With closure, we can solve this problem. It was not working because i was refering to the same memory location [console.log(i) → to i in for loop]

So, we have to do something that i is supplied with a new value everytime.

```
function x() {
    for (var i=1; i<=5; i++) {
        function closure(x) {
            setTimeout(function() {
                console.log(x)
            }, x * 1000
        }
        closure(i)
    }
}
```

We can use any name instead of x.

Here, there is a new copy of i everytime and so, we can achieve the desired output.

Episode - 12

JS Interview Questions ft. Closures

→ Key Points:

1) An inner function can be directly called using two parenthesis ()().

→ Even parameters can be passed this way (Reminder that the function needs to be returned to do this)

e.g.
```
function outer() {
    var a = 10
    function inner() {
        console.log(a)
    }
    return inner
}
```

outer()()

→ This is same as:

```
var close = outer()
close()
```

⇒ 10

(First outer is called, then inner and returned)

→ let declarations and function parameters are also closed over in closures.

→ If we use let a = 10, same result

→ For function parameter, the parameter is also included in a closure.

e.g.
```
function outer(b) {
    function inner() {
        console.log(a, b)
    }
```

```
    let a = 10
    return inner
}

Outer('Khushi')()
```

⟹ 10 Khushi

* If we write return function inner() instead of return inner at the end, in that case we have to define the variable before the return function statement.

```
    return function inner() {
        console.log(a, b, c)
    }

    var a = 10
    let b = 20
    const c = 30
}
```

⟹ undefined for var

Reference Error: Cannot access 'b' / 'c' before initialization.

↓

For let and const

→ If we had, return inner at the end after declaration, no error or undefined.

2) Closures can be used for data hiding and encapsulation. So other code cannot access the variable.

e.g.
```
var counter = 0

function increment() {
    counter++
}
```

Here anybody can access the counter variable.

To ensure nobody can modify the counter, we can use closures.

e.g.
```
function counter() {
    var count = 0
    return function increment() {
        count++
        console.log(count)
    }
}

var counter1 = counter()
counter1()
```

⇒ 1

If I do, console.log(counter) it will throw error, showing it is not defined. i.e. data privacy is provided.

→ Function constructors in JS

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

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

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

var counter1 = new Counter()
```

→ we have to use new keyword while using constructors.

```
counter1.increment()
counter1.increment()
counter1.decrement()
```

⟹
1
2
1

→ This is more good and scalable code than previous one. If we had both increment and decrement, then using function constructors is a good way.

Miracle
Pg.No :
Date: / /20

3) Disadvantages of Closures:

→ Memory Consumption
→ Memory Leaks
⋮

4) Garbage Collector:

→ Unused variables are automatically deleted in High Level Programming Language (JS) by garbage collector.

→ Closures allocate a lot of memory which cannot be deleted so this acts as a disadvantage.

→ The primary purpose is to prevent memory leaks.

→ Some browsers now have smart garbage collectors that automatically delete variables that are not used outside closures.

→ Memory leaks: A piece of memory that is no longer being used or required by an application but for some reason it is still in the memory.
Forgotten data forever waiting to be used - In simple words.

Miracle
Pg No :
Date: / /20

Episode - 13

First Class Functions ft. Anonymous Functions

1) Function Statement / Function Declaration

→ A normal function that we create using naming convention.

e.g
```
function a() {
    console.log("a called")
}
```

2) Function Expression

→ When we assign a function to a variable.

e.g.
```
var b = function() {
    console.log("b called")
}
```

⇒ Another difference is hoisting. a() is hoisted as a function and var b as a variable.

So, if I call b() before creating it, it will throw error.

a() will be run without any error.

```
a()
b()

function a(){
    :
}

var b = function(){
    :
}
```

⟹ a called
TypeError: b is not a function

3) Anonymous Function

→ A function without a name. It is used where functions are treated as value.

```
var c = function(){
    :
}
```

In function statement, we cannot use anonymous functions.

```
function(){
    :
}
```

is invalid. Syntax Error: Function statements require a function name.

4) Named Function Expression

→ A function expression with a name.

e.g
```
var a = function xyz() {
    console.log("a called")
}

a()
xyz()
```

⇒ a called
ReferenceError: xyz is not defined

→ xyz is not a global scope but created as a local variable.

```
var a = function xyz() {
    console.log(xyz)
}
```
→ we can also access xyz inside a function. (it is local variable)

```
a()
```

⇒
```
f xyz() {
    console.log(xyz)
}
```

5) Difference between Parameters and Arguments

→ Parameter : Variable in function definition

Miracle
Pg.No :
Date: / /20

Argument : Value passed during function invocation.

parameter

e.g. function a(param1, param2) {
console.log(param1, param2)
}

argument

a(4,5)

=> 4 5

→ We can also pass functions as arguments

a(function() {
:
})

OR

function b() {
:
}

a(b)

(Only 1 parameter is passed for example)

6) First Class Functions / First Class Citizens

→ Functions are treated like any other variables.

- Can be passed as argument
- Can be assigned to other variable
- Can be returned from functions

→ FCF: The ability to use functions as value is called FCF.

Miracle
Pg No :
Date: / /20

→ Assigned to variable

e.g. var a = function (param) {
Returned ← return function xyz() {
from another console.log("FCF")
function } }

a(function () {
:
} → Passed as an argument

→ When we use let, const instead of var, it behaves the same way.