Closures and lexical scoping

Absolutely! Let's dive deep into **Closures** and **Lexical Scoping** in JavaScript — two foundational (and powerful) concepts.

🔍 What is Lexical Scoping?

Lexical scoping means that a function's scope is determined by its **location within the source code**, and **nested functions have access to variables declared in their outer scope**.

✅ In JavaScript, **scope is created by functions and blocks (`{}`)**.

🧠 Example

```
function outer() {
  const outerVar = "I'm from outer";

  function inner() {
    console.log(outerVar); // ✅ Has access to outerVar
  }

  inner();
}

outer();
```

> The function inner() can access the variable outerVar because it's lexically scoped inside outer().

🧩 What is a Closure?

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

In other words, **a closure gives you access to an outer function's variables even after the outer function has returned.**

🔧 Basic Closure Example

```
function outer() {
  const outerVar = "I'm still here!";

  return function inner() {
    console.log(outerVar); // This forms a closure
  };
}

const closureFunc = outer(); // outer() is called and returns inner()
closureFunc(); // Logs: I'm still here!
```

- Even though `outer()` has finished execution, `inner()` still **remembers** `outerVar`.
- That **remembering** is a **closure**.

🧪 Real-Life Analogy

Think of a **closure** like a backpack 🎒:

- When a function is created, it **packs up** all variables it might need from its surrounding scope.
- Even if you move that function somewhere else, it keeps that backpack of variables.

🧱 Practical Use Cases of Closures

✅ 1. Data Privacy / Encapsulation

You can hide variables from the outside world using closures.

```
function counter() {
  let count = 0;
  return function () {
    count++;
    console.log(count);
  };
}

const increment = counter();
increment(); // 1
increment(); // 2
```

- `count` is **private** — cannot be accessed directly outside `counter()`.

✅ 2. Function Factories

Closures let you create functions with preset behavior.

```
function multiplier(factor) {
  return function (num) {
    return num * factor;
  };
}

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

console.log(double(4)); // 8
console.log(triple(4)); // 12
```

Each inner function **remembers** the `factor` variable from its outer lexical scope.

✅ 3. Maintaining State in Asynchronous Code

Closures are used in asynchronous loops:

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

Output:

```
1
2
3
```

Without the closure (`function(j)`), all timeouts would log `4` because of how `var` scopes.

⚠️ Common Mistakes with Closures

❌ Accidentally sharing state

```
const counters = [];

for (var i = 0; i < 3; i++) {
  counters.push(function () {
    console.log(i);
  });
}

counters[0](); // 3 (not 0)
counters[1](); // 3
counters[2](); // 3
```

All functions share the same `i`.

✅ Fix using `let` (block scoped) or closure:

```
for (let i = 0; i < 3; i++) {
  counters.push(function () {
    console.log(i);
  });
}
```

🔄 Summary

Concept	Description
Lexical Scope	Functions are scoped based on where they’re written in code
Closure	A function remembers variables from its outer scope, even after that scope has returned
Use Cases	Data hiding, currying, async logic, custom factories

Let me know if you'd like visual diagrams, or want to continue to the next topic like **IIFE** or **Function Currying**!