**CLOSURES**

function outer() {

  let a = 10;

  return function inner(b) {

    return a + b;

  };

}

let fn = outer();

let ans = fn(20);

console.log(ans);

**USE-CASE**

Closure is useful in hiding implementation detail in JavaScript. In other words, it can be useful to create private variables or functions.

let counter = (function () {

  let privateCounter = 0;

  function changeBy(val) {

    privateCounter += val;

  }

  return {

    increment: function () {

      changeBy(1);

    },

    decrement: function () {

      changeBy(-1);

    },

    value: function () {

      return privateCounter;

    },

  };

})();

alert(counter.value()); // 0

counter.increment();

counter.increment();

alert(counter.value()); // 2

counter.decrement();

alert(counter.value()); // 1

In the above example, increment(), decrement() and value() becomes public function because they are included in the return object, whereas changeBy() function becomes private function because it is not returned and only used internally by increment() and decrement().

**DEBOUNCING**

**THROTTLEING**

**CURRYING**

It is a technique in functional programming, transformation of the function of multiple arguments into several functions of a single argument in sequence.

The parent function takes the first provided argument and returns the function that takes the next argument and this keeps on repeating till the number of arguments ends. Hopefully, the function that receives the last argument returns the expected result.

function volumeOfCube(length, breadth, height) {

  return length \* breadth \* height;

}

let ans = volumeOfCube(4, 5, 6);

console.log(ans); // 120

function volumeOfCube(length) {

  return function (breadth) {

    return function (height) {

      return length \* breadth \* height;

    };

  };

}

let ans = volumeOfCube(4)(5)(6);

console.log(ans); // 120

**HOISTING**

In JavaScript, variable and function names can be used before declaring it. The JavaScript compiler moves all the declarations of variables and functions at the top so that there will not be any error. This is called hoisting.

* Declarations are Hoisted
* Variables defined with let and const are hoisted to the top of the block, but not initialized.

carName = "Volvo";------- ReferenceError  
let carName;

carName = "Volvo";";------- SyntaxError  
const carName;

* Initializations are Not Hoisted
* JavaScript compiler moves the function definition at the top in the same way as variable declaration.
* JavaScript compiler does not move function expression.
* Functions definition moves first before variables.

alert(Sum(5, 5)); // 10

function Sum(val1, val2)----------function definition---------OK

{

return val1 + val2;

}

Add(5, 5); // Type error

var Add = function Sum(val1, val2)-----function expression-----Error

{

return val1 + val2;

}

**Differences between var, let, and const**

|  |  |  |
| --- | --- | --- |
| var | let | const |
| functional scope. | block scope. | block scope. |
| It can be updated and re-declared into the scope. | It can be updated but cannot be re-declared into the scope. | It cannot be updated or re-declared into the scope. |
| It can be declared without initialization. | It can be declared without initialization. | It cannot be declared without initialization. |
| It can be accessed without initialization as its default value is “undefined”. | It cannot be accessed without initialization otherwise it will give ‘referenceError’. | It cannot be accessed without initialization, as it cannot be declared without initialization. |
| hoisting done, with initializing as ‘default’ value | Hoisting is done , but not initialized (this is the reason for the error when we access the let variable before declaration/initialization | Hoisting is done, but not initialized (this is the reason for error when we access the const variable before declaration/initialization |

**PROMISES**

**ARROW FUNCTIONS, THIS, CALL, APPLY, BIND**

**EVENT LOOP**