

Explain closures in JS



explain this :

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(add10(2)));
```

it is confusing to me

Perfect this is a **great example of closures** in JavaScript — and it's completely normal that it feels confusing at first.

Let's go step-by-step very slowly

Code

```
js

function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(add10(2)));
```

Step 1: Understanding `makeAdder`

```
js

function makeAdder(x) {
  return function (y) {
    return x + y;
}
```

- This function **returns another function**.
- The returned inner function uses the variable **x** from the outer function.
- That means — when you call `makeAdder`, it **remembers** the value of **x** even after the outer function has finished executing.

That's what we call a **closure** —

"A closure is a function that remembers the variables from the place where it was created."

Step 2: Creating add5 and add10

```
js

var add5 = makeAdder(5);
var add10 = makeAdder(10);
```

Now let's see what each variable stores:

- **add5** becomes:

```
js

function (y) {
    return 5 + y;
}
```

- **add10** becomes:

```
js

function (y) {
    return 10 + y;
}
```

So:

- **add5(2)** → returns $5 + 2 = 7$
- **add10(2)** → returns $10 + 2 = 12$

Step 3: Understanding the final line

```
js

console.log(add5(add10(2)));
```

Let's evaluate step by step 

Step 3.1 — Evaluate inner part first:

```
js

add10(2)
```

→ from above, **add10(2) = 10 + 2 = 12**

So now the line becomes:

```
js
console.log(add5(12));
```

Step 3.2 — Now evaluate add5(12)

add5 adds 5 to its argument:

$$\text{add5}(12) = 5 + 12 = 17$$

✓ Final Output:

17

🔍 Recap of the flow

Step	Expression	Result
makeAdder(5)	returns a function <code>(y) => 5 + y</code>	stored in add5
makeAdder(10)	returns a function <code>(y) => 10 + y</code>	stored in add10
add10(2)	$10 + 2$	12
add5(12)	$5 + 12$	17

💡 Real-life analogy

Think of **makeAdder** like a **factory** that makes “adding machines”:

- **makeAdder(5)** → a machine that always adds 5
- **makeAdder(10)** → a machine that always adds 10

Now you’re chaining them:

“Take the number 2, first put it in the +10 machine (gives 12), then put that result into the +5 machine (gives 17).”

Would you like me to show this same concept visually (like how closures “remember” variables in memory)?



yes

Perfect — let’s visualize how this works in memory 🔎

Step 1: When the code starts

Memory (environment) is empty.

Step 2: Define the function

```
js

function makeAdder(x) {
  return function (y) {
    return x + y;
}
}
```

In memory:

```
sql

Global Memory:
makeAdder → <function definition>
```

Step 3: Call makeAdder(5)

```
js

var add5 = makeAdder(5);
```

When you call `makeAdder(5)`:

1. A new **execution context** (temporary box) is created for `makeAdder`.
2. Inside it, `x = 5`.
3. `makeAdder` returns a new function `(y) => x + y`.
4. That returned function **remembers** `x = 5` from the outer scope, even after `makeAdder` finishes.

Now memory looks like this:

```
javascript

Global Memory:
makeAdder → <function>
add5 → function (y) { return x + y } with closure { x: 5 }
```

That "with closure { x: 5 }" means this function carries a small backpack  holding `x = 5`.

Step 4: Call makeAdder(10)

```
js
var add10 = makeAdder(10);
```

Same thing again:

- New **x = 10**
- A new inner function is returned that **remembers** this **x = 10**

Memory now:

```
javascript
Global Memory:
makeAdder → <function>
add5 → function (y) { return x + y } with closure { x: 5 }
add10 → function (y) { return x + y } with closure { x: 10 }
```

Step 5: Execute add5(add10(2))

Let's do it visually, step-by-step 

Inner call first: add10(2)

1. JS calls the function stored in **add10**.

2. Inside it:

- **y = 2**
- It looks inside its closure and finds **x = 10**
- Returns **10 + 2 = 12**

 Closure for **add10**: { **x: 10** }

Result = 12

Outer call: add5(12)

Now this runs:

- **add5** has closure { **x: 5** }
- **y = 12**
- It returns **x + y = 5 + 12 = 17**

 Final Output = 17

Visualization Summary

```
yaml  
  
makeAdder  
└─ add5 → (y) => 5 + y [closure: {x:5}]  
└─ add10 → (y) => 10 + y [closure: {x:10}]  
  
Execution:  
add10(2) → 12  
add5(12) → 17
```

🧩 What's a closure again?

A **closure** is like a small “backpack 🎒” that a function carries.

It keeps a reference to the variables from its original scope, **even if that scope has finished executing.**

Would you like me to draw this as a **visual diagram** (like boxes showing memory and closures)? It will make this even clearer.



yes

