



## Episode 12 : Famous Interview Questions ft. Closures

### Q1: What is Closure in Javascript?

**Ans:** A function along with reference to its outer environment together forms a closure. Or in other words, A Closure is a combination of a function and its lexical scope bundled together. eg:

```
function outer() {  
  var a = 10;  
  function inner() {  
    console.log(a);  
  } // inner forms a closure with outer  
  return inner;  
}  
outer(); // 10 // over here first `()` will return inner function and then using  
second `()` to call inner function
```

### Q2: Will the below code still forms a closure?

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

**Ans:** Yes, because inner function forms a closure with its outer environment so sequence doesn't matter.

### Q3: Changing var to let, will it make any difference?

```
function outer() {  
  let a = 10;  
  function inner() {  
    console.log(a);  
  }  
  return inner;  
}
```

```
}  
outer(); // 10
```

**Ans:** It will still behave the same way.

**Q4: Will inner function have the access to outer function argument?**

```
function outer(str) {  
  let a = 10;  
  function inner() {  
    console.log(a, str);  
  }  
  return inner;  
}  
outer('Hello There')(); // 10 "Hello There"
```

**Ans:** Inner function will now form closure and will have access to both a and str.

**Q5: In below code, will inner form closure with outest?**

```
function outest() {  
  var c = 20;  
  function outer(str) {  
    let a = 10;  
    function inner() {  
      console.log(a, c, str);  
    }  
    return inner;  
  }  
  return outer;  
}  
outest()('Hello There')(); // 10 20 "Hello There"
```

**Ans:** Yes, inner will have access to all its outer environment.

**Q6: Output of below code and explanation?**

```
function outest() {  
  var c = 20;  
  function outer(str) {  
    let a = 10;  
    function inner() {  
      console.log(a, c, str);  
    }  
  }  
}
```

```

    }
    return inner;
  }
  return outer;
}
let a = 100;
outest()('Hello There')(); // 10 20 "Hello There"

```

**Ans:** Still the same output, the inner function will have reference to inner a, so conflicting name won't matter here. If it wouldn't have find a inside outer function then it would have went more outer to find a and thus have printed 100. So, it try to resolve variable in scope chain and if a wouldn't have been found it would have given reference error.

## Q7: Advantage of Closure?

- Module Design Pattern
- Currying
- Memoize
- Data hiding and encapsulation
- setTimeouts etc.

## Q8: Discuss more on Data hiding and encapsulation?

```

// without closures
var count = 0;
function increment(){
  count++;
}
// in the above code, anyone can access count and change it.

-----

// (with closures) -> put everything into a function
function counter() {
  var count = 0;
  function increment(){
    count++;
  }
}
console.log(count); // this will give referenceError as count can't be accessed.
So now we are able to achieve hiding of data

-----

```

```

// (increment with function using closure) true function
function counter() {
  var count = 0;
  return function increment(){
    count++;
    console.log(count);
  }
}

var counter1 = counter(); // counter function has closure with count var.
counter1(); // increments counter

var counter2 = counter();
counter2(); // here counter2 is whole new copy of counter function and it won't
            // impact the output of counter1

*****

// Above code is not good and scalable for say, when you plan to implement
// decrement counter at a later stage.
// To address this issue, we use *constructors*

// Adding decrement counter and refactoring code:
function Counter() {
  // constructor function. Good coding would be to capitalize first letter of
  // constructor function.
  var count = 0;
  this.incrementCounter = function() { // anonymous function
    count++;
    console.log(count);
  }
  this.decrementCounter = function() {
    count--;
    console.log(count);
  }
}

var counter1 = new Counter(); // new keyword for constructor fun
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();
// returns 1 2 1

```

## Q9: Disadvantage of closure?

**Ans:** Overconsumption of memory when using closure as everytime as those closed over variables are not garbage collected till program expires. So when creating many closures, more memory is accumulated and this can create memory leaks if not handled.

**Garbage collector** : Program in JS engine or browser that frees up unused memory. In highlevel languages like C++ or JAVA, garbage collection is left to the programmer, but in JS engine its done implicitly.

```
function a() {  
  var x = 0;  
  return function b() {  
    console.log(x);  
  };  
}
```


```
var y = a(); // y is a copy of b()  
y();
```

// Once a() is called, its element x should be garbage collected ideally. But fun b has closure over var x. So mem of x cannot be freed. Like this if more closures formed, it becomes an issue. To tacke this, JS engines like v8 and Chrome have smart garbage collection mechanisms. Say we have var x = 0, z = 10 in above code. When console log happens, x is printed as 0 but z is removed automatically.

---

Watch Live On Youtube below:



 [Edit this page](#)