**7. Functions**

**1) Function Declaration vs. Expressions**:

**Function in JavaScript**:

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which we wish to call it.

**Define Function in JavaScript**:

In JavaScript there are two way to define a function.

1. Function declaration

**Example**:

function walk() {

console.log("walk");

}

walk(); //walk

1. Function Expression

**Example**: Anonymous function expression

let run = function() {

console.log("run");

};

run(); //run

**2) Hoisting**:

Hoisting is the process of moving function declarations to the top of the file. This is done automatically by the JavaScript engine, that is executing the code.

**Example**:

//valid

walk(); //run

function walk() {

console.log("walk");

}

**Example**:

//invalid

run(); //error

let run = function() {

console.log("run");

};

This statement is like

console.log(x);

let x = 10;

**3) Arguments**:

Arguments is an Array-like object accessible inside functions that contains the values of the arguments passed to that function.

**Note**:

“Array-like” means that arguments has a length property and properties indexed from zero, but it doesn't have Array's built-in methods like forEach and map.

**Example**:

function sum(a, b) {

return a + b;

}

console.log(sum(1, 2)); //3

console.log(sum(1, 2, 3, 4)); //3

console.log(sum(1)); //NaN -> 1 + undefined -> NaN

Every function in JavaScript have a special object called argument.

function sum(a, b) {

console.log(arguments);

return a + b;

}

console.log(sum(1, 2, 3, 4, 5));

**Output**:

/\*

Arguments(5) [1, 2, 3, 4, 5, callee: ƒ, Symbol(Symbol.iterator): ƒ]0: 11: 22: 33: 44: 5callee: ƒ sum(a, b)length: 5Symbol(Symbol.iterator): ƒ values()\_\_proto\_\_: Object

\*/

function sum() {

let total = 0;

for (let value of arguments) total += value;

return total;

}

console.log(sum(1, 2, 3, 4, 5)); //15

**4) The Rest Operator**:

Rest Operator is an improved way to handle function parameter, allowing us to more easily handle various input as parameters in a function. And it is aa actual array.

JavaScript has allowed a variable number of function parameters of a function but the problem is that it is not an array. It is an array like object. Therefore performing some operations on “arguments” will give an error.

Rest operator is added in ES2015 or ES6 which improved the ability to handle parameter.

**Example**:

function sum(...args) {

console.log(args);

}

console.log(sum(1, 2, 3, 4, 5)); //[1, 2, 3, 4, 5]

**Example**:

function sum(...args) {

return args.reduce((a, b) => a + b);

}

console.log(sum(1, 2, 3, 4, 5)); // 15

**Example**:

function sum(discount, ...prices) {

const total = prices.reduce((a, b) => a + b);

return total \* (1-discount);

}

console.log(sum(0.1, 20, 30)); //45

**Note**:

We must have to place the rest operator as the end argument as a function. If we place any argument after “rest operator” we will get error. For this reason this operator is called rest operator.

**Example**:

function sum(discount, ...prices, value) {

const total = prices.reduce((a, b) => a + b);

return total \* (1-discount);

}

console.log(sum(0.1, 20, 30));

/\*

Error: Uncaught SyntaxError: Rest parameter must be last formal parameter

\*/

**5) Default parameters**:

In JavaScript, function parameters default to [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined). However, it's often useful to set a different default value. This is where default parameters can help.

**Example**:

function interest(principal, rate, years){

return principal\*rate / 100\*years;

}

console.log(interest(10000, 3.5, 5)); //1750

We can use default value as

**Example**:

function interest(principal, rate, years) {

rate = rate || 3.5;

years = years || 5;

return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //1750

From ES6 we can write

**Example**:

function interest(principal, rate = 3.5, years = 5) {

return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //1750

**Note**:

If we give a default value of a parameter after that all value we have to give a default value otherwise we will get error.

**Example**:

function interest(principal, rate = 3.5, years) {

return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //NaN

**6) Getters and Setters**:

We can use getter method to access properties and setter method to change or mute them.

getters => access properties

getters => change (mute) them

**Example**:

const person = {

firstName: 'Mosh',

lastName: 'Hamedani',

get fullName(){

return `${person.firstName} ${person.lastName}`;

},

//value receive here is a valid string

set fullName(value){

const parts = value.split(" ");

this.firstName = parts[0];

this.lastName = parts[1];

}

};

console.log(person.fullName); //Mosh Hamedani

person.fullName = 'John Smith'; // set "John Smith"

console.log(person.fullName); // John Smith

**7) Try and Catch**:

The try/catch/finally statement handles some or all of the errors that may occur in a block of code, while still running code. Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The finally statement lets you execute code, after try and catch, regardless of the result.

**Example**:

const person = {

firstName: 'Mosh',

lastName: 'Hamedani',

get fullName(){

return `${person.firstName} ${person.lastName}`;

},

//value receive here is a valid string

set fullName(value){

const parts = value.split(" ");

this.firstName = parts[0];

this.lastName = parts[1];

}

};

console.log(person.fullName); //Mosh Hamedani

person.fullName = true;

*//Uncaught TypeError: value.split is not a function*

person.fullName = null;

*//* *Uncaught TypeError: Cannot read property 'split' of null*

For resolved this error we should add try-catch in the above program.

**Example**:

const person = {

firstName: "Mosh",

lastName: "Hamedani",

get fullName() {

return `${person.firstName} ${person.lastName}`;

},

//value receive here is a valid string

set fullName(value) {

//if(typeof value !== 'string') return

if (typeof value !== "string") {

throw new Error("Value is not a string");

}

const parts = value.split(" ");

this.firstName = parts[0];

this.lastName = parts[1];

}

};

try {

person.fullName = null;

} catch (e) {

console.log(e);

}

console.log(person);

/\*

Error: Value is not a string

at Object.set fullName [as fullName]

at index.js:20

{firstName: "Mosh", lastName: "Hamedani"}

\*/

**Example**:

Pass empty String as full name

const person = {

firstName: "Mosh",

lastName: "Hamedani",

get fullName() {

return `${person.firstName} ${person.lastName}`;

},

//value receive here is a valid string

set fullName(value) {

//if(typeof value !== 'string') return

if (typeof value !== "string") {

throw new Error("Value is not a string");

}

const parts = value.split(" ");

if(parts.length !== 2){

throw new Error('Enter a valid name');

}

this.firstName = parts[0];

this.lastName = parts[1];

}

};

try {

person.fullName = '';

} catch (e) {

console.log(e);

}

console.log(person);

/\*

Error: Enter a valid name

at Object.set fullName [as fullName]

{firstName: "Mosh", lastName: "Hamedani"}

\*/

**8) Local vs. Global Scope**:

Scope determines the accessibility (visibility) of variables. In JavaScript there are two types of scope.

1. Local Scope
2. Global Scope

JavaScript has function scope: Each function creates a new scope. Variables defined inside a function are not accessible (visible) from outside the function.

**Local Scope**:

Variables declared within a JavaScript function, become LOCAL to the function.

Local variables have Function scope: They can only be accessed from within the function.

**Example**:

// code here can NOT use carName

function myFunction() {

var carName = "Volvo";

// code here CAN use carName

}

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

**Global Scope**:

A variable declared outside a function, becomes GLOBAL. A global variable has global scope: All scripts and functions on a web page can access it.

**Example**:

var carName = "Volvo";

// code here can use carName

function myFunction() {

// code here can also use carName

}

**Note**:

In general we should avoid defining global variables or constants. That is considered bad practice. Global variable accessible everywhere, globally and each function can accidentally change their value, and this will lead all kind of bugs and issues in our programs.

**9) Let vs. Var vs. Const**:

When we declare a variable with var, its scope is not limited to the block in which it’s defined. It’s one of the weird things in JavaScript that we have had for a long time. Before ES6 var was the only way to declare variables and constants.

Starting from ES6, also called ES2015, now we have two new keywords let and const, to define different variables and constants. These two keywords create block scope variables.

But var creates function scoped variables.

When we use var, outside of a function, this creates a global variable and attaches that global variable to the window object in browser. So in browser we have this object with lots of properties and method.

It is bad to attach a window to a variable object. Because window object is something central, there is only one instance of the window object. When we are using a third party library and if it has a variable with the same name, that variable can override our variable.

**Example**: var

function myFunction(){

for(var i=0; i<5; i++){

console.log(i);

}

console.log(i); //for var i is accessible here

}

myFunction(); //print from 0 to 5

In contrast when we use the let keyword to define a global variable, that global variable is not attached to the window object.

**Example**: let

function myFunction(){

for(let i=0; i<5; i++){

console.log(i);

}

console.log(i); //for var i is not accessible here

}

myFunction(); //print from 0 to 4

"const" is similar to C or C++'s "const" only. Once you assign a value to the variable declared as `const`, you cannot assign some other value to the const variable.

**Note**:

When we define a function, this function is technically a global function and is added to the window object.

**Example**:

function sayHi(){

console.log('Hi');

}

/\*

Output:

window.sayHi()

Hi

\*/

**10) The This Keyword**:

In JavaScript the "this" keyword confused a lot of developer. The "this" keyword references the object that is executing the current function.

IF the function is part of an object we call that function a method. If the function is a method in an object, "this" references that object itself.

method -> object

IF the function is a regular function, which means it's not part of an object, "this" references the global object, which is the window object in browser and global in Node.

function -> global (window, global)

**Node**:

For JavaScript, any HTML DOM element is a node.

<div>

<span></span>

<form> .... </form>

</div>

For example, in the HTML snippet above, span and form are child nodes of div.

Node.js is just the name of a library. It was named Node.js for other different reasons.

**Document Object Model (DOM)**:

With the HTML DOM, JavaScript can access and change all the elements of an HTML document. When a web page is loaded, the browser creates a Document Object Model of the page. The HTML DOM model is constructed as a tree of Objects.

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

1. JavaScript can change all the HTML elements in the page
2. JavaScript can change all the HTML attributes in the page
3. JavaScript can change all the CSS styles in the page
4. JavaScript can remove existing HTML elements and attributes
5. JavaScript can add new HTML elements and attributes
6. JavaScript can react to all existing HTML events in the page
7. JavaScript can create new HTML events in the page



**Example-1 (method -> object)**:

const video ={

title: 'a',

play(){

console.log(this);

}

};

video.play(); //{title: "a", play: ƒ}

Here play() is a method of video object and "this" references the video object itself. By the same token we can add a method later in this object, and we will get the same result.

**Example-2 (method -> object)**:

const video ={

title: 'a',

play(){

console.log(this);

}

};

video.stop = function(){

console.log(this);

};

video.stop(); // {title: "a", play: ƒ, stop: ƒ}

/\*

{title: "a", play: ƒ, stop: ƒ}

play: ƒ play()

stop: ƒ ()

title: "a"

\_\_proto\_\_: Object

\*/

Once again we are going to see the video object. Because stop() is a method in video object.

**Example-3 (function -> global (window, global)**:

const video ={

title: 'a',

play(){

console.log(this);

}

};

function playVideo(){

console.log(this);

}

playVideo();

/\*

Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}

\*/

We see a global object which is Window in browser and global in Node.

Now the above function (Example-3) is a constructor function. We do some change in the function. By conversion the first letter of the function should be capital letter.

**Example-4 (function -> global (window, global)**:

function Video(title){

this.title = title;

console.log(this);

};

const v = new Video('b'); //Video {title: "b"}

If we call a function using the new operator, which is the case for constructor function, will reference a new empty object and set "this" in this constructor function to point to this empty object. So in the line "this.title = title;" we add the title property to the empty object.

Now we will see another example

**Example-5 (this inside anonymous function)**:

const video ={

title: 'a',

tags: ['a', 'b', 'c'],

showTags(){

this.tags.forEach(function(tag){

console.log(this.title, tag); //Line-6

});

}

};

video.showTags();

/\*

undefined "a"

undefined "b"

undefined "c"

\*/

Here “this.title” in line-6 is undefined. Now if we remove the title property the output is

const video ={

title: 'a',

tags: ['a', 'b', 'c'],

showTags(){

this.tags.forEach(function(tag){

console.log(this, tag); //Line-6

});

}

};

video.showTags();

/\*

Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …} "a"

Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …} "b"

Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …} "c"

\*/

Here Line-6 “this” reference to the window object. But why it happened? We are still in video object. Because here we are inside the call back function and this function is just a regular function. It is not method in a video object. The only method we have inside video is showTags(). Because this is a regular function it references to the global object.

So it is the global object that is executing the anonymous callback function. But how can we add title in the output with each tag? We have few difference solutions for this.

**Solution-1**:

const video ={

title: 'a',

tags: ['a', 'b', 'c'],

showTags(){

this.tags.forEach(function(tag){

console.log(this.title, tag);

}, this);

}

};

video.showTags();

/\*

a a

a b

a c

\*/

7. Functions