Functions

Functions are the building blocks of javascript. A function is a set of statements that performs the task or calculates a value.

```
function greet()
{
    console.log('Hello World!');
}
greet() // Hello World!
```

Code Reusability

Functions add code reusability e.g.

```
function greet(name)
{
    console.log(`Hello ${name}!`);
}
greet('John'); // Hello John!
greet('Marry'); // Hello Marry!
```

Types of Functions

Non-Returning Function

```
function add(a, b )
{
    console.log(a+b);
}
add(10,5); // 15
```

Returning Function

```
function add(a, b )
{
    return a + b ;
}

var a = add(10, 5 );
console.log(a); //15
```

Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

```
console.log(animal);//ReferenceError: Cannot access 'animal' before initialization
let animal = 'bear';

console.log(animal);//ReferenceError: Cannot access 'animal' before initialization
const animal = 'bear';

console.log(animal);//undefined
var animal = 'bear';
```

Javascript has two methods two define functions.

```
Function Declaration
function walk()
{
    console.log('walk');
}
```

Function declaration can be hoisted i.e. we can call that function before it is defined.

```
Anonymous Function Expression
let run = function()
{
    console.log('run');
}

Named Function Expression
let run = function running()
{
    console.log('run');
}
```

In many situations we are forced to use function expressions such as when we are creating callbacks.

```
var add = function(a, b )
{
    return a + b ;
}
```

Function expression can’t be hoisted. Variable add can be hoisted but not function add().

```
walk(); // walk

function walk()
{
    console.log('walk');
}

sum(4,5); // ReferenceError: Cannot access 'sum' before initialization
const sum = function(a, b)
{
    console.log(a + b);
}

sum(4,5); // ReferenceError: Cannot access 'sum' before initialization
let sum = function(a, b)
{
    console.log(a + b);
}

sum(4,5); // sum is not a function
var sum = function(a, b)
{
    console.log(a + b);
}

console.log(sum); // undefined
var sum = function(a, b)
{
    console.log(a + b);
}
```

1. Variables declared with let and const cannot be hoisted.
2. Function expressions cannot be hoisted.
3. Function declarations can be hoisted.
4. Function expressions using var can be hoisted. Variable can be hoisted but the function can not be hoisted.

Callback Function

A callback function is a function passed into another function as an argument which is then invoked inside the outer function.

```
function outer(fx)
{
    fx();
}

function hello( )
{
    console.log('Hello world!');
}

outer(hello) ; // hello here is a callback function.
```

```
const addition = (a, b) => a + b

const subtraction = (a, b) => a - b

const multiplication = (a, b) => a * b

const divison = (a, b) => a / b

const calculate = (a, b, cb) => cb(a, b)

let result = calculate(4, 5, (a, b)=> a + b); //9

result = calculate(5, 6, addition); //11

result = calculate(36, 7, subtraction); //29

result = calculate(5, 6, multiplication); //30

result = calculate(48, 6, divison); //8
```

Function Scope

Function scope is like variable visibility. The location where a variable is defined dictates where we have access to that variable.

Block Scope

```
function lol()
{
    let x = 2;
    console.log(x);
}
lol();
console.log(x); // Error
```

```
let x = 3;
function lol()
{
    let x = 2;
    console.log(x);
}
lol();// 2
console.log(x); // 3
```

Variables declared with let can be modified from within a narrow scope.This can be useful but also dangerous.

```
let x = 3;
function lol()
{
    x = 2;
    console.log(x);
}
lol();// 2
console.log(x); // 2


if(true)
{
    let animal = 'eel';
}

console.log(animal);// error: animal is undefined because animal has block scope.

var i = 10;
for( var i = 0 ; i < 3 ; i++ )
{

}
console.log(i); // 3
```

Lexical Scope

Lexical scope is an important concept in JavaScript that determines the accessibility and visibility of variables and functions during the runtime of a program. It defines the region in your code where a variable or function is accessible. In simpler terms, lexical scope refers to the set of variables and functions that are accessible from a particular point in your code.

In JavaScript, scope is defined by the location of the variable or function declaration in the source code, rather than where it is called during the program's execution. This concept is also known as "static scoping" or "lexical scoping."

```
var g = "global scope";
console.log('1 > ', g);

function outer()
{
    var o = "outer scope";
    console.log('2 > ',g, o);

    function inner()
    {
        var i = "inner scope";
        console.log('3 > ',g, o, i);
    }
    inner();
}
outer();
```

A variable defined outside the function can be accessed inside that function if it is defined after the variable declaration but the opposite is not true. The variables defined inside a function will not be accessible outside that function.

```
function outer()
{
    var num = 10 ;

    function inner ()
    {
        console.log(`num is ${num}`);
    }
    inner();
}
outer();

console.log(`num is ${num}`); //error
```

Function Hoisting Revised

```
console.log(animal); //undefined
var animal = 'Lion';
```

Javascript hoists up the variable declaration.

```
var animal;
console.log(animal) ;
animal = 'Lion';
```

It does not reorganizes the code but it executes `var animal;` this statement first. This behavior is called hoisting.

Variable declarations with let or const are not hoisted.

```
console.log(animal); // can’t be hoisted
let animal = 'Lion';
```

Function Declarations

Function declaration can be hoisted i.e. we can call that function before it is defined.

```
hello();
function hello()
{
    console.log("Hello World !");
}
```

Function Expressions

```
hello(); // error
var hello = function()
{
    console.log("Hello World !");
}

console.log(hello); // OK
var hello = function()
{
    console.log("Hello World !");
}
```

Function expression can’t be hoisted. Variable hello can be hoisted but not function hello().

Arrow Functions

It is an alternative way of defining function expressions.

Message Functions

```
const hello = () =>
{
    console.log("Hello World !");
}
hello();
```

Argument Functions

```
const add = (x, y) =>
{
    return x + y;
}
console.log(add(5, 10));
```

Implicit Return in Argument Functions

```
const add = (x,y) => x + y;
console.log(add(10,5));
```

Arguments and Rest Parameter

```
var sum = function(a, b)
{
    console.log(a + b);
}

sum();// undefined + undefined = NaN
sum(1);//1 + undefined = NaN
sum(1, 2)// 3
sum(1, 2, 3, 4, 5);//3

var sum = function()
{
    console.log(arguments); // { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }
}

sum(1, 2, 3, 4, 5);
```

Rest parameter is the last formal parameter.

```
var sum = function()
{
    [...args] = arguments;
    const total = args.reduce((a, b)=> a+ b);
    console.log(total);
}

sum(1, 2, 3, 4, 5); //15
```

Default Arguments

```
var sum = function(a, b = 4, c = 9)
{
    console.log(a + b + c );
}

sum(1); //14
sum(1, 2); //12
sum(1, 2, 5); //8
```