Core JavaScript Mastery Guide

Perfect! Let's do a **deep dive into Closures in JavaScript** — one of the most important concepts for both advanced JS and interviews.

1️⃣ What is a Closure?

A **closure** is a function that **remembers the variables from its outer scope** even after the outer function has finished executing.

> In simple words: **a function + the environment in which it was created**.

2️⃣ Basic Example

```
js

function outer() {
  let count = 0; // outer variable
  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer(); // outer() executed, returns inner()
counter(); // 1
counter(); // 2
counter(); // 3
```

- `inner()` **remembers** `count` from `outer()` scope.
- This is a **closure**.

3️⃣ How Closures Work Internally

1. When a function is created, JS engine stores **Lexical Environment** (variables around it).

2. Even if the outer function finishes execution, **the inner function keeps a reference to the environment**.
3. Memory for outer variables **is preserved** as long as the inner function exists.

```js
function outer() {
  let a = 10;
  return function inner() {
    return a + 5;
  };
}
const fn = outer();
console.log(fn()); // 15
```

- `a` still exists because `fn` (inner function) is holding a reference to it.

4️⃣ Common Use Cases of Closures

(a) Data Encapsulation / Private Variables

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

const counter = createCounter();
counter.increment();
console.log(counter.get()); // 1
```

- `count` is private — cannot access directly from outside.

(b) Function Factories

```js
```

```
function multiplyBy(x) {
  return function(y) {
    return x * y;
  };
}

const double = multiplyBy(2);
console.log(double(5)); // 10

const triple = multiplyBy(3);
console.log(triple(5)); // 15
```

- Inner function **remembers** `x` from outer scope.

(c) Event Handlers / Callbacks

```
js

function setup() {
  let message = "Hello";
  document.querySelector("button").addEventListener("click", function() {
    alert(message);
  });
}
setup();
```

- Even after `setup()` finishes, the function inside `addEventListener` **retains access to** `message`.

5️⃣ Advanced Closure Patterns

(a) Loop + Closure Problem

```
js

for(var i=1; i<=3; i++){
  setTimeout(function(){
    console.log(i);
```

```
  }, 1000);
}
```

- Prints: `4 4 4` ✅ Why?
  - `var` is function-scoped → all functions share same `i`.
- Fix using closure:

```
js

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

- Or using `let` (block-scoped):

```
js

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

(b) Once Function / Memoization

```
js

function once(fn) {
  let called = false;
  return function(...args) {
    if(!called){
      called = true;
      return fn(...args);
    }
  };
}

const init = once(() => console.log("Initialized"));
```

```
init(); // Initialized
init(); // nothing
```

- Closure **stores** `called` between calls.

(c) Module Pattern

```
js

const counter = (function(){
  let count = 0;
  return {
    increment: function(){ count++; },
    decrement: function(){ count--; },
    get: function(){ return count; }
  };
})();
counter.increment();
console.log(counter.get()); // 1
```

- Classic pattern before ES6 modules.

6️⃣ Interview Tricks / Questions

1. **Closure inside loop**
- Always ask: `var` vs `let`.
2. **Private variables**

```
js

function secret(){
  let value = 42;
  return function(){ return value; };
}
```

- How to access `value`? Only via returned function.
3. **Memory Usage**
- Closures **keep variables alive** → useful but can cause memory leaks if overused.
4. **Callbacks inside closures**

```
js
```

```
function setup(x){
  return function(y){ return x + y; };
}
const add5 = setup(5);
console.log(add5(10)); // 15
```

✅ Summary

- **Closure = function + lexical environment**.
- Preserves **outer variables** even after outer function finishes.
- Useful for **private variables, function factories, callbacks, module patterns**.
- Common pitfalls: loops with `var`, memory leaks.

I can next move on to **Advanced Objects + Prototype Chain + Inheritance** because closures often combine with objects in real-world scenarios.

Do you want me to go **objects → prototypes → inheritance** next?

Absolutely! Let's go **deeper into closures** with **common interview-style questions**, solutions, and explanations. I'll make sure each is clear so you can **apply it anywhere**.

1️⃣ Closure inside a Loop

Problem:

```
js

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

Question: What will this print? Why?

Answer:

```
4
4
4
```

Explanation:

- `var` is **function-scoped**, not block-scoped.
- All three functions share the same `i` from outer scope.
- By the time `setTimeout` executes, the loop is done (`i = 4`).

Fix 1: Use IIFE

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

- `j` creates a new copy for each iteration → closure preserves it.

Fix 2: Use `let`

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

- `let` is **block-scoped**, each iteration gets its own `i`.

2️⃣ Private Counter using Closure

Problem: Create a counter with **increment**, **decrement**, and **getValue** methods, but the counter value should **not be accessible directly**.

Solution:

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

  return {
    increment: function() { count++; },
    decrement: function() { count--; },
    getValue: function() { return count; }
  };
}

const counter = createCounter();
counter.increment();
counter.increment();
counter.decrement();
console.log(counter.getValue()); // 1
console.log(counter.count); // undefined
```

Explanation:

- `count` is in **outer function's scope**.
- Returned object's methods form closures → can access `count`.
- Outside code cannot access `count` directly → **encapsulation achieved**.

3️⃣ Function Factory using Closure

Problem: Create a `multiplyBy` function that returns a function to multiply any number by a fixed value.

Solution:

```js
function multiplyBy(x){
  return function(y){
```

```
    return x * y;
  };
}

const double = multiplyBy(2);
const triple = multiplyBy(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Explanation:

- Inner function remembers `x` from outer function.
- Multiple closures can exist simultaneously with different `x` values.

4️⃣ Once Function (Execute Only Once)

Problem: Create a function `once` that **runs a function only once**, no matter how many times it's called.

Solution:

```
js

function once(fn) {
  let called = false;

  return function(...args){
    if(!called){
      called = true;
      return fn(...args);
    }
  };
}

const init = once(() => console.log("Initialized"));
init(); // Initialized
init(); // nothing
```

Explanation:

- `called` is preserved in closure → inner function remembers it.

- Ensures `fn` executes only once.

5️⃣ Closure and SetTimeout – Common Trap

Problem: Why this code prints `undefined`?

```
js

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

Answer:

- All functions share the same `i`.
- After loop ends, `i = 3`. So prints:

```
3
3
3
```

Fix using Closure:

```
js

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

- Now prints: `0 1 2` at 1-second intervals.

6️⃣ Closure in IIFE (Module Pattern)

Problem: Create a **module** for adding/removing items from an array with private access.

```js
const shoppingList = (function(){
  let items = []; // private

  return {
    add: function(item){ items.push(item); },
    remove: function(item){ items = items.filter(i => i !== item); },
    getList: function(){ return [...items]; }
  };
})();

shoppingList.add("Milk");
shoppingList.add("Eggs");
shoppingList.remove("Milk");
console.log(shoppingList.getList()); // ["Eggs"]
console.log(shoppingList.items); // undefined
```

Explanation:

- `items` is **private** due to closure.
- Module pattern uses IIFE + closure to **encapsulate state**.

✅ Key Interview Notes for Closures

1. Closures **remember outer variables** after outer function executes.
2. Common use cases:
   - Private variables
   - Callbacks / async functions
   - Function factories
   - Module pattern
   - Memoization / caching
3. Pitfalls:
   - Loops with `var`
   - Memory leaks if references are not cleared
4. Arrow functions also create closures.

I can next make a **full set of 15–20 tricky closure questions** with solutions that are **commonly asked in interviews**, including loops, callbacks, recursion, and modules.

Do you want me to prepare that next?