Core JavaScript Mastery Guide

Perfect! Let's go **through 15+ tricky closure questions** in JavaScript, complete with **solutions and explanations** — these are **classic interview questions**.

1️⃣ Closure Inside a Loop (var vs let)

```js
for(var i = 1; i <= 3; i++){
  setTimeout(function(){
    console.log(i);
  }, 1000);
}
// Output: 4 4 4
```

Fix using IIFE:

```js
for(var i = 1; i <= 3; i++){
  (function(j){
    setTimeout(function(){ console.log(j); }, 1000);
  })(i);
}
// Output: 1 2 3
```

Fix using let:

```js
for(let i = 1; i <= 3; i++){
  setTimeout(() => console.log(i), 1000);
}
// Output: 1 2 3
```

2️⃣ Private Counter

```js
```

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

const c = createCounter();
c.increment(); c.increment();
console.log(c.get()); // 2
```

- `count` is **private**; closures preserve its value.

3️⃣ Function Factory

```
js

function multiplyBy(x) {
  return function(y){ return x*y; };
}
const double = multiplyBy(2);
console.log(double(5)); // 10
```

- Inner function **remembers** `x`.

4️⃣ Once Function

```
js

function once(fn) {
  let called = false;
  return function(...args){
    if(!called){ called=true; return fn(...args); }
  };
}
const init = once(() => console.log("Init"));
init(); // Init
init(); // nothing
```

- Closure preserves `called` flag.

5️⃣ Closure with IIFE (Module Pattern)

```js
const module = (function(){
  let secret = 0;
  return {
    increment: () => secret++,
    get: () => secret
  };
})();
module.increment();
console.log(module.get()); // 1
```

- IIFE + closure **encapsulates private data**.

6️⃣ Closure in Event Handler

```js
function setup() {
  let msg = "Hello";
  document.querySelector("button").addEventListener("click", function() {
    console.log(msg);
  });
}
setup();
```

- Even after `setup()` finishes, the handler **remembers** `msg`.

7️⃣ Loop with Async Closures

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

```
}
// Output: 0 1 2
```

- Closure ensures **correct value per iteration.**

8️⃣ Recursive Closure

```
js

function factorial(n){
  if(n <=1) return 1;
  return n * factorial(n-1);
}
```

- Named function expression can also be used:

```
js

const factorial = function fact(n){
  if(n<=1) return 1;
  return n*fact(n-1);
};
```

9️⃣ Closure with setTimeout & var

```
js

for(var i=1;i<=3;i++){
  setTimeout(function(){ console.log(i); },i*1000);
}
// Output: 4 4 4
```

- Fix using let:

```
js

for(let i=1;i<=3;i++){
  setTimeout(()=>console.log(i),i*1000);
}
// Output: 1 2 3
```

🔟 Closure for Caching / Memoization

```js
function memo(fn){
  const cache = {};
  return function(x){
    if(cache[x]!==undefined) return cache[x];
    return cache[x]=fn(x);
  };
}

const square = memo(x=>x*x);
console.log(square(5)); // 25
console.log(square(5)); // 25 (from cache)
```

- Closure **stores cache**.

1️⃣1️⃣ Closure with Private Array

```js
const listModule = (function(){
  let arr = [];
  return {
    add: item => arr.push(item),
    get: () => [...arr]
  };
})();
listModule.add("Anoop");
console.log(listModule.get()); // ["Anoop"]
```

- Keeps `arr` **private**.

1️⃣2️⃣ Multiple Closures Sharing Same Outer Variable

```js
```

```
function outer() {
  let x = 10;
  return [
    function(){ return x; },
    function(){ x++; return x; }
  ];
}
const [a,b] = outer();
console.log(a()); //10
console.log(b()); //11
console.log(a()); //11
```

- Both closures share **same lexical environment**.

1️⃣3️⃣ Closure in Callback Functions

```
js

function waitAndRun(msg){
  setTimeout(function(){ console.log(msg); },1000);
}
waitAndRun("Hello"); // "Hello"
```

- Closure preserves `msg`.

1️⃣4️⃣ Closure & Loop with Different Variables

```
js

for(let i=1;i<=3;i++){
  let j=i*2;
  setTimeout(()=>console.log(j),1000);
}
// Output: 2 4 6
```

- Block-scoped `let` ensures **unique closure per iteration**.

1️⃣5️⃣ Nested Closures

```js
function outer(a){
  return function middle(b){
    return function inner(c){
      return a+b+c;
    };
  };
}
console.log(outer(1)(2)(3)); // 6
```

- **Each function remembers outer variables.**

✅ Key Takeaways for Interviews

1. **Closure = function + lexical environment**
2. Common pitfalls: `var` in loops, async callbacks, memory leaks.
3. Use cases:
   - Private variables
   - Callbacks / Event handlers
   - Function factories / currying
   - Memoization / caching
   - Module pattern
4. Named function expressions + closures = recursion safe
5. Arrow functions also create closures (lexical `this`)

If you want, I can **make a "Closure Mastery Cheat Sheet"** with **all patterns, pitfalls, and interview tricks** in **one place** — perfect for last-minute prep.

Do you want me to do that next?

Is this conversation helpful so far?