

Closure interview questions



give 10 hard output questions most frequently asked for closures and 5 medium

Perfect — here's a **curated list of the 15 most frequently asked closure output questions** in JavaScript interviews —

5 medium-level and **10 hard-level**, all focused on **closure behavior, scoping, async behavior, and variable capture**.

1 Medium-Level (5 Questions)



```
js

function makeCounter() {
  let count = 0;
  return function () {
    return ++count;
  };
}

const counter1 = makeCounter();
console.log(counter1());
console.log(counter1());
const counter2 = makeCounter();
console.log(counter2());
console.log(counter1());
```

Expected Output:

```
1
2
1
3
```

Each call to `makeCounter()` creates a new closure with its own `count` variable.



```
js

for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

Expected Output:

3
3
3

Because **var** is function-scoped, not block-scoped. The loop ends before **setTimeout** runs, so **i = 3**.

3

```
js

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

Expected Output:

0
1
2

let creates a **new lexical scope** for each iteration → closure captures the correct **i**.

4

```
js

function outer() {
  let a = 10;
  function inner() {
    console.log(a);
    a++;
  }
  return inner;
}

const fn = outer();
fn();
fn();
outer()();
```

Expected Output:

10
11
10

Each new call to **outer()** makes a new **a** (since closures preserve their own copy).

5

```
js

function test() {
  var x = 1;
  return function () {
    console.log(++x);
  };
}

var fn1 = test();
var fn2 = test();
fn1();
fn2();
fn1();
```

Expected Output:

```
2
2
3
```

Each closure created by `test()` has its own `x`.

Hard-Level (10 Questions)

6

```
js

function foo() {
  var a = 10;
  return function (b) {
    console.log(a + b);
  };
}

var x = foo();
x(5);
foo()(5);
```

Output:

```
15
15
```

Each closure remembers its own `a`, but both `a` are initialized to 10.

7

```
js

function outer() {
  let count = 0;
  return function inner() {
    if (count < 3) {
```

```

        console.log(++count);
        inner();
    };
}

const fn = outer();
fn();

```

Output:

```

1
2
3

```

Recursion inside closure still has access to the same **count** variable.

8

```

js

function x() {
  for (var i = 0; i < 3; i++) {
    (function (i) {
      setTimeout(() => console.log(i), i * 1000);
    })(i);
  }
}
x();

```

Output:

```

0
1
2

```

IIFE creates a new scope that preserves each **i**.

9

```

js

function foo() {
  var a = b = 0;
  a++;
  return a;
}

foo();
console.log(typeof a);
console.log(typeof b);

```

Output:

```
typescript
```

```
undefined
number
```

b becomes global (no `var`), **a** is local to `foo`.

10

```
js
```

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(add10(2)));
```

Output:

17

Inner closure adds `x` from its own scope to `y`.

11

```
js
```

```
let funcs = [];
for (var i = 0; i < 3; i++) {
  funcs.push(() => i);
}
console.log(funcs[0](), funcs[1](), funcs[2]());
```

Output:

3 3 3

All closures share same `i` from `var`.

12

```
js
```

```
let funcs = [];
for (let i = 0; i < 3; i++) {
  funcs.push(() => i);
```

```
}
console.log(funcs[0](), funcs[1](), funcs[2]());
```

Output:

```
0 1 2
```

Each **let** iteration gets its own scope.

1 3

```
js

function createBase(num) {
  return function (innerNum) {
    return num + innerNum;
  };
}

var addSix = createBase(6);
console.log(addSix(10));
console.log(addSix(21));
```

Output:

```
16
27
```

Classic closure — function “remembers” **num = 6**.

1 4

```
js

function counter() {
  let count = 0;
  return {
    inc: function () {
      count++;
      console.log(count);
    },
    dec: function () {
      count--;
      console.log(count);
    },
  };
}

const c1 = counter();
c1.inc();
c1.inc();
c1.dec();
```

Output:

```
1  
2  
1
```

`count` is private and shared by both `inc` and `dec`.

1 5

```
js
```

```
var a = 10;  
(function () {  
    console.log(a);  
    var a = 20;  
})();
```

Output:

```
javascript
```

```
undefined
```

`var a` is hoisted inside the IIFE, shadowing the outer `a`.

Would you like me to make the **same list as a VS Code “practice file”** (with questions + empty “Output:” lines so you can fill in answers yourself)?
That’s great for interview drilling.