Closure

Created	@May 21, 2021 1:10 PM
Class	
Type	
Materials	
Reviewed	☐

What is a closure ?

Closure is when a function "remembers" it's lexical scope even when the function is executed outside that lexical scope.

Let's say we take an example

```
function do(task) {
  setTimeout(function fun() {
    console.log(task);
  }, 100);
}
do("Please do something!");
```

At the time instance when function `fun` runs, the function `do` has already been executed. The variable `question` which it is closed over, should have gone away. But it didn't right because closure preserved it. So any language that gives us the functionality of lexical scoping and first class functions should have closures, because what will we do with a function that lexically access some variables and then we pass it somewhere it forgets everything.

Another example of closure:

```
function do(task) {
  return function fun() {
    console.log(task);
  }
}
```

```
var myTask = do("Do something");
myTask();
```

Here, `myTask` executes later, but function `do` remembers the variables in its closure.

Note:

Closure is not about snapshotting a value at any point of time. We always close over variables and not close over values.

```
var name = "Sanket Singh";
var myName = function print() {
  console.log(name);
}

name = "Sarthak";

myName(); // It will print Sarthak
```

The above code is the simplest example of why you should never consider closure as some snapshotting of values or closing over values of preserving values. It has nothing to do with values.

Let's take another example:

```
for(var i = 1; i <= 3; i++) {
  setTimeout(function print() {
    console.log(`i: ${i}`);
  }, i*1000);
}
// i: 4
// i: 4
// i: 4
```

Here is another scenario when we are not closing over the value rather than closing on a variable (or to be precise bucket of the variable. Why ? We will see that now)

So can we never close over values ?

No we can't, and if we want to then we need to make new variable every time when we expect a change in the variable's value. So to achieve this we can either initialise a variable `j` of type `let` and assign value of `i` into it, so in every iteration it creates a new bucket for storing the value, and closures can close over these buckets.

```
for(var i = 1; i <= 3; i++) {
  let j = i;
  setTimeout(function print() {
    console.log(`j: ${j}`);
  }, i*1000);
}
// j: 1
// j: 2
// j: 3
```

So here what is happening is `let j = i` is going to work in the block scope of the function and creates a new bucket that can store the value i. So from es6 JS included a new pattern in the language so that we can achieve the same functionality as above by just make the loop variable initialised by let.

```
for(let i = 1; i <= 3; i++) {
  setTimeout(function print() {
    console.log(`i: ${i}`);
  }, i*1000);
}
// i: 1
// 2: 2
// 3: 3
```

So here every time the loop runs a new variable bucket of label `i` is created. But we are still not closing over it's value rather than we are closing over the bucket in the memory.

So closure is preservation of the linkage to the variable in memory not capturing of it's value.