

#14 video

CRAZY JS Interview ft. closure

What is closure in JavaScript?

A function along with a reference with outer environment together form a closure

In other words you can say that closure is a combination of a function and its lexical scope bundled together forms closure.

can you explain little bit more about it?

It's like each and every function in JS has access to outer level lexical environment that mean like access to the variable & function which is present in the environment of its parents. So it has each and every function have access to that so even when this function is like executed in some other scope, not in original scope but even it is executed in some other scope it still remember — (remember its outer lexical environment where it is originally present in the code. That is what closure is.

```

function outer() {
  var c = 50;
  function inner(p) {
    let a = 10;
    console.log('now called "inner"');
  }
  return inner;
}
  
```


→

```
function outer() {
  function inner() {
    console.log(a);
  }

```

```
  let a = 10;

```

```
  return inner;
}
```

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

we can replace by

// outer()();

both syntax are valid
& meaning is same.O/P → 10

→ No difference using let instead of var

→ function outer(b) {

```
  let a = 10;

```

```
  function inner() {

```

```
    console.log(a, b);

```

```
  }

```

```
  return inner;
}
```

var close = outer("Hello world");

close();

* inner() will have
access of parameters
& function, variable
of outer();(i.e. will be included
in closure)

→

function outest() {

```
  var c = 20;

```

```
  function outer(b) {

```

```
    let a = 10;

```

```
    function inner() {

```

```
      console.log(a, b, c);

```

```
    }

```

```
    return inner;
  }

```

```
  return outer("hello world");
}
```


outer() will be called with "Hello world" as argument and will return inner()

```
var close = outest()("Hello world");  
close();
```

in close → inner function will be assigned
o/p - 10 "Hello world" 20

```
function outest() {  
  var c = 20;  
  function outer(b) {  
    function inner(c) {  
      console.log(a, b, c);  
    }  
  }  
}
```

```
  let a = 10;  
  return inner;  
}
```

```
return outer;  
}
```

O/P
10 "hello world" 20
however if a in outer is removed then output will be

100 "hello world" 20
[a is present in global execution context & will be accessed through scope chain - resolution.]

```
let a = 100;  
var close = outest()("hello world");  
close();
```

* If a is removed from both sites, then it will throw ReferenceError. a is not defined.

Advantages of JS

Earlier mentioned

1. Data hiding and encapsulation

function counter() {

var count = 0;

return function incrementCounter() {

count++;
console.log(count);

}

}

var count1 = counter();

count1;

count1;

O/P

1
2

Count can be accessed through count1. But if we try to access count variable directly then it will give Error.

→ var count = 0;

function incrementCounter() {

count++;

}

Now count is available for all function and operation (public variable).

So data hiding & encapsulation is possible through closure.

function counter() {

return function incrementCount() {

count++;

console.log(count);

}

}

var count1 = counter();] separate closure & EC

count1();

count1();

var count2 = counter();] separate closure

count2();

count2();

O/P

1

2

1

2

count1

count2

→ function counter() {

var count = 0;

this.counterIncrement = function() {

count++;

console.log(count);

}

this.counterDecrement = function() {

count--;

console.log(count);

}

var count1 = new counter(); // constructor

// hence new

keyword is

required

count1.counterIncrement();

count1.counterIncrement();

count1.counterDecrement();

O/P

1

2

1

Disadvantage

- Variable do not garbage collected Hence over consumption of memory.
- If not handled properly then it will lead to memory leak.

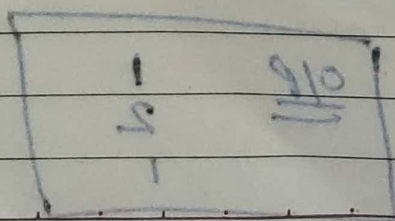
Javascript is a high level language. It is upto programmer how memory should be used. A garbage collector is a program which frees the variable which are no longer needed.

* Relation between closure & Garbage collection

```
function a() {
  var x = 100;
  return function b() {
    console.log(x);
  }
}
var y = a();
y();
```

→ variable x will not be garbage collected.

V8 engine in chrome browser (V8) smartly does the task of garbage collection.




```
function a(x) {
  var x = 10;
  var z = 10;
  return function b(y) {
    console.log(x);
  };
  var y = 9(2);
  y();
}
```

Although `z` is part of closure still it will be garbage collected

```
console.log(z) → 10
console.log(z) →
```

Reference Error:
`z` is not defined

```
function a() {
  var p = function() {
    console.log("a called");
  };
  p();
  console.log("a called");
}
a();
```

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

```
function a() {
  var p = function() {
    console.log("a called");
  };
  p();
  console.log("a called");
}
a();
```

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

Difference in hoisting in memory
of execution context exact copy of `a()` is created as `a()` is complete function and hence it is hoisted.