JavaScript Closures Cheat Sheet

---

1. What is a Closure?

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

Key Points: - Created whenever a function is defined inside another function. - Inner function “closes over” variables from the outer function. - Useful for **data privacy**, **maintaining state**, or **functional programming**.

---

2. Basic Example

```
function outer() {
    let message = "Hello Closure";

    function inner() {
        console.log(message); // Accesses outer scope
    }

    return inner;
}

const fn = outer();
fn(); // Output: Hello Closure
```

---

3. Example with Counter

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

    return {
        increment() {
            count++;
        },
        getValue() {
            return count;
        }
    };
}
```

```
const counter = Counter();
console.log(counter.getValue()); // 0
counter.increment();
counter.increment();
console.log(counter.getValue()); // 2
```

Explanation: - `count` is private. - Only `increment()` and `getValue()` can access it — thanks to closure.

---

4. Closure with Function Factories

```
function makeMultiplier(x) {
    return function(y) {
        return x * y; // inner function remembers x
    }
}

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

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

Explanation: - Each returned function remembers the value of `x` that was passed when it was created.

---

5. Why Use Closures?

1. **Data Privacy** – private variables.
2. **Stateful Functions** – maintain internal state.
3. **Function Factories** – generate customized functions.
4. **Callbacks & Event Handlers** – remember variables when the callback runs later.

---

6. Visual Representation

```
Outer function scope: { message: "Hello Closure" }
Inner function (closure) -> has access to Outer scope variables
Even after outer function execution ends, inner retains access
```

---

7. Key Takeaways

- Closures are everywhere in JS — callbacks, modules, IIFEs, etc.
- They allow **encapsulation** and **persistent state**.
- Think of closures as **functions with memory**.