Episode 12: Crazy JS Interview → ft. Closures

Q1: What is a Closure in JavaScript?

Answer:

A **closure** is formed when a function remembers its **lexical environment** even after it's executed.
In simple terms:

> A Closure = Function + Reference to its Outer Scope

Example:

```
function outer() {
  var a = 10;
  function inner() {
    console.log(a); // 👈 has access to a
  }
  return inner;
}
outer()(); // Output: 10
```

➡️ First `()` returns `inner`, second `()` executes it.

Q2: Does this still form a closure?

```
function outer() {
  function inner() {
    console.log(a);
  }
  var a = 10;
  return inner;
}
outer()(); // Output: 10
```

Answer:

Yes! Closure forms irrespective of the variable declaration order.
The inner function still has access to `a` via **lexical scoping**.

Q3: If we change `var` to `let`, will behavior change?

Answer:

Nope! `let` is also block-scoped and works the same in this context.

📌 Q4: Will inner function have access to outer function's argument?

```
function outer(b) {
  let a = 10;
  function inner() {
    console.log(a, b);
  }
  return inner;
}
outer("Hello There")(); // Output: 10 "Hello There"
```

☑ Answer:

Yes, closures capture both **local variables** and **function parameters**.

📌 Q5: Will `inner` form closure with `outest`?

```
function outest() {
  var c = 20;
  function outer(b) {
    let a = 10;
    function inner() {
      console.log(a, c, b);
    }
    return inner;
  }
  return outer;
}
outest()("Hello There")(); // Output: 10 20 "Hello There"
```

☑ Answer:

Absolutely! 🔐 `inner` forms closure with `outer` and `outest`.

📌 Q6: What's the output of the below and why?

```
function outest() {
  var c = 20;
  function outer(b) {
    let a = 10;
    function inner() {
      console.log(a, c, b);
```

```
    }
    return inner;
  }
  return outer;
}
let a = 100;
outest()("Hello There")(); // Output: 10 20 "Hello There"
```

☑ Answer:

Even though there's another `a = 100`, the closure captures the **closest scoped `a` (inside `outer`)**.
If `a` wasn't found in inner scopes, it would search outward.

📌 Q7: 🔥 Advantages of Closures

- ☑ **Module Design Pattern**
- ☑ **Currying**
- ☑ **Memoization**
- ☑ **Data hiding & encapsulation**
- ☑ **Async programming like setTimeout**

📌 Q8: Data Hiding & Encapsulation Example

❌ Without Closure:

```
var count = 0;
function increment() {
  count++;
}
console.log(count); // Accessible globally ❗
```

☑ With Closure:

```
function counter() {
  var count = 0;
  return function increment() {
    count++;
    console.log(count);
  };
}
const counter1 = counter();
counter1(); // 1
```

🔐 `count` is private here. Cannot be accessed from outside!

🛠️ Want to add `decrement`? Refactor with constructor:

```
function Counter() {
  var count = 0;

  this.incrementCounter = function() {
    count++;
    console.log(count);
  };

  this.decrementCounter = function() {
    count--;
    console.log(count);
  };
}

const counter1 = new Counter();
counter1.incrementCounter(); // 1
counter1.incrementCounter(); // 2
counter1.decrementCounter(); // 1
```

📌 Q9: ❗ Disadvantages of Closures

> Closures can **overuse memory** if not handled well.

💡 Why?

Because variables in closures are not garbage-collected until the closure is gone.

📦 Example:

```
function a() {
  var x = 0;
  return function b() {
    console.log(x); // x is retained
  };
}
const y = a();
y(); // x not garbage collected
```

🧹 JavaScript engines like V8 are smart!
Unused variables (e.g., `z = 10`) in closures may still be garbage collected if not referenced.

🧠 Summary

Closures = Function + Lexical Scope
They're **powerful** for encapsulation, async patterns, and performance optimizations.
But use with care to avoid memory issues!
