**Closures and Prototypal Inheritance**

**Functions are Objects: special object ->callable object function\_name.variable\_name=value;**

function one(){

return 1;

}

**one();**

const obj ={ ***ES6 – two()***

two:function()/**two(){**

return 2;

}

}

**obj.two();**

function three(){

return 3;

}

**three.call()**

**const four = new function(‘num’,’return num’);**

four(4);

Function\_name. (dot) -> has Methods and properties are available because it is a special object

Like data **Functions can be passed around like objects**

**Functions are first class citizens:**

**Return functions as values** from other functions

function b(){

**return function c(){**

console.log(‘bye’);

**}**

}

**b( )( );** (or) **var d= b(); d( )**

Functions can be passed **as arguments**

function a**(fn)**{

**fn();**

}

**a(function(){console.log(‘hi’)})**

Functions can be **assigned to variables**

var stuff = function(){}

Don’t initialize in loops

Try to use default parameters like a(var=10){return var}

**Higher order functions: that take functions as arguments or return functions**

**function authenticate(level){**

**let arr=[];**

**for(i=0;i<level;i++)**

**arr.push(i);**

**}**

**const giveAccess = (person) => console.log('access given to ' +person);**

**function letPersonLogin(person,fn){**

**if(person.level==='admin')**

**fn(50000);**

**else if(person.level ==='user')**

**fn(1000);**

**return giveAccess(person.name)**

**}**

**letPersonLogin({level:'admin', name:'msk'},authenticate);**

Exercise:

**const multiplyBy = (val)=> (num)=>console.log( val\*num); // multiplyBy(6)(4)**

**const multiplyByTwo = multiplyBy(2);**

**multiplyByTwo(4);**

**Closure:** (combination of **function and the lexical** environment from which it was declared)

Allows a function to access a variables from an enclosing scope / environment even after it leaves the scope from which it was declared.

**JavaScript engine keeps around anything that is being referenced by child function**

**const test = string => name1 = name=> name2= name2=> string + name +name2;**

let temp = test("hi");

temp = temp(' sample');

temp = temp(' test')

console.log(temp);

function a(){

let grandpa='grandpa'

**return function b(){**

let father='father' // not deleted

let random = 12345; //garbage collected

**return function c(){**

let son='son'

**return `${grandpa} > ${father} > ${son}`;**

}

}

}

**const innerFunction = (name) => (name1)=>(name2)=> name + name1 + name2;**

console.log(innerFunction('hi')(' best')(' happens to you'));

**Closures and Memory** : closures are memory efficient

function callMeMaybe() {

const callMe = 'Hi!';

setTimeout(function() { // a function call to web api

console.log(callMe); // it still has variable callme

}, 4000); // because of closure

}

callMeMaybe();

o/p hi (after 4000 ms of waiting)

const getHeavyDuty = heavyDuty2();

getHeavyDuty(699)

getHeavyDuty(699)

getHeavyDuty(699)

// but i dont want to pollute the global namespace..

function heavyDuty2() {

const bigArray = new Array(7000).fill('😄')

console.log('created Again!')

**return function(item) {**

**return bigArray[item]**

**}**

} just by adding return function closure is created.

function heavyDuty(item) {

const bigArray = new Array(7000).fill('😄')

console.log('created!');

return bigArray[item]

}

heavyDuty(699)

heavyDuty(699)

heavyDuty(699)

here the bigArray is created three times , but **by using closures the array is not created everytime when a call is made .**

**Closures and Encapsulation** : closures allow us to protect important functions or data through encapsulation

**setTimeout**(function(){ alert("Hello"); }, 3000);

The setTimeout() method calls a function or evaluates an expression after a specified number of milliseconds.

**setInterval**(function(){ alert("Hello"); }, 3000);

The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds).

**clearInterval()**

**clearTimeout()**

const makeNuclearButton = () => {

let timeWithoutDestruction = 0;

const passTime = () => timeWithoutDestruction++;

const totalPeaceTime = () => timeWithoutDestruction;

const launch = () => {

timeWithoutDestruction = -1;

return '💥';

}

**setInterval(passTime, 1000); // calls pastime() every 1000ms(1s)**

return {totalPeaceTime}// by returning only totalpeacetime , launch() is protected

}

const ww3 = makeNuclearButton();

ww3.totalPeaceTime()

Exercise: closure

Self-Invoking Functions

**(function () { /**/can take parameters

**var x = "Hello!!"; // I will invoke myself**

**})();** // can pass arguments

**Question : to print all the I values / array values**

const array = [1,2,3,4];

for(var i=0; i < array.length; i++) {

setTimeout(function(){

console.log('I am at index ' + i)

}, 3000)

}

o/p I am at index 4 (repeated 4 times)

**Solution : 1**

const array = [1,2,3,4];

for(let i=0; i < array.length; i++) {

setTimeout(function(){

console.log('I am at index ' + array[i])

}, 3000)

}

**Solution :2**

const array = [1,2,3,4];

for(var i=0; i < array.length; i++) {

(function(closureI) { //immediately invoked function wraps

setTimeout(function(){ //set timeout function

console.log('I am at index ' + array[closureI])

}, 3000)

})(i) // reference I so that it should not be removed

} // so passing i to parameter closure I

**Prototypal inheritance:** (inherit properties) Array object and function object inherit methods and properties from Object

**lizard.\_ \_ proto \_ \_ = dragon**

This is not the proper way to inherit. \_ \_ proto \_ \_ should never be used. Performance issue, so use

**let lizard = Object.create(dragon)**

**array.\_ \_ proto\_ \_ .\_ \_proto \_ \_**

**// parent – object |base object**

**\_ \_ (to go one place top of the chain)**

**Array.toString();**

In Browser console

const array = [] (enter)

**array.\_ \_ proto\_ \_** (enter)

**array[] - > constructor**

Same goes with function

To inherit properties of one object into another

dragon = {};

lizard = {};

**lizard.\_ \_ proto \_ \_ = dragon** // inherit properties of dragon (Base) into lizard

**(baseObject).isPrototypeOf(inheritedObject) //** to check whether obj is inherited from another obj

It doesn’t copy the properties to the derived object, it just goes up the chain to reference

Scope chain and prototype chain are different

**for ( let prop in lizard) {**

**Console.log(prop);**

**}**

**Prints all the properties**

**for ( let prop in lizard) {**

**if(lizard.hasownproperty(prop){**

**console.log(prop)**

**} // prints lizards prop**

**}**

// by using **\_ \_proto\_ \_ as property** which points to prototype of base objects(up the chain) . even the Object (top most) has \_ \_ proto \_ \_ property pointing to null . Array.prototype and (the new array created say array )array.\_ \_ proto\_ \_ points to the same base Object . **typeOf {}(object) 🡪 functions** . Object is a function

Only functions have prototype property.Every function has a prototype property and it references to an obj used to attach properites that will be inherited further down the prototype chain.

**Exercise**

**create your own .bind() method using call or apply.**

Function.prototype.bind = function(whoIsCallingMe){

const self = this;

return function(){

return self.apply(whoIsCallingMe, arguments);

};

}

//Array.map() => to print '🗺'

Array.prototype.map = function() {

arr = [];

for (let i = 0; i < this.length; i++) {

arr.push((this[i]+'🗺'));

}

return arr;

}

console.log([1,2,3].map())

**Change built in objects**

//Date object => to have new method .lastYear() which shows you last year 'YYYY' format.

**Date.prototype.lastYear = function(){**

**return this.getFullYear() -1;// existing function**

**}**

new Date('1900-10-10').lastYear()//'1899'

new Date().lastYear()//'2018'

this method wont work with arrow functions because it is lexically scoped(this current ())