Top JavaScript Interview Questions (with Answers) 💡

1. What is the difference between `==` and `===` in JavaScript?

In JavaScript, `==` is the **loose equality** operator, which compares two values for equality after performing type coercion if necessary. This means it converts the operands to the same type before comparing.

`===` is the **strict equality** operator, which compares both the values and their types, without performing type conversion.

2. What would be the result of 3+2+"7"?

`let result = 3 + 2 + "7";`

```
console.log(result);
```

Output:

```
57
```

- `3 + 2` is evaluated first, and since both are numbers, it results in `5`.
- Then, `5 + "7"` is calculated. Since one of the operands is a string (`"7"`), JavaScript converts the number `5` to a string and concatenates it with `"7"`, resulting in `"57"`.

3. What's new in ECMAScript 2025 (ES2025)?

ECMAScript 2025 (ES2025) Key Features

- **`Promise.withResolvers()`** → Returns a promise with its `resolve` and `reject`, simplifying async control.
- **Immutable Array Methods** → New methods like `findLast()`, `toReversed()`, and `toSorted()` return new arrays instead of mutating originals.

- **RegExp `v` flag** → Adds better Unicode handling in regular expressions.
- **Hashbang grammar** → Officially allows `#!` at the start of JS files for CLI scripts.

4. Is JavaScript compiled or interpreted?

JavaScript is mostly interpreted, but modern browsers also compile it just-in-time (JIT) to make it faster.

- The browser reads your JavaScript code line by line and runs it directly.
- Modern browsers (like Chrome's V8 engine) first translate parts of your code into machine code while running it, so the computer can execute it much faster.

5. Are JavaScript and Java related?

No, there names sounds similar but they are not related in any terms, below are some key differences:

Java	JavaScript
Java is a strongly typed language and variables must be declared first to use in the program. In Java, the type of a variable is checked at compile-time.	JavaScript is a loosely typed language and has a more relaxed syntax and rules.
Java is an object-oriented programming language primarily used for developing complex enterprise applications.	JavaScript is a **scripting language** used for creating interactive and dynamic web pages.
Java applications can run in any virtual machine(JVM) or browser.	JavaScript code used to run only in the browser, but now it can run on the server via Node.js.
Objects of Java are class-based even we can't make any program in java without creating a class.	JavaScript Objects are prototype-based.

> To learn more you can refer to difference between Java and JavaScript.

6. How many ways an HTML element can be accessed in JavaScript code?

There are four possible ways to access HTML elements in JavaScript which are:

- **getElementById() Method:** It is used to get the element by its id name.
- **getElementsByClass() Method:** It is used to get all the elements that have the given classname.
- **getElementsByTagName() Method:** It is used to get all the elements that have the given tag name.
- **querySelector() Method:** This function takes CSS style selector and returns the first selected element.

7. What's the return-value difference between `x++` and `++x`?

Both increment, but return different values.

- `x++`: **post-increment** → returns the old value, then increments.
- `++x`: **pre-increment** → increments first, then returns the new value.

8. What's the difference between `var`, `let`, and `const` and what is the Temporal Dead Zone?

JavaScript provides three ways to declare variables: var, let, and const, but they differ in scope, hoisting behaviour, and re-assignment rules.

- **var:** Declares variables with function or global scope and allows re-declaration and updates within the same scope.
- **let:** Declares variables with block scope, allowing updates but not re-declaration within the same block.
- **const:** Declares block-scoped variables that cannot be reassigned after their initial assignment.

```
// var: function or global scope, can be re-declared and updated
var x = 10;
var x = 20; // re-declaration allowed
console.log("var:", x); // 20

// let: block scope, can be updated but not re-declared in same block
let y = 30;
```

```
// let y = 40; Error (can't re-declare in same block)
y = 40; // update allowed
console.log("let:", y); // 40

// const: block scope, cannot be reassigned
const z = 50;
// z = 60; Error (can't reassign)
console.log("const:", z); // 50
```

Temporal Dead Zone (TDZ): It's the period between entering a scope and the point where a `let` or `const` variable is declared. During this time, accessing the variable causes a ReferenceError, because the variable exists but hasn't been initialized yet.

```
// TDZ example
console.log(x);          // ReferenceError (x in TDZ)
let x = 10;

console.log(y);          // undefined (var is initialized at hoist time)
var y = 10;
```

9. What is a Variable Scope in JavaScript?

In JavaScript, we have each variable are accessed and modified through either one of the given scope:

- **Global Scope**: Outermost level (accessible everywhere).
- **Local Scope**: Inner functions can access variables from their parent functions due to lexical scoping.
- **Function Scope**: Variables are confined to the function they are declared in.
- **Block Scope**: Variables declared with `let` or `const` are confined to the nearest block (loops, conditionals, etc.).

10. What the difference between Lexical and Dynamic Scoping?

Lexical Scoping (Static Scoping)

- The scope of a variable is determined by its position in the source code at the time of writing.
- JavaScript uses lexical scoping.
- The inner function looks up variables in the outer function where it was defined, not where it was called.

Dynamic Scoping (Not in JS)

- The scope is determined by the call stack at runtime, not where the function is written.
- Languages like older versions of Lisp or Bash use dynamic scoping.
- The function uses variables from the function that called it, even if it was defined else.

11. What is the use of `isNaN`, and how is it different from `Number.isNaN`?

The number **isNan** function determines whether the passed value is NaN (Not a number) and is of the type "Number". In JavaScript, the value NaN is considered a type of number. It returns true if the argument is not a number, else it returns false.

- **`Number.isNaN(x)`** → returns `true` **o**nly if `x` is the `NaN` value. No coercion.
- **`isNaN(x)`** → converts `x` to a number then checks if that result is `NaN`.

```
Number.isNaN(NaN)           // true
Number.isNaN("foo")         // false  (string, not NaN)
isNaN("foo")                // true   (coerces "foo" → NaN)

Number.isNaN(undefined)     // false
isNaN(undefined)            // true   (undefined → NaN)

Number.isNaN("")            // false
isNaN("")                   // false  ("" → 0)
```

```
Number.isNaN(0/0)          // true   (is NaN)
isNaN(0/0)                 // true
```

12. What does this code log?

```
const arr = [1, 2, 3];
arr[10] = 99;
console.log(arr.length);
```

Output: 11

Explanation: When you assign to `arr[10]` you create empty slots from index 3 to 9, making the new `length` one more than the highest index: `10 + 1 = 11`.

13. What is negative infinity?

The negative infinity is a constant value represents the lowest available value. It means that no other number is lesser than this value. It can be generate using a self-made function or by an arithmetic operation. JavaScript shows the NEGATIVE_INFINITY value as -Infinity.

14. Why is `typeof null === "object"`?

It's a long-standing historical bug in the original `typeof` tag encoding that can't be fixed without breaking the web.

- `null` is a primitive, but `typeof null` returns `"object"`.
- Practical tip: check for `null` using `value === null` or `value == null` (to match `null` or `undefined` intentionally).

15. Is it possible to break JavaScript Code into several lines?

Yes, it is possible to break the JavaScript code into several lines in a string statement. It can be broken by using the **'\n'** (backslash n).

Example:

```
console.log("A Online Computer Science Portal\n for Geeks")
```

The code-breaking line is avoid by JavaScript which is not preferable.

```
let gfg= 10, GFG = 5,
Geeks =
gfg + GFG;
console.log(Geeks)
```

16. What do you mean by Null in JavaScript?

The **null** value represents that no value or no object. It is known as empty value/object.

17. How to delete property-specific values?

The **delete keyword** deletes the whole property and all the values at once like

```
let gfg={Course: "DSA", Duration:30};
delete gfg.Course;
```

18. What will be the output of this code?

```
let x = 0;
console.log(x++);
console.log(++x);
```

Output:

```
0
2
```

Explanation:

- `x++` returns the current value (0), then increments → `x` becomes 1.
- `++x` increments first (1 → 2), then returns the new value (2).

19. What is the difference between null and undefined in JavaScript?

undefined:

- A primitive value automatically assigned to:
- Uninitialized variables
- Missing function arguments
- Missing object properties

It means: **"value not assigned yet"**

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

function foo(a) {
  console.log(a); // undefined if no argument is passed
}
foo();
```

null:

- A primitive value that you assign intentionally to represent:
- "no value", "empty", or "non-existent"
- It means: "value is deliberately empty"

```
let user = null; // explicit assignment
console.log(user); // null
```

20. What are template literals and when do you use them?

Backtick strings that support interpolation, multi-line text, and tagged processing.

- `${expr}` interpolation without manual concatenation.
- Preserve newlines/indentation as written.
- Tagged templates for advanced parsing (e.g., sanitization, i18n).

```
const name = "Geeks";
const msg = `Hello, ${name}!
Your total is $${(19.99 * 2).toFixed(2)}.`;
```

21. What is the output of this snippet?

```
const a = [1, 2, 3];
const b = [1, 2, 3];
console.log(a == b, a === b);
```

Output:

```
false
false
```

Explanation: Arrays are objects and compared by reference. `a` and `b` are distinct objects, so both loose (`==`) and strict (`===`) comparisons yield `false`.

22. Can closures leak memory?

JavaScript closures capture variables from their outer scope, but if not managed properly, they can sometimes lead to memory leaks.

Memory Leak Possibility: Closures can leak memory when they unintentionally keep references to variables or objects that are no longer needed. This prevents the garbage collector fro5 freeing that memory.

23. Does JavaScript allow multiple inheritance?

JavaScript does not support multiple inheritance in the traditional sense, but it provides ways to reuse and combine functionality.

- **Classes**: JavaScript classes only allow single inheritance, meaning a class can extend only one parent class.
- **Prototypes**: Objects can inherit from one prototype at a time, not multiple.

- **Mixins**: To achieve similar behavior to multiple inheritance, JavaScript uses *mixins*—functions or objects that copy properties and methods into a class or object.

24. Is JavaScript statically typed or dynamically typed?

JavaScript is dynamically typed because we don’t have to tell JavaScript what kind of data (number, text, true/false, etc.) a variable will hold when you create it. The type is decided automatically when the program runs.

25. What is the 'this' keyword in JavaScript?

Functions in JavaScript are essential objects. Like objects, it can be assign to variables, pass to other functions, and return from functions. And much like objects, they have their own properties. 'this' stores the current execution context of the JavaScript program.

```
// 1. Global scope
console.log(this); // window (in browser)

// 2. Inside an object method
const obj = {
  name: "JS",
  say() {
    console.log(this.name);
  }
};
obj.say(); // "JS"

// 3. Regular function
function test() {
  console.log(this);
}
test(); // window (or undefined in strict mode)

// 4. Arrow function
const arrow = () ⇒ console.log(this);
```

```
arrow(); // inherits from outer scope (global → window)

// 5. Constructor function
function Person(name) {
  this.name = name;
}
const p = new Person("Alice");
console.log(p.name); // "Alice"
```

Thus, when it use inside a function, the value of 'this' will change depending on how the function is defined, how it is invoked, and the default execution context.

To Be Continued...

FOLLOW FOR MORE 💡