https://shorturl.at/kAHNV

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Functions**

**Passing parameters to functions**

**Prototype**

**Prototype chaining**

**JSON**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

===============================================

Function:-

===============================================

- a particular logic is called as a function.

- functions are used to reuse the logic.

- There are following type of functions

i) Named functions

ii)Anonymous functions

iii)Constructor functions

- we can pass various type of parameters to functions

i) Rest parameters

ii)Optional parameters

iii)Default parameters

i) Named functions

- the function with name is called as named functions

- Syntax

//defining a function

function function\_name(parameters)

{

//business logic

}

//calling a function

function\_name(parameters)

Eg01

Create a function

@fun\_one

fun\_one return "Hello...!"

function fun\_one(){

return `Hello...!`

}

console.log(fun\_one)

console.log(fun\_one())

Eg02

create a function

@fun\_one

fun\_one have three arguments

@arg1

@arg2

@arg3

fun\_one return above arguments

function fun\_one(arg1, arg2, arg3) {

//return arg1 + " <--> " + arg2 + " <--> " + arg3

return [arg1, arg2, arg3]

}

console.log(fun\_one(`Tea`, `Milk`, `Coffee`))

console.log(fun\_one(123, `abc`, true))

Eg03

fun\_one return fun\_two definition

fun\_two return `Good Morning...!`

print above message using fun\_one

function fun\_one() {

return fun\_two

}

function fun\_two() {

return `Good Morning...!`

}

console.log(fun\_one) //??

console.log(fun\_one()) //??

console.log(fun\_one()())//??

ii)Anonymous functions

- the function without name is called as anonymous functions

- these are also called as arrow functions or callback functions

- we can represent arrow functions by '=>' symbol.

- arrow functions introduced in ES6.

- arrow functions are more secure than name functions.

- arrow functions utilize heap memory effectively.

- Syntax

//defining anonymous function

let variable\_name = function(arguments){

//business logic

}

//defining arrow function

let variable\_name =(arguments)=>{

//business logic

}

//calling a function

variable\_name(arguments)

Eg01

create arrow function using following variables

@fun\_one

fun\_one return "I am from arrow function...!"

let fun\_one = () => {

return "I am from arrow function...!"

}

console.log(fun\_one())

Eg02

Create arrow function

@fun\_one

fun\_one have three arguments

@arg1,

@arg2,

@arg3

fun\_one prints above arguments

let fun\_one = (arg1, arg2, arg3) => {

console.log(arg1 + " <...> " + arg2 + " <...> " + arg3)

}

fun\_one(10, 20, 30)

fun\_one('Tea', 'Milk', 'Coffee')

Eg03

one arrow function return another arrow function

inner arrow function return 'Have a nice day...!'

let fun\_one = () => {

return () => {

return `Have a nice day...!`

}

}

console.log(fun\_one) //?

console.log(fun\_one()) //?

console.log(fun\_one()())//?

===============================================

Inputs and outputs

===============================================

prompt

let var\_one = document.getElementById(id).value

alert

document.getElementById(id).innerHTML = res

window.open(<filename.html>)

Assignment:-

Design a simple apple calculator

- accept numbers in input

- display output

- operations + - x /

- Apply appropriate styling

===============================================

Constructor Functions:-

===============================================

- Constructor function is used to create a class like structure.

- In constructor function everything(variables and functions)

start with 'this' keyword.

- We can create the object to the constructor using 'new' keyword.

//Eg01

function class\_one(){

this.wish = `Welcome to constructor functions`

}

let obj = new class\_one()

console.log(obj.wish)

//Eg02

function class\_one() {

this.sub\_one = 'Angular'

this.sub\_two = 'NodeJS'

this.sub\_three = "Mongodb"

}

let obj = new class\_one()

console.log(obj.sub\_one, obj.sub\_two,obj.sub\_three)

//Eg03

function class\_one() {

this.myWish = `Good Morning`

this.newWish = () => {

return 'Happy Birthday...!'

}

}

let obj = new class\_one()

console.log(obj.myWish, obj.newWish())

//Eg04

function class\_one() {

this.sub\_one = `Javascript`

this.wish = () => {

return `Welcome to ${this.sub\_one}`

}

}

let obj = new class\_one()

console.log(obj.wish())

//Eg05 Parameterised constructor

function class\_one(arg1, arg2, arg3){

this.sub\_one = arg1

this.sub\_two = arg2

this.sub\_three = arg3

}

let obj = new class\_one(1,'2',3)

console.log(obj.sub\_one, obj.sub\_two, obj.sub\_three)

===============================================

prototype

===============================================

- prototype is a property which adds members

dynamically to constructor functions

//Eg01

function class\_one() { }

class\_one.prototype.wish = `Welcome to Prototype`

let obj = new class\_one()

console.log(obj.wish)

//Eg02

class\_one.prototype.sub\_one = `Electronics`

class\_one.prototype.sub\_two = `Computers`

class\_one.prototype.sub\_three = `Electricals`

console.log(obj.sub\_one, obj.sub\_two, obj.sub\_three)

//Eg03

class\_one.prototype.myWish = () =>{

return `Good Morning`

}

console.log(obj.myWish())

===============================================

Prototype Chaining:-

===============================================

- Nesting one constructor prototype to another constructor

is called prototype chaining.

- Concept of inheritance is achieved with prototype chaining

//Eg01 Single inheritance

function class\_one() { }

class\_one.prototype.fun\_one = function () {

return "I am from Function one"

}

function class\_two() { }

class\_two.prototype = Object.create(class\_one.prototype)

class\_two.prototype.fun\_two = function () {

return "I am from Function two"

}

let obj = new class\_two()

console.log(obj.fun\_one())

console.log(obj.fun\_two())

//Eg02 Multilevel inheritance

function class\_one(){}

class\_one.prototype.fun\_one = function(){

return "I am from Function one"

}

function class\_two(){}

class\_two.prototype = Object.create(class\_one.prototype)

class\_two.prototype.fun\_two = function(){

return "I am from Function two"

}

function class\_three(){}

class\_three.prototype = Object.create(class\_two.prototype)

class\_three.prototype.fun\_three = function(){

return "I am from Function three"

}

let obj = new class\_three()

console.log(obj.fun\_one())

console.log(obj.fun\_two())

console.log(obj.fun\_three())

//Eg03 Heirarchical inheritance

function class\_one(){}

class\_one.prototype.fun\_one = function(){

return "I am from Function one"

}

function class\_two(){}

class\_two.prototype = Object.create(class\_one.prototype)

class\_two.prototype.fun\_two = function(){

return "I am from Function two"

}

function class\_three(){}

class\_three.prototype = Object.create(class\_one.prototype)

class\_three.prototype.fun\_three = function(){

return "I am from Function three"

}

let obj1 = new class\_two()

let obj2 = new class\_three()

console.log(obj1.fun\_one())

console.log(obj1.fun\_two())

console.log(obj2.fun\_one())

console.log(obj2.fun\_three())

//Function Overriding

function class\_one() { }

class\_one.prototype.dbFun = () => {

return "MySql data soon"

}

function class\_two() { }

class\_two.prototype = Object.create(class\_one.prototype)

class\_two.prototype.dbFun = () => {

return "MongoDB data soon"

}

let obj = new class\_two()

console.log(obj.dbFun())

===============================================

Passing parameters to functions

===============================================

1. Default Parameters

- It allows named parameters to be initialized with

default values if no value or undefined is passed.

function fun\_one(arg1 = 10, arg2 = `FullStack`, arg3 = "Angular"){

console.log(arg1, arg2, arg3)

}

fun\_one() //10 FullStack Angular

fun\_one(25) //25 Fullstack Angular

fun\_one(undefined, "MERN") //10 MERN Angular

fun\_one(undefined, 'undefined', undefined) //10 undefined Angular

fun\_one(null, null, null) //null null null

fun\_one(null, undefined, "MongoDB") //null Fullstack MongoDB

fun\_one(undefined, undefined, undefined) //10 FullStack Angular

2. Optional Parameters

- While calling a function no need to pass all arguments.

- We can keep a few arguments as Optional.

- Optional Parameters introduced in ES6.

function fun\_one(arg1, arg2, arg3) {

console.log(arg1, arg2, arg3)

}

fun\_one() //undefined undefined undefined

fun\_one("Hello\_1") //Hello\_1 undefined undefined

fun\_one(null, "Hello\_2") //null Hello\_2 undefined

fun\_one(10, 15.7, "MEAN") //10 15.7 MEAN

fun\_one(null, null, null) //null null null

3. Rest Parameters

- It is an improved way to handle function parameters.

- It allows us to represent indefinite number of arguments

as an array.

- Rest parameters represented by '...' ( ... is called as spred operator)

function fun\_one(...arg) {

console.log(arg)

}

fun\_one("Angular") //[ 'Angular' ]

fun\_one() //[]

fun\_one(`Angular`, `Fullstack`) //[ 'Angular', 'Fullstack' ]

fun\_one(undefined, null) //[ undefined, null ]

function fun\_one(arg1, arg2 = "Hello\_2", ...arg3) {

console.log(arg1, arg2, arg3)

}

fun\_one("Hello\_1") //Hello\_1 Hello\_2 []

fun\_one("Hello\_1", undefined, "Hello\_3") //Hello\_1 Hello\_2 [ 'Hello\_3' ]

fun\_one("Hello\_1", "Hello\_2", "Hello\_3", "Hello\_4") //Hello\_1 Hello\_2 [ 'Hello\_3', 'Hello\_4' ]

fun\_one(undefined, undefined, undefined) //undefined Hello\_2 [ undefined ]

fun\_one(undefined, undefined, [1, 2, 3], [4, 5, 6]) //undefined Hello\_2 [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]

fun\_one(undefined, undefined, [1, 2, 3], 4, 5, 6) //undefined Hello\_2 [ [ 1, 2, 3 ], 4, 5, 6 ]

fun\_one("Hello\_1", [1, 2, 3], 4, 5, 6) //Hello\_1 [ 1, 2, 3 ] [ 4, 5, 6 ]

//function fun\_one(...arg1, ...arg2){} //SyntaxError: Rest parameter must be last formal parameter

//function fun\_one(...arg1, arg2){} //SyntaxError: Rest parameter must be last formal parameter

===============================================

JSON

===============================================

- JSON stands for Java Script Object Notation

- JSON is used to share data over network.

- JSON is lightweight as compared to XML.

- Parsing of JSON is easy as compared to XML.

- Syntax

Object {}

Arrays []

Data Key and value pairs

keys and values are saparated by ':'

key value pairs are saparated by ','

//Eg01

let demo = {

'sub\_one': 'HTML',

"sub\_two": "CSS",

sub\_three: 'Bootstrap'

}

console.log(demo)

console.log(demo.sub\_one, demo.sub\_two, demo.sub\_three)

//Iterate JSON

for(let x in demo)

//console.log(x)

console.log(demo[x])

//Eg02

let demo = {

wish: `Welcome`,

myWish: () => {

return `JSON Sessions`

}

}

console.log(demo.wish, 'to', demo.myWish())

//freeze() seal() //GO WITH HTML

let obj = {

p\_id : 111

}

console.log(obj)

//add new data

obj.p\_name = "P\_one"

console.log(obj)

//modify data

obj.p\_id = 101

console.log(obj)

/\*//Lock object using freeze function

Object.freeze(obj)

console.log("After Freeze",obj)

//try to modify data

obj.p\_id = 100

console.log(obj) //no error no modification

//try to add new data

obj.p\_cost = 10000

console.log(obj) //no error no modification

//try to delete data

delete obj.p\_id

console.log(obj) //no error no modification

\*/

//Lock object using seal function

Object.seal(obj)

console.log("After Seal",obj)

//try to modify data

obj.p\_id = 100

console.log(obj) //modification success

//try to add new data

obj.p\_cost = 10000

console.log(obj) //no error no modification

//try to delete data

delete obj.p\_id

console.log(obj) //no error no modification

/\*

freeze()

-> Read Allowed

-> Update Denined <=============

-> Write Denined

-> Delete Denined

seal()

-> Read Allowed

-> Update Allowed <=============

-> Write Denined

-> Delete Denined

\*/

===============================================

defineProperty:-

defineProperties:-

===============================================

//defineProperty

let obj = {

p\_id: 111

}

console.log(obj)

Object.defineProperty(obj, "p\_name", { value: "P\_one", writable: false })

console.log(obj)

obj.p\_id = 101

obj.p\_name = "P\_ONE"

console.log(obj)

//defineProperties

let obj = {

p\_id: 111

}

console.log(obj)

Object.defineProperties(obj, {

p\_name: { value: 'P\_one', writable: true },

p\_cost: { value: 10000, writable: false }

})

console.log(obj)

obj.p\_id = 101

obj.p\_name = 'P\_ONE'

obj.p\_cost = 111111

console.log(obj)

===============================================

entries() -> object to array

fromEntries() -> array to object

===============================================

var obj1 = {

p\_id: 111,

p\_name: "p\_one",

p\_cost: 10000

}

console.log(obj1) //{p\_id: 111, p\_name: 'p\_one', p\_cost: 10000}

let arr1 = Object.entries(obj1)

console.log(arr1) //[["p\_id", 111],["p\_name", "p\_one"],["p\_cost", 10000]]

let arr2 = [["p\_id", 111], ["p\_name", "p\_one"], ["p\_cost", 10000]]

console.log(arr2) //[["p\_id", 111],["p\_name", "p\_one"],["p\_cost", 10000]]

let obj2 = Object.fromEntries(arr2)

console.log(obj2) //{p\_id: 111, p\_name: 'p\_one', p\_cost: 10000}

================================================================

================================================================