JavaScript - Day 8

CLOSURE :-

→ FUNCTIONS ARE FIRST CLASS CITIZENS IN JS.

→ ① We can assign functions to variables.

e.g

```
var sayHello = function () { }
```

② We can pass functions as arguments

e.g

```
function sayHello (message) {
    message ();
}
```

↳ Accepting function as an argument.

```
sayHello (function () {
    console.log ("Hello");
});
```

⇒ Passing function as an argument.

③ Return functions from another function

```
function sayHello() {
    return function message() {
        console.log ("Hey");
    }
}
sayHello()();
```

① Has the return value of sayHello function

② The returned value is a function and we want to call that function that is returned.

We are able to achieve all this because, we functions are just objects. right? And we can pass objects as arguments, return them etc.

* HIGHER ORDER FUNCTIONS :-

→ A function that takes a function as an argument, return a function, ~~as an argum~~ or both.

→ Let's say we want to multiply 2 numbers, but before that we want to check if they are numbers

```
function multiply (num1, num2) {
  if (num1 && num2 && typeof num1 === 'number'
      && typeof num2 === 'number')
    {
      return actualMultiplication (num1, num2)
    }
}
```

→ So basically instead of doing everything in multip we first check if arguments are valid and then call the multiplication method.

→ Now let's say we decides that we want to add both, the numbers, will we write another function doing the same check? can we pass add or multiply as argument? [assume add & multiply exists].

```
function operation (num1, num2, performOperation)
{
  if(checkIfValid (num1, num2)) {
      performOperation (num1, num2);
  }
}
operation (5, 2, multiply); // we can now pass a func
operation (5, 2, add);
```

Basically checks if valid.

as argument to decides what we want to do at runtime.

* CLOSURES :-

→ Closures allow functions to access variables from the enclosing scope even after it ~~leves~~ leaves the scope in which it was declared.

Confusing right?

```
function first() {
    let breakfast = 'Breakfast';
    return function second() {
        let lunch = 'Lunch';
        let random = 'xyz';
        return function third() {
            let dinner = 'dinner';
            return `${breakfast} ⇒ ${lunch} ⇒
                ${dinner}`;
        }
    }
}

first()()();
```

output >> "Breakfast" ⇒ lunch ⇒ dinner

① ⇒ When you called first() ⇒ it returned second.
→ First got pushed on the call stack and when it returned the result, it got poped off the stack.

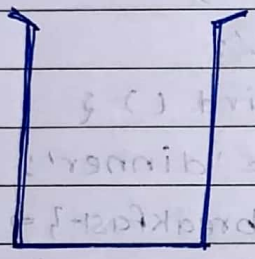
② ⇒ We invoked the result of first() [which is a function] by first()() so second got invoked;
⇒ second was pushed on the call stack and returned function three. So second is popped off the stack.

③ → We finally invoked function three by first ()()()

→ function third printed breakfast, lunch & dinner But wait!!! How did function third get access to breakfast and lunch when the functions owing them were popped off the stack?

→ well, the magic lies in closures.

→ closure is actually a feature by Javascript.

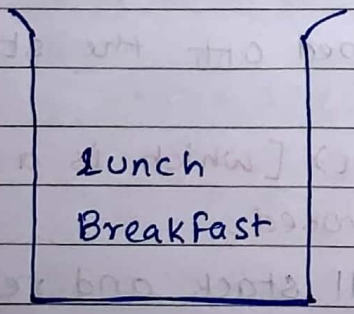


→ Let's say we have a closure box somewhere, when JE sees that breakfast in accessed, somewhere down the line, it put's breakfast into the closure box.

→ Even though first is done executing, it keeps variable breakfast in the closure box.

→ similarly it sees lunch is being accessed later, and puts it in closure box.

→ function third will now get access to breakfast and launch from closure box.



→ Well, we also declared a variable called random inside second.

Why is it not in the closure box?

Becaused it's not being refereneed.

→ later and can be cleaned up the garbage collector :)

- **So what enables this Feature?**
  1. Functions are first class citizens.
     functions can be returned from another funct!
  2. Lexical scope.
     What variables we have access to depends on where the function was declared.

→ So using the concept of higher order functions and scope chaining, we can enables closures.

* **NOTE**
  1. First and second are both higher order functions as they bothe return functions.
  2. Before we actually executes the code, the JE knows what variables your code has access to.