```
console.log(1 / 0);    // Infinity
console.log(-1 / 0);   // -Infinity
console.log(0 / 0);    // NaN
```

Q16 What will be the output of the statement below?

```
console.log(NaN == NaN);
```

✅ Answer: `false`

🔍 Explanation:

In JavaScript, `NaN` (Not-a-Number) is **not equal to anything**, including itself. That's part of the ECMAScript specification.

So:

```
NaN == NaN  // false
NaN === NaN // also false
```

To check if a value is `NaN` You should **not** use equality comparisons — instead, use `isNaN()` or `Number.isNaN()`.

Q17) What is the difference between `isNaN()` and `isFinite()` method?

Feature	`isNaN(value)`	`isFinite(value)`
Purpose	Checks if a value is `NaN` after coercion	Checks if a value is a **finite number**
Returns	`true` if the value is `NaN` or coerces to `NaN`	`true` if it's a finite number (not `NaN` / `Infinity`)
Type Coercion	Yes (converts the value to number first)	Yes (also coerces the value to number)

🔍 Examples:

```
isNaN('hello');      // true  ('hello' becomes NaN)
isFinite('10');     // true  ('10' becomes 10, which is finite)
isNaN(123);          // false
isFinite(Infinity);  // false
isFinite(NaN);       // false
console.log(NaN==NaN) //false
```

👉 If you want a stricter check without type coercion, use:

- `Number.isNaN()` instead of `isNaN()`
- `Number.isFinite()` instead of `isFinite()`.

Q17) Explain the syntactical features of the arrow function

Arrow functions are a **shorter syntax** for writing functions in JavaScript, introduced in ES6.

✅ Syntax:

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

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

// Arrow function (implicit return)
const add = (a, b) => a + b;
```

🔍 Key Features:

- **Shorter syntax**
- **No `function` keyword**
- **Implicit return** (if one expression, no `{}` needed)
- **No own `this`, `arguments`, `super`, or `new.target`**

Q18) Why `this` Does it not work in an arrow function?

Arrow functions **do not have their own `this`**. Instead, they **lexically bind `this`** — meaning they use `this` from the surrounding code **where the arrow function was defined**.

🔍 So why?

- Arrow functions are designed for callbacks and short functions, where we often want to **inherit** `this` **from the outer scope** (like inside classes or event handlers).

```
function NormalFunction() {
  console.log(this); // refers to the caller
}

const ArrowFunction = () => {
  console.log(this); // refers to the outer `this` when it was defined
}
```

Q19) Explain the output of the following code with a reason

```
const obj = {
  method: () => {
    console.log(this);
  }
}
obj.method();
```

✅ Output: `Window` (in browser) or `undefined` (in strict mode / Node.js)

🔍 Reason:

- Arrow functions **don't bind their own** `this`.
- So `this` inside `method` refers to **the outer lexical scope**, not `obj`.
- Since it's defined at the top level, `this` refers to the global object (`window` in browser, or `undefined` in strict mode in Node.js).

> Even though you're calling obj.method(), the arrow function ignores the object it's in.

Q20) How can you handle `arguments` Like functionality in an arrow function?

Arrow functions **do not have their own** `arguments` **object**, but you can:

✅ Use rest parameters:

```
const func = (...args) => {
  console.log(args);
}
func(1, 2, 3); // [1, 2, 3]
```

✅ Or use closure to access `arguments` from the outer function:

```
function outer() {
  const arrow = () => {
    console.log(arguments);
  };
  arrow(4, 5);
}
outer(1, 2, 3); // [1, 2, 3]
```

Q21) Can you write IIFE with arrow function syntax?

Yes! You can write an **IIFE (Immediately Invoked Function Expression)** using an arrow function.

✅ Syntax:

```
(() => {
  console.log("IIFE with arrow function!");
})();
```

> You wrap it in parentheses to make it an expression, then invoke it right away with ().

✅ Q22) How can you access private variables or functions outside the scope?

In **modern JavaScript**, private variables are created using:

✔️ 1. Classes with `#` (true private)

```
class Person {
  #name = 'John'; // private variable

  getName() {
    return this.#name;
  }
}

const p = new Person();
console.log(p.getName()); // ✅ John
console.log(p.#name);     // ❌ Error: Cannot access private field
```

You can only access `#name` from inside the class, not from outside.

✔️ 2. Closures (older but common pattern)

You can also make variables private using **closures**, like this:

```
function Counter() {
    let count = 0; // private variable

    return {
        increment: () => count++,
        getCount: () => count
    };
}

const c = Counter();
console.log(c.getCount());  // 0
c.increment();
console.log(c.getCount());  // 1
```

🧠 `count` It is **not directly accessible** outside, but you can work with it using the methods `increment()` and `getCount()`.

It helps you:

- **Hide internal details** of your code (like how a counter works)

- **Protect data** from being accidentally changed
- **Keep state** between function calls

✅ Q23) What is the advantage of closure?

A **closure** means a function can "remember" variables from where it was created.

🔒 **Closures are useful for:**

- Keeping data **private**
- Remembering state (like a counter)
- Making code **modular** and **secure**

The `Counter()` The example above is a perfect use case of closure.

🧠 Summary:

Feature	Description
`#privateField`	New syntax, works only in classes
Closure	Function remembers outer variables
Benefit	Private data, persistent state

Sure! Here are the direct answers to your questions:

✅ Q24) **What is the function of currying?**

Function currying is a technique in JavaScript where a function with multiple arguments is transformed into a series of functions, each taking one argument at a time.

✅ Q25) `const multiplication = a ⇒ b ⇒ c ⇒ a * b * c`

This is a **curried function** that multiplies three numbers.

- `multiplication(2)` returns a function that takes `b`
- `(...)(3)` returns a function that takes `c`
- `(...)(4)` returns the final result: `2 * 3 * 4 = 24`

Example:

```
multiplication(2)(3)(4); // returns 24
```

✅ Q26) **Explain the practical usage of function currying**

- Helps create **reusable** and **configurable** functions
- Useful in **event handling**, **React props**, and **functional programming**
- Improves **code readability** by separating concerns

Example:

```
const greet = greeting => name => `${greeting}, ${name}`;
const sayHello = greet("Hello");
console.log(sayHello("Alice"));

//other way
console.log(greet("Hello")("Hriday"));
```

✅ Q27) What is the purpose of the iterator?

The **purpose of an iterator** is to provide a **standard way to access elements of a collection one at a time**, without exposing the underlying structure. It allows sequential access to data using the `next()` method and works seamlessly with constructs like `for...of`, spread syntax (`...`), and other iterable-supporting features.

Iterators are especially useful for:

- Traversing data structures (arrays, sets, maps, custom objects).
- Controlling iteration manually.
- Making custom objects iterable.

An **iterator** must have a `next()` method that returns an object with two properties:

- `value`: the current item
- `done`: a boolean indicating if the iteration is finished

✅ Q28) How do you create an iterator?

You can create an iterator in two ways:

a) Manually (Custom `next()` method)

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

b) Using `Symbol.iterator` (Standard Way)

`iterator(arr)` returns a plain object with a `next()` method, but **not an iterable**. To use `for...of` The object needs to implement the `[Symbol.iterator]()` method.

```
const range = {
  start: 1,
  end: 5,

  [Symbol.iterator]() {
    let current = this.start;
    let last = this.end;

    return {
      next() {
        if (current <= last) {
          return { value: current++, done: false };
        } else {
          return { done: true };
        }
      }
    };
```

```
  }
};

for (let num of range) {
  console.log(num); // 1 2 3 4 5
}
```

✅ Q29) Explain a practical use of an iterator

Practical Use Case: Pagination / Lazy Loading

Suppose you're fetching items in chunks (like infinite scrolling):

```
function createPaginator(items, pageSize) {
  let index = 0;

  return {
    [Symbol.iterator]() {
      return {
        next() {
          if (index < items.length) {
            const page = items.slice(index, index + pageSize);
            index += pageSize;
            return { value: page, done: false };
          }
          return { value: [], done: true };
        }
      };
    }
  };
}

const paginator = createPaginator([1, 2, 3, 4, 5, 6, 7], 3);

for (let page of paginator) {
  console.log("Page:", page);
}

// Output:
```

```
// Page: [1, 2, 3]
// Page: [4, 5, 6]
// Page: [7]
```

✅ Q30) What are generator functions? Explain the syntax.

A **generator function** is a special type of JavaScript function that can be **paused and resumed**. It returns a **generator object**, which is an iterator.

Instead of returning a value once, like normal functions, a generator can `yield` multiple values, one at a time, **on demand**.

🔷 Syntax:

```
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
```

- `function*` (note the) declares a generator.
- Inside the function, `yield` is used to pause and return values.
- Each call to `next()` resumes from where it left off.

✅ Q31) Which is the right syntax: `function* () {}` or `function *() {}` ?

Both are correct! ✅

JavaScript allows both styles:

- `function* name() {}` ✅
- `function *name() {}` ✅

However, the **more common convention** (per MDN and most style guides) is:

```
function* name() {}
```

It's mostly about readability preference.

✅ Q32) Explain all methods of generator objects

When you call a generator function, it returns a **generator object**. This object has the following methods:

🔷 `next(value)`

- Resumes execution and returns the next `{ value, done }` pair.
- Optionally passes a value into the generator (used with `yield`).
- Optionally passes a value into the generator (used with `yield`).

```
const gen = myGen();
gen.next();      // Starts the generator
gen.next("value"); // Passes "value" back to the generator
```

🔷 `return(value)`

- Forces the generator to finish immediately.
- Returns `{ value, done: true }`.
- Any remaining `yield` Statements are skipped.

```
gen.return("bye"); // Forces termination
```

🔷 `throw(error)`

- Throws an error inside the generator at the point of the current `yield`.
- Can be caught with a `try...catch` Inside the generator.

```
gen.throw(new Error("Oops"));
```

✅ Q33) Explain the use of `yield*`

The `yield*` The expression is used to **delegate to another generator or iterable**. It allows a generator to yield values from another generator or iterable as if they were its own.

🔷 Example:

```
function* numbers() {
  yield 1;
  yield 2;
}

function* moreNumbers() {
  yield* numbers(); // Delegating
  yield 3;
}

const gen = moreNumbers();
console.log([...gen]); // [1, 2, 3]
```

This helps you **compose generators** cleanly and reuse logic.

✅ Q34) Can you prevent `return()` from terminating the generator?

👉 **No**, you **cannot prevent** `return()` from terminating a generator.

Calling `generator.return()` will **immediately terminate** the generator, and it won't yield any more values, even if there's more code after the `yield`.

Even if you use `try...finally`, the `finally` block will run, but the generator will still be considered **done**.

🔷 Example:

```
function* example() {
  try {
    yield 1;
    yield 2;
  } finally {
    console.log("Cleanup"); // This will run
  }
}

const gen = example();
console.log(gen.next());  // { value: 1, done: false }
```

```
console.log(gen.return()); // "Cleanup", then { value: undefined, done: true }
console.log(gen.next());   // { value: undefined, done: true }
```

As you can see, after `return()`, the generator is terminated.

✅Q35. What is the concept of GC (Garbage Collection) in JavaScript? Explain with an example. Also, explain the Mark-and-Sweep algorithm.

🧠 What is Garbage Collection (GC) in JavaScript?

Garbage Collection is the process of **automatically reclaiming memory** that is no longer being used by the program. JavaScript has **automatic memory management**, so developers don't need to manually allocate or free memory.

When variables or objects are no longer **reachable** (i.e., no part of the program can access them), the JavaScript engine considers them "garbage" and removes them to free up space.

📌 Example of Garbage Collection:

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

greet();
// After the function finishes, 'message' is no longer accessible and is garbage collected
```

Here:

- `message` is created inside the function `greet`.
- Once the function completes, `message` goes out of scope.
- Since no one can access it anymore, it becomes **unreachable**.
- The Garbage Collector will clean it up.

🧹 Mark-and-Sweep Algorithm

This is one of the most common garbage collection algorithms used by JavaScript engines (like V8).

🔍 How it works:

1. **Mark Phase**:
   - Start from **roots** (like global objects or local variables in scope).
   - Recursively **mark** all objects that are **reachable** from the roots.
2. **Sweep Phase**:
   - Iterate through all objects in memory.
   - Any object **not marked** is considered **unreachable** and is **deleted**.

✅ Mark-and-Sweep Example:

```
let a = {
  name: "Alice"
};

let b = {
  friend: a
};

a = null; // a is null, but the object is still referenced by b.friend
```

- Even though `a` is set to `null`, the object `{ name: "Alice" }` is still reachable through `b.friend`.
- If `b` is also set to `null`, then there are **no references left**, and the object becomes unreachable.
- The GC will then **remove** it from memory.

📘 Summary:

- **Garbage Collection** helps manage memory by removing unreachable data.
- **Mark-and-Sweep**:

- **Mark** all reachable objects.
- **Sweep** and remove unmarked (unreachable) ones.

✅ Q.36) When do you need `try...catch`?

You use `try...catch` when you want to **handle errors gracefully** instead of letting them crash your program.

✅ Use it when:

- You expect something **might fail** (e.g., API calls, JSON parsing).
- You want to **handle exceptions** and show custom behavior.

```
try {
  let data = JSON.parse("{ bad JSON }");
} catch (error) {
  console.log("Failed to parse JSON:", error.message);
}
```

✅ Q.37) How can you generate an error?

You can generate an error using the `throw` statement:

```
throw new Error("Something went wrong!");
```

Or throw custom error types:

```
throw new TypeError("Expected a string");
```

✅ Q.38) Can you generate `SyntaxError` or `ReferenceError` kind of errors?

Yes, you can manually throw these types of errors:

```
throw new SyntaxError("Invalid syntax!");
throw new ReferenceError("x is not defined!");
```

⚠️ Note: Actual syntax errors during parsing (e.g., a missing bracket) can't be caught with `try...catch` if they prevent the code from running at all.

✅ Q.39) What is the purpose of the `finally` block?

The `finally` block runs **regardless of whether an error occurred or not**. It's great for cleanup code like:

- Closing resources
- Logging
- Releasing memory

```
try {
  console.log("Doing something...");
} catch (err) {
  console.log("Caught an error");
} finally {
  console.log("Always runs!");
}
```

✅ Q.40) How can you refer to the name or description of an error?

You can access them using `error.name` and `error.message`:

```
try {
  throw new TypeError("This is a type error!");
} catch (error) {
  console.log("Error Name:", error.name);       // TypeError
  console.log("Error Message:", error.message); // This is a type error!
}
```

✅ Q.41) Can we have it `finally` without a `catch` block as well?

Yes! You can use `try...finally` without a `catch`.

```
try {
  console.log("Doing something risky...");
} finally {
```

```
  console.log("Always runs, error or not.");
}
```

But if an error occurs and there's no `catch`, the error will still be thrown **after** the `finally` block finishes.

✅ Q.42) What is the difference between `for...in` and `for...of`?

Feature	`for...in`	`for...of`
Iterates over	**Keys (property names)**	**Values (of iterable objects)**
Suitable for	Objects, Arrays (for keys/indexes)	Arrays, Strings, Maps, Sets (for values)
Use case	When you need to access **keys**	When you need to access **values**

🔍 Example:

```
let arr = ['a', 'b', 'c'];

for (let key in arr) {
  console.log(key); // 0, 1, 2 (indexes)
}

for (let value of arr) {
  console.log(value); // a, b, c (values)
}
```

✅ Q.43) What will be the output of the code below?

```
let obj = { a: 1, b: 2, c: 3 };

for (let key in obj) {
  console.log(key, obj[key]);
}
```

✅ Output:

```
a 1
b 2
c 3
```

Explanation:

- `for...in` iterates over the object's **keys** (`a`, `b`, `c`).
- `obj[key]` gives the corresponding values (`1`, `2`, `3`).

✅ Q.44) What will be the output of below statements?

```
let str = "hi";

for (let char in str) {
  console.log(char);
}

for (let char of str) {
  console.log(char);
}
```

✅ Output:

Using `for...in`:

```
0
1
```

Using `for...of`:

```
h
i
```

Explanation:

- `for...in` gives the **indexes** (as strings) → `"0"`, `"1"`.
- `for...of` gives the **characters** directly → `"h"`, `"i"`.

✅ Q.45) What is the difference between the `push()` and `unshift()` methods?

- `push()` adds an element to the **end** of an array.
- `unshift()` adds an element to the **beginning** of an array.

```
let arr = [1, 2];
arr.push(3);      // [1, 2, 3]
arr.unshift(0);   // [0, 1, 2, 3]
```

✅ Q.46) What is the difference between `pop()` and `shift()`?

- `pop()` removes the **last** element from an array.
- `shift()` removes the **first** element from an array.

```
let arr = [1, 2, 3];
arr.pop();   // returns 3, array becomes [1, 2]
arr.shift(); // returns 1, array becomes [2]
```

✅ Q.47) How can you insert an element at a given position?

Use the `splice()` method.

```
let arr = [1, 2, 4];
arr.splice(2, 0, 3);  // insert 3 at index 2 → [1, 2, 3, 4]
array.splice(start, deleteCount, item1, item2, ...)

//start – The index at which to start changing the array.

//deleteCount – (Optional) The number of elements to remove.

//item1, item2, ... – (Optional) Items to add to the array starting from the star
t index.
//splice() modifies the original array.

//It returns an array of removed elements.
```

✅ Q.48) How can you remove a specific element?

Use `splice()` along with `indexOf()` to find the element.

```
let arr = [1, 2, 3];
let index = arr.indexOf(2);
if (index !== -1) arr.splice(index, 1);  // [1, 3]
```

✅ Q.49) What does `splice()` return?

It returns an **array of removed elements**.

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

✅ Q.50) If there is no element removed, then what will the `splice()` method return?

It returns an **empty array** `[]`.

```
let arr = ["One", "Two", "Three", "Four", "Five"];
console.log(arr.splice(2, 0, "New"));  //Output: []
```

✅ Q.51) What is the difference between `find()` and `filter()` method?

- `find()` returns the **first matching element**.
- `filter()` returns **all matching elements** in a new array.

```
let arr = [1, 2, 3, 4];
arr.find(x ⇒ x > 2);   // returns 3
arr.filter(x ⇒ x > 2); // returns [3, 4]
```

✅ Q.52) If there is no value to return, what will `findIndex()` return?

- It returns `-1` if no element matches the condition.

```
let arr = [1, 2, 3];
arr.findIndex(x ⇒ x === 5); // returns -1
```

✅ Q.53) What is the difference between `indexOf()` and `includes()` method?

- `indexOf()` returns the **index** of the element (1 or 0 if not found).

- `includes()` returns `true` or `false` based on whether the element exists.

```
let arr = [1, 2, 3];
arr.indexOf(2);     // returns 1
arr.includes(2);    // returns true
```

✅ Q.54) How will you search multiple values in an array?

- Use `filter()` or `includes()` in combination:

```
let arr = [1, 2, 3, 4, 5];
let searchValues = [2, 4];
let result = arr.filter(x ⇒ searchValues.includes(x));  // [2, 4]
```

✅ Q.55) What will be the output of this code?

```
let arr = ["One", "Two", "Three", "Four", "Five"];
console.log(arr.lastIndexOf("Abcd"));
```

- Output: - `1`

  👉 `"Abcd"` is not in the array, so `lastIndexOf()` returns `-1.`

```
array.map(callback(currentValue, index, array), thisArg)

//The map() method in JavaScript is used to transform each element of an array and return a new array with the results.

//callback: A function that gets called for every element in the array.

//currentValue: The current element being processed.

//index (optional): The index of the current element.

//array (optional): The full array being mapped.

//thisArg (optional): Value to use as this inside the callback.
```

✅Q.56) Find the length of each element in a new array.

Use `map()` to get the length of each string element.

```
let arr = ["One", "Two", "Three", "Four", "Five"];
let lengths = arr.map(item ⇒ item.length);
console.log(lengths);  // [3, 3, 5, 4, 4]
```

✅ Q.57) Find the square root of every element and store it in a new array.

Use `map()` with `Math.sqrt()`.

```
let numbers = [1, 4, 9, 16, 25];
let roots = numbers.map(num ⇒ Math.sqrt(num));
console.log(roots);  // [1, 2, 3, 4, 5]

//another example

const multiplier = {
  factor: 3
};

const numbers = [1, 2, 3];

const result = numbers.map(function (num) {
  return num * this.factor;  // `this` refers to the `multiplier` object
}, multiplier); // Passing `thisArg`

console.log(result); // [3, 6, 9]
```

✅ Q.58) There is an array called products:

```
let products = [
  { pCode: 1, pName: "Apple" },
  { pCode: 2, pName: "Banana" },
  { pCode: 3, pName: "Grapes" },
```

```
  { pCode: 4, pName: "Oranges" }
];
```

Get all product names (`pName`) in a new array:

```
let productNames = products.map(product ⇒ product.pName);
console.log(productNames);  // ["Apple", "Banana", "Grapes", "Oranges"]
```

✅ Q.59) How will you flatten an array (e.g., converting a 2D array into a 1D)?

Use the `.flat()` method or `reduce()` to flatten nested arrays.

```
let arr = [[1, 2], [3, 4], [5]];
let flatArr = arr.flat();  // [1, 2, 3, 4, 5]
```

For deeper levels: `arr.flat(Infinity)`

```
array.reduce(callback(accumulator, currentValue, index, array), initialValue)

//Parameters:
//callback: A function run for each element in the array.

//accumulator: The running total or combined value.

//currentValue: The current element being processed.

//index (optional): The index of the current element.

//array (optional): The full array.

//initialValue: (optional but recommended) Starting value for the accumulat
or.
```

✅ Q.60) Get the sum of a key field of an object literal (e.g., total salary):

```
const employees = [
  { eNo: 1001, salary: 3000 },
  { eNo: 1002, salary: 2200 },
  { eNo: 1003, salary: 3400 },
```

```
  { eNo: 1004, salary: 6000 }
];

let totalSalary = employees.reduce((sum, emp) => sum + emp.salary, 0);
console.log(totalSalary);  // 14600
```

✅ Q.61) Find the average value of all elements of an array:

```
let arr = [10, 20, 30, 40, 50];
let average = arr.reduce((sum, val) => sum + val, 0) / arr.length;
console.log(average);  // 30
```

✅ Q.62) Find the sum or product of all elements:

```
let arr = [1, 2, 3, 4];
let sum = arr.reduce((acc, val) => acc + val, 0);        // 10
let product = arr.reduce((acc, val) => acc * val, 1);    // 24
```

✅ Q.63) What is the difference between `reduce()` and `reduceRight()`?

- `reduce()` processes the array from **left to right** (start to end).
- `reduceRight()` processes the array from **right to left** (end to start).

```
let arr = ["a", "b", "c"];
arr.reduce((acc, val) => acc + val);       // "abc"
arr.reduceRight((acc, val) => acc + val);  // "cba"
```

✅ Q.64) What will be the output if an array is `undefined` while sorting the values?

- `undefined` values are usually **moved to the end** of the array when using `sort()`.

```
let arr = [5, undefined, 3, 1];
arr.sort();
console.log(arr);  // [1, 3, 5, undefined] (sorted as strings, undefined at end)
```

✅ Q.65) How will you sort an object literal?

You **can't directly sort an object**, but you can sort an **array of objects** by a key using `sort()`.

```
let products = [
  { pCode: 3, pName: "Grapes" },
  { pCode: 1, pName: "Apple" },
  { pCode: 2, pName: "Banana" }
];

// Sort by pCode
products.sort((a, b) ⇒ a.pCode - b.pCode);

console.log(products);
// [
//   { pCode: 1, pName: "Apple" },
//   { pCode: 2, pName: "Banana" },
//   { pCode: 3, pName: "Grapes" }
// ]
```

✅ Q.66) How will you sort a numeric array?

Use a custom compare function to ensure numeric sorting:

```
let nums = [40, 10, 5, 100];
nums.sort((a, b) ⇒ a - b);  // Ascending
console.log(nums);  // [5, 10, 40, 100]
```

✅ Q.67) Sort all values of the array in descending order:

Use `sort()` with a descending compare function:

```
let nums = [40, 10, 5, 100];
nums.sort((a, b) ⇒ b - a);
console.log(nums);  // [100, 40, 10, 5]
```

✅ Q.68) What is the destructuring assignment?

Destructuring assignment is a **syntax** that lets you **unpack values** from arrays or properties from objects into separate variables.

```
let arr = [1, 2, 3];
let [a, b, c] = arr;
console.log(a, b, c);  // 1 2 3
```

✅ Q.69) Swap values using destructuring:

You can swap two variables **without using a temporary variable**.

```
let x = 10, y = 20;
[x, y] = [y, x];
console.log(x, y);  // 20 10
```

✅ Q.70) What will be the output of this code?

```
let [a, b, c] = [5, , 7];
console.log(a, b, c);
```

- Output: `5 undefined 7`

  👉 `b` is not assigned a value, so it's `undefined`.

✅ Q.71) How will you set a default value while destructuring an array?

Use `=` to assign default values.

```
let [a = 1, b = 2, c = 3] = [10];
console.log(a, b, c);  // 10 2 3
```

✅ Q.72) String basics, [UTF-16] - `\u` - Unicode

JavaScript uses **UTF-16 encoding** for strings. Unicode characters can be included using escape sequences:

```
let heart = "\u2665";      // ❤ (Basic Multilingual Plane)
let emoji = "\u{1F600}";   // 😀 (Supplementary Plane using ES6+)
console.log(heart, emoji);
```

✅ Q.73) Various ways to declare a string variable

You can declare strings using:

```
let str1 = "Double quotes";
let str2 = 'Single quotes';
let str3 = `Backticks (template literal)`; // ES6+
```

✅ Q.74) How to deal with Unicode characters

Use escape sequences to represent characters:

- Basic: `\uXXXX`
- Supplementary (emoji, symbols): `\u{XXXXX}` (ES6+)

```
let snowman = "\u2603";      // ☃
let music = "\u{1F3B5}";     // 🎵
console.log(snowman, music);
```

✅ Q.75) Syntax to display long Unicode characters

Use curly brace syntax for characters beyond 0xFFFF:

```
let rocket = "\u{1F680}";  // 🚀
console.log(rocket);
```

✅ Q.76) What is a template literal?

A **template literal** is a string defined with backticks (``) that allows:

- Multi-line strings
- Embedded expressions with `${}`
- Easier formatting

✅ Q.77) How to display a value or expression inside a template string

Use `${expression}` inside backticks:

```
let name = "Bob";
console.log(`Hello, ${name}`);
console.log(`2 * 3 = ${2 * 3}`);
```

✅ Q.78) What is the advantage of using a template string?

- Simplifies string interpolation
- Supports multi-line strings
- Easier to read and write than concatenation
- Embeds expressions directly

✅ Q.79) `.length` property and search methods

- `.length`: Returns the number of UTF-16 code units in a string.

✅ Q.80) `indexOf()` and `lastIndexOf()` with syntax

```
str.indexOf(searchValue, fromIndex);
str.lastIndexOf(searchValue, fromIndex);
```

Example:

```
let str = "This is a test";
console.log(str.indexOf("is", 5));      // 5
console.log(str.lastIndexOf("is", 1));  // -1
```

Explanation:

- `indexOf("is", 5)` starts at index 5 → finds "is" at 5.
- `lastIndexOf("is", 1)` looks backward from index 1 → does **not** find "is".

✅ Q.81) Output of:

```
let str = "Hello World";
console.log(str.slice(-5, -2)); // "Wor"
```

Explanation:

- `5` → position 6 ("W")
- `2` → position 9 ("l")
- Slices from index 6 to 9 (excluding 9) → "Wor"

✅ Q.82) Difference between `substr()` and `substring()`

Method	Parameters	Behavior

`substr()`	(start, length)	Extracts `length` characters
`substring()`	(start, end)	Extracts up to `end` (not included)

✅ Q.83) Output of:

```
let str = "This is a test";
console.log(str.substring(-5));
```

Explanation:

`substring(-5)` treats negative as 0 → equivalent to `str.substring(0)` → returns whole string.

✅ **Output:** `"This is a test"`

✅ Q.84) Output of:

```
let str = "This is a test";
console.log(str.substring(3, 3));
```

Explanation:

Start and end are equal → returns empty string.

✅ **Output:** `""`

✅ Q.85) Output of:

```
console.log(str.charAt());
```

Explanation:

`charAt()` without argument defaults to index 0 → returns first character.

✅ **Output:** `"T"` (since `str = "This is a test"`)

✅ Q.86) Explain different ways of creating a date/time object

JavaScript provides several ways to create `Date` objects:

```
// 1. Current date and time
let now = new Date();

// 2. From a date string
```

```
let dateStr = new Date("2025-05-05");

// 3. From year, month (0-based), day, hours, minutes, seconds, ms
let specific = new Date(2025, 4, 5, 10, 30, 0, 0); // May 5, 2025

// 4. From milliseconds since Unix Epoch
let fromEpoch = new Date(0); // Jan 1, 1970

// 5. Using ISO 8601 string (standard format)
let iso = new Date("2025-05-05T10:30:00Z");
```

> 🔷 Note: Month is 0-based, so 0 = January, 11 = December.

✅ Q.87) What will be the output of the code below?

```
const dt = new Date(2020, 08, 23);
console.log(dt);
```

Explanation:

- `08` is an octal literal in older JS versions but works as `8` in modern JS.
- So this is interpreted as:

  ```
  new Date(2020, 8, 23); // September 23, 2020
  ```

✅ Sample Output:

```
Wed Sep 23 2020 00:00:00 GMT+0000 (Coordinated Universal Time)
```

✅ Q.88) Explain various formats of the ISO standard followed by JavaScript

JavaScript follows **ISO 8601** for date/time strings. Common formats include:

Format Example	Description
`"2025-05-05"`	Date only (treated as UTC at midnight)
`"2025-05-05T10:30:00"`	Date + time (local timezone)
`"2025-05-05T10:30:00Z"`	Date + time in UTC (`Z` = Zulu/UTC time)

`"2025-05-05T10:30:00+05:30"`	Date + time with timezone offset

> 📌 JavaScript Date.parse() and the Date() constructor parse these ISO formats reliably.

✅ Q.89) Can you have dynamic keys with an object literal?

Yes, you can use **computed property names** (introduced in ES6) to define **dynamic keys** in an object literal using square brackets `[]`.

🔷 Example:

```
let keyName = "age";
let person = {
  name: "Alice",
  [keyName]: 25  // dynamic key
};

console.log(person.age); // ✅ Output: 25
```

✅ Q.90) How can you add read-only properties to an object?

You can make a property **read-only** using `Object.defineProperty()` and setting `writable: false`.

🔷 Example:

```
let user = {};
Object.defineProperty(user, "id", {
  value: 123,
  writable: false,  // makes it read-only
  configurable: true,
  enumerable: true
});

console.log(user.id);  // ✅ Output: 123
```

```
user.id = 456;
console.log(user.id);  // ✅ Still 123 (not changed)
```

✅ Q.91) What is property value shorthand with object literal?

The **property value shorthand** allows you to **omit the key-value repetition** if the property name is the same as the variable name.

🔷 Example:

```
let name = "John";
let age = 30;

let person = { name, age };  // shorthand for { name: name, age: age }

console.log(person);  // ✅ Output: { name: "John", age: 30 }
```

✅ Q.92) What will be the output of this code?

```
let obj = { a: 'First' };
let obj1 = obj;
obj1.a = "Second";
console.log(obj.a);
```

✅ Output:

```
Second
```

🔍 Explanation:

- `obj1` is assigned the reference to `obj`.
- So both `obj` and `obj1` point to the **same object in memory**.
- When you change `obj1.a` It also changes `obj.a`, because they are the same object.

✅ Q.93) How can we create a clone or separate copy of an object literal?

To create a **shallow copy** (clone) of an object literal (not referencing the same object), you can use:

1. **Spread Operator `...` (ES6+):**

```
let obj = { a: 1, b: 2 };
let clone = { ...obj };
```

2. **Object.assign()**:

```
let obj = { a: 1, b: 2 };
let clone = Object.assign({}, obj);
```

> 🔶 Both methods copy only top-level properties (shallow copy).
>
> For **deep copy** (including nested objects), use:

```
let deepClone = JSON.parse(JSON.stringify(obj));
```

✅ Q.94) Explain `JSON.stringify()` and `JSON.parse()` in JavaScript.

These two methods are used to **convert JavaScript objects to JSON strings and back** — useful for **data storage**, **transfer**, or **deep cloning**.

🔷 `JSON.stringify()`

- Converts a JavaScript **object or value** into a **JSON-formatted string**.
- Useful for sending data to a server or saving in localStorage.

✅ Example:

```
let person = { name: "Alice", age: 25 };
let jsonString = JSON.stringify(person);

console.log(jsonString);  // Output: '{"name":"Alice","age":25}'
console.log(typeof jsonString); // Output: string
```

🔷 `JSON.parse()`

- Converts a **JSON-formatted string** back into a **JavaScript object**.
- Useful when receiving JSON data from a server.

✅ Example:

```
let jsonStr = '{"name":"Alice","age":25}';
let personObj = JSON.parse(jsonStr);

console.log(personObj.name);  // Output: Alice
console.log(typeof personObj); // Output: object
```

🔄 Common Use Case: Deep Cloning

```
let original = { a: 1, b: { c: 2 } };
let clone = JSON.parse(JSON.stringify(original));

clone.b.c = 999;
```

```
console.log(original.b.c); // Output: 2 (✅ Not affected)
```

> ✅ Important: This method performs a deep copy, but:
>
> - It **won't work with** functions, `undefined`, `Symbol`, circular references, or special object types like `Date`, `Map`, or `Set`.

✅ Q.95) What will be the output of this code if you run it in the browser, and why?

```
function test(){
  console.log(this);
}
test();
```

✅ Output (in a browser):

```
Window {...}  // the global object in browsers
```

🔍 Explanation:

- In **non-strict mode**, when a regular function is called like `test()`, `this` refers to the **global object**, which is `window` in the browser.
- If you ran this in **strict mode (`'use strict'`)**, `this` would be `undefined`.

✅ Q.96) What is the context of `this` inside an arrow function?

Or

What will be the output of this code?

```
let obj = {
  test: () => {
    console.log(this);
  }
```

```
};
obj.test();
```

✅ Output:

```
Window {...}  // again, the global object in node js env
```

🔍 Explanation:

- Arrow functions **do not have their own** `this`.
- Instead, they **inherit** `this` **from their surrounding lexical scope**.
- In this example, the surrounding scope of the arrow function is the **global scope**, not the `obj` object.
- Therefore, `this` inside the arrow function refers to `window` (in browsers), **not** `obj`.

> ✅ To access the object context correctly, use a regular function, like:

```
let obj = {
   test() {
      console.log(this);  // ✅ will print obj
   }
};
```

✅ Q.97) What is `this` in JavaScript?

Answer:

- `this` is a **keyword** in JavaScript that refers to the **context** in which a function is executed.
- Its value depends on **how** the function is called, not where it's defined.

✅ Q.98) What is the value of `this` inside a regular function?

Answer:

- In **non-strict mode**, `this` refers to the **global object** (`window` in browsers) when a regular function is called in the global context.
- In **strict mode**, `this` is `undefined`.

```
function test() {
  console.log(this);
}
test(); // window (non-strict), undefined (strict)
```

✅ Q.99) How does `this` Behave inside an arrow function?

Answer:

- Arrow functions do **not have their own** `this`.
- Instead, they **inherit** `this` **from their lexical (surrounding) scope**.

```
let obj = {
  name: "Alice",
  arrowFn: () => {
    console.log(this.name); // `this` is from outer scope, not obj
  }
};

obj.arrowFn(); // undefined (in browser, it's window.name which is likely un
defined)
```

✅ Q.100) What will be the output of this code, and why?

```
let obj = {
  name: "Alice",
  greet: function () {
    console.log(this.name);
```

```
  }
};

let greetFn = obj.greet;
greetFn();
```

Answer:

```
undefined
```

Explanation:

- `greetFn` is assigned to `obj.greet` but called **without context** (`greetFn()`).
- So `this` inside the function refers to the **global object**, not `obj`.

✅ Q.101) What will be the output of the following, and explain why?

```
javascript
CopyEdit
let obj = {
  name: "Bob",
  greet: () => {
    console.log(this.name);
  }
};

obj.greet();
```

Answer:

```
undefined
```

Explanation:

- Arrow functions inherit `this` from the **lexical scope** (likely the global scope).
- So `this.ame` refers to `window.name` (undefined unless set), not `obj.name`.

✅ Q.107) What is the difference between `call()`, `apply()`, and `bind()`?

Answer:

All three are used to **set the value of** `this` inside a function, but they differ in **syntax and behavior**.

Method	Usage	Executes Immediately?	Arguments Passed
`call`	`fn.call(thisArg, arg1, arg2, ...)`	✅ Yes	Individual arguments
`apply`	`fn.apply(thisArg, [arg1, arg2, ...])`	✅ Yes	Arguments as an array
`bind`	`let newFn = fn.bind(thisArg, arg1, ...)`	❌ No (returns a new fn)	Individual arguments (deferred execution)

🔷 Example:

```
function greet(greeting, name) {
  console.log(greeting + " " + name + " from " + this.company);
}

const obj = { company: "OpenAI" };

greet.call(obj, "Hello", "Alice");  // Hello Alice from OpenAI
greet.apply(obj, ["Hi", "Bob"]);    // Hi Bob from OpenAI

const boundGreet = greet.bind(obj, "Hey");
boundGreet("Carol");              // Hey Carol from OpenAI
```

✅ Q.108) What will be the output of this code?

Or

Can you assign value to `this` using the assignment operator `=`?

```
const obj = { a: 6 };

function test() {
  this = obj;
}
test();
```

✅ Output:

❌ Error: Invalid left-hand side in assignment

🔍 Explanation:

- In JavaScript, `this` **It is a special keyword** — you **cannot assign a value to** `this` **directly** using the `=` operator.
- `this` is **automatically set** based on how a function is called (regular call, method call, constructor, etc).
- Trying to assign a value to `this` like `this = obj` Inside a function results in a **SyntaxError**.
