JavaScript Output Questions with Explanations

---

1. Spread Operator with Array

```
let arr = [1, 2, 3];
let arr2 = [...arr, arr];
console.log(arr2);
```

Output: `[1, 2, 3, [1, 2, 3]]`
Explanation: The spread operator expands elements of `arr`, but the second `arr` adds the entire array as a nested element.

---

2. Variable Hoisting in Function

```
var a = 1;
function foo() {
  console.log(a);
  var a = 2;
}
foo();
```

Output: `undefined`
Explanation: `var a` inside `foo` is hoisted and initialized as `undefined`, shadowing the outer variable.

---

3. Function and Variable Hoisting

```
console.log(typeof abc);
var abc = function () {};
function abc() {}
```

Output: `function`
Explanation: Function declarations are hoisted before variable declarations, so `abc` refers to the hoisted function.

---

4. Closure Counter Example

```
function createCounter() {
  var count = 0;
  return {
    increment: function () { count++; },
    get: function () { return count; }
```

```
  };
}
var counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.get());
```

Output: 2
Explanation: count is private due to closure; it retains state across multiple calls.

---

5. Variable Shadowing in Nested Function

```
var a = 1;
function outer() {
  function inner() {
    console.log(a);
    var a = 2;
  }
  inner();
}
outer();
```

Output: undefined
Explanation: Inner a is hoisted and shadows the outer a, so a is undefined when logged.

---

6. Function Hoisting with Variables

```
function test() {
  console.log(a);
  console.log(foo());
  var a = 1;
  function foo() { return 2; }
}
test();
```

Output:

```
undefined
2
```

Explanation: a is hoisted as undefined, while function declarations are hoisted with full definition.

---

7. Slice vs Original Array

```
var arr = [1, 2, 3, 4, 5];
console.log(arr.slice(1, 3));
console.log(arr);
```

Output:

```
[2, 3]
[1, 2, 3, 4, 5]
```

Explanation: `slice()` does not mutate the original array; it returns a shallow copy.

---

8. Var vs Let in Loops

```
for (var i = 0; i < 3; i++) {
  console.log(i);
}
for (let j = 0; j < 3; j++) {}
console.log(j);
```

Output:

```
0
1
2
ReferenceError: j is not defined
```

Explanation: `var` is function-scoped; `let` is block-scoped and inaccessible outside the loop.

---

9. Default Function Parameters

```
function test(a = 1, b = 2) {
  console.log(a + b);
}
test();
test(5);
test(5, 10);
```

Output:

```
3
7
15
```

Explanation: Default parameters apply when arguments are not passed or are `undefined`.

---

10. Arrow Function and `this`

```
var obj = {
  name: "John",
  greet: () => {
    console.log(this.name);
  },
};
obj.greet();
```

Output: `undefined`
Explanation: Arrow functions do not bind their own `this`; they use `this` from the outer scope.

---

11. Promise Microtask Order

```
Promise.resolve()
  .then(() => console.log("1"))
  .then(() => console.log("2"));
Promise.resolve()
  .then(() => console.log("3"))
  .then(() => console.log("4"));
```

Output:

```
1
3
2
4
```

Explanation: Microtasks are queued in order. Each `.then()` executes after the previous microtask chain completes.

---

12. Type Coercion with Operators

```
console.log(5 + "5");
console.log(5 - "5");
console.log(5 * "5");
console.log(5 / "5");
```

Output:

```
55
0
25
1
```

Explanation: + performs string concatenation; other arithmetic operators coerce strings into numbers.

---

13. Equality vs Strict Equality

```
console.log(null == undefined);
console.log(null === undefined);
console.log(null == 0);
console.log(null === 0);
```

Output:

```
true
false
false
false
```

Explanation: == allows coercion between `null` and `undefined`, but === compares type as well.

---

14. Floating Point Precision

```
console.log(0.1 + 0.2 == 0.3);
console.log(0.1 + 0.2);
let val = (0.1 + 0.2).toFixed(1);
console.log(val == 0.3);
```

Output:

```
false
0.30000000000000004
true
```

Explanation: Floating-point arithmetic causes precision errors; `toFixed(1)` returns a string `'0.3'`.

---

15. Object and Array Coercion

```
console.log([] + []);
console.log({} + {});
```

Output:

```
""
"[object Object][object Object]"
```

Explanation: Arrays convert to empty strings; objects convert to `'[object Object]'` in string context.

---

16. Function Context

```
var name = "Global";
var obj = {
  name: "Object",
  method: function () {
    console.log(this.name);
    function inner() {
      console.log(this.name);
    }
    inner();
  },
};
obj.method();
```

Output:

```
Object
Global
```

Explanation: `this` in a regular function inside another function points to the global scope.

---

17. Boolean Conversion

```
console.log(!!"");
console.log(!!0);
console.log(!!"0");
console.log(!![]);
```

Output:

```
false
false
true
true
```

Explanation: `!!` converts a value to Boolean. Empty string and `0` are falsy; everything else here is truthy.

---

18. Async Function Execution Order

```
async function test() {
  console.log("1");
  await console.log("2");
```

```
  console.log("3");
}
console.log("4");
test();
console.log("5");
```

Output:

```
4
1
2
5
3
```

Explanation: `await` pauses inside `test`; the rest of the code runs synchronously first.

---

19. Async Await Execution

```
async function foo() {
  console.log("foo start");
  await bar();
  console.log("foo end");
}
async function bar() {
  console.log("bar");
}
console.log("script start");
foo();
console.log("script end");
```

Output:

```
script start
foo start
bar
script end
foo end
```

Explanation: Async functions return promises; `await` pauses within `foo`, scheduling continuation as a microtask.

---

20. Var vs Let Scope

```
function test() {
  if (true) {
    var a = 1;
    let b = 2;
    const c = 3;
```

```
  }
  console.log(a);
  console.log(b);
  console.log(c);
}
test();
```

Output:

```
1
ReferenceError: b is not defined
```

Explanation: `var` is function-scoped; `let` and `const` are block-scoped.

---

21. call, apply, bind and `this`

```
var obj = { a: 1, b: function () { console.log(this.a); } };
var store = obj.b;
store.call({ a: 2 });
store.apply({ a: 3 });
store.bind({ a: 4 })();
```

Output:

```
2
3
4
```

Explanation: `call`, `apply`, and `bind` explicitly set `this` to the provided object.

---

22. setTimeout with var in loop

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}
```

Output:

```
3
3
3
```

Explanation: `var` is function-scoped; all callbacks share the same `i` after the loop completes.

---

23. typeof of a typeof

```
console.log(typeof typeof 1);
```

Output: `string`
Explanation: `typeof 1` is `'number'`; `typeof 'number'` is `'string'`.

---

24. Mixed + and - with strings

```
console.log('5' + 2 - 1);
```

Output: `51`
Explanation: `'5' + 2` makes `'52'`; subtracting converts to number: `52 - 1 = 51`.

---

25. map without return in block body

```
const nums = [1, 2, 3];
const doubled = nums.map(n => { n * 2; });
console.log(doubled);
```

Output: `[undefined, undefined, undefined]`
Explanation: Arrow functions with `{}` need `return`; otherwise `undefined` is returned.

---

26. Function/var name hoisting

```
function foo() {
  console.log(a);
  var a = 10;
  function a() {}
  console.log(a);
}
foo();
```

Output:

```
[Function: a]
10
```

Explanation: The function declaration for `a` is hoisted first; later `a` is reassigned to `10`.

---

27. Object reference after nulling original variable

```
var a = { name: 'JS' };
var b = a;
```

```
a = null;
console.log(b);
```

Output: `{ name: 'JS' }`
Explanation: Only `a` lost its reference; `b` still points to the object.

---

28. isNaN coercion cases

```
console.log(isNaN('Hello'));
console.log(isNaN(550));
console.log(isNaN('1203'));
console.log(isNaN(true));
console.log(isNaN(false));
console.log(isNaN(undefined));
```

Output:

```
true
false
false
false
false
true
```

Explanation: `isNaN` coerces to number; `'Hello'` and `undefined` become `NaN`, others become valid numbers.

---

29. Arrow function and lexical this

```
let x = { y: 'z', print: () => this.y === 'z' };
console.log(x.print());
```

Output: `false`
Explanation: Arrow functions capture `this` from the outer scope (not `x`), so `this.y` is `undefined`.

---

30. Filtering truthy values

```
let x = [null, 0, '0', false, 'a'];
let y = x.filter(v => v);
console.log(y);
```

Output: `['0', 'a']`
Explanation: `filter` keeps only truthy values; `null`, `0`, and `false` are removed.

---

31. Microtasks vs macrotasks order

```
console.log('start');
setTimeout(() => console.log('setTimeout'), 0);
Promise.resolve()
  .then(() => console.log('promise1'))
  .then(() => console.log('promise2'));
console.log('end');
```

Output:

```
start
end
promise1
promise2
setTimeout
```

Explanation: Promise callbacks (microtasks) run before `setTimeout` (macrotask).

---

32. b ordering: numeric vs string keys

```
const obj = { 1: 'one', '07': 'zero-seven', '01': 'zero-one', 10: 'ten' };
console.log(Object.keys(obj));
```

Output: `['1', '10', '07', '01']`
Explanation: Integer-like keys are listed in ascending numeric order; string keys keep insertion order.

---

33. Sparse arrays and map

```
const arr = [1, , 3];
console.log(arr.length);
console.log(arr.map(x => x * 2));
```

Output:

```
3
[2, empty, 6]
```

Explanation: The hole counts toward `length`; `map` skips holes, preserving them as `empty`.

---

34. Function.length with default parameters

```
function test(a, b = 2, c) { return a + b + (c || 0); }
console.log(test.length);
```

Output: 1
Explanation: `.length` counts parameters before the first default; only a qualifies.

---

35. Async/await scheduling

```
async function test() {
  console.log('1');
  await Promise.resolve();
  console.log('2');
}
console.log('3');
test();
console.log('4');
```

Output:

```
3
1
4
2
```

Explanation: Code after `await` runs as a microtask after the current synchronous work.

---

36. Two Promise.resolve chains

```
Promise.resolve(1)
  .then(r => { console.log(r); return r + 1; })
  .then(r => console.log(r));

Promise.resolve(3)
  .then(r => console.log(r));
```

Output:

```
1
3
2
```

Explanation: Microtask order: first `.then` of the first chain, then the other chain's first `.then`, then the second `.then` of the first chain.

---

37. Chained comparisons pitfall

```
console.log(3 > 2 > 1);
```

Output: `false`
Explanation: `3 > 2` is `true` (coerced to `1`), and `1 > 1` is `false`.

---

38. typeof null

```
console.log(typeof null);
```

Output: `object`
Explanation: Historical spec quirk: `null` reports as `'object'` for backward compatibility.

---

39. String vs number operations

```
console.log('5' * '4');
console.log('5' + '4');
```

Output:

```
20
54
```

Explanation: * coerces both to numbers; + concatenates strings.

---

40. Calling a function before it appears

```
foo();
function foo() { console.log('Hello'); }
```

Output: `Hello`
Explanation: Function declarations are hoisted with their definitions.

---

41. Triple spread over string and length

```
console.log([...[...'123']].length);
```

Output: `3`
Explanation: Spreading a string produces an array of its characters; extra spreads do not change length.

---

42. Array reference behavior

```
let a = [1, 2, 3];
let b = a;
```

```
b.push(4);
console.log(a);
```

Output: `[1, 2, 3, 4]`
Explanation: Arrays are reference types; `a` and `b` point to the same array.

---

43. Promise rejection, catch, and finally

```
const p = new Promise((resolve, reject) => { reject('Error!'); });

p.then(res => { console.log('Success:', res); })
 .catch(err => { console.log('Caught:', err); })
 .finally(() => { console.log('Finally block'); });
```

Output:

```
Caught: Error!
Finally block
```

Explanation: Rejection is handled by `catch`; `finally` runs regardless of outcome.

---

44. String and string operations

```
console.log('5' * '4');
console.log('5' + '4');
```

Output:

```
20
54
```

Explanation: Multiplication coerces to numbers; addition concatenates strings.

---

45. Two-step typeof

```
console.log(typeof typeof 1);
```

Output: `string`
Explanation: Same as question 23; included here for completeness across sets.

---

46. Object mutation through shared reference

```
var obj = { a: 1 };
const obj2 = obj;
obj2.a = 2;
console.log(obj.a);
```

Output: 2
Explanation: `obj` and `obj2` reference the same object; mutation is visible through both.

---

47. Two chains with finally

```
Promise.resolve()
  .then(() => 'ok')
  .finally(() => console.log('done'))
  .then(v => console.log(v));
```

Output:

done
ok

Explanation: `finally` runs before the subsequent `.then`; it does not change the fulfillment value.

---

48. Default parameter with missing argument

```
function test(a, b = 2) { console.log(a + b); }

test();
```

Output: `NaN`
Explanation: a is `undefined`; `undefined + 2` is `NaN`.

---

49. Promise and immediate logs

```
async function test() { return 1; }
console.log(test());
```

Output: `Promise {<fulfilled>: 1}`
Explanation: `async` functions always return Promises; a plain return value becomes a fulfilled Promise.

---

50. Two immediate then-chains interleaving

```
Promise.resolve()
  .then(() => console.log('A1'))
  .then(() => console.log('A2'));

Promise.resolve()
```

```
  .then(() => console.log('B1'))
  .then(() => console.log('B2'));
```

Output:

```
A1
B1
A2
B2
```

Explanation: First microtasks from each chain run in registration order, then the second-level microtasks.

---

51. Returning object literal after return on new line

```
function foo() {
  return
  {
    x: 1
  }
}
console.log(foo());
```

Output: `undefined`
Explanation: Automatic semicolon insertion ends `return` before the object; nothing is returned.

---

52. `typeof null` vs `=== null`

```
const v = null;
console.log(typeof v);
console.log(v === null);
```

Output:

```
object
true
```

Explanation: `typeof null` is `'object'` (quirk); strict equality correctly detects `null`.

---

53. Equality of null/undefined only with ==

```
console.log(null == undefined);
console.log(null === undefined);
```

Output:

```
true
false
```

Explanation: Only loose equality equates `null` and `undefined`; strict equality does not.

---

54. Coercion with empty array

```
console.log([] + 1);
console.log(1 + []);
```

Output:

```
"1"
"1"
```

Explanation: `[]` coerces to `""` in string context; addition with a string produces a string.

---

55. Unary plus and booleans

```
console.log(+true);
console.log(+false);
```

Output:

```
1
0
```

Explanation: Unary `+` converts booleans to numbers: `true` → `1`, `false` → `0`.
