Advance Javascript

1. **Difference between window and this?**

Window is the main JavaScript object root, aka the global object in a browser, also can be treated as the root of the document object model. You can access it as window

Since *window* is the global object you can reference any properties of it with just the property name

window == this //true

**window** is global object

**this** refer to global object

In NodeJS we don’t have window object, window object only refer to browser.

1. Difference between window and document object?

window is javascript global object

document is an object which is part of window object which holds the information about page element, page title, page size etc..

we can refer document object directly with out referring window object



1. **Scene behand Hoisting in Javascript?**

Javascript engine creates a memory space for all global variables and function by setting variable value to “undefined”.

It’s a miss understanding that Hoisting means javascript moves all functions and variables to the top of execution, but behand the scene it won’t move any code to the top of execution but it set the execution context by setting default value to the variable called undefined.

1. **Undefined, null, “”?**

undefined is a special keyword, undefined is the default value set by the javascript engine if we forget to set any value to our variable manually or at run time.

var a;

console.log(a) // undefined

console.log(b) // b is not defined

variable “a” is in memory with default “undefined” value

variable “b” is not exist in memory so it gives an error that “b is not defined”

we can set value “undefined” to our variable

var c= undefined

console.log(c) // undefined

1. Why javascript is called Single Threaded, synchronous?

Single Threaded means one command will be executed at a time.

Synchronous Execution, code will be executed line by line or command by command.

So javascript is called single threaded synchronous execution by behaviour.

Example:-

function a(){

console.log(“I am funcation a”)

}

function b(){

console.log(“I am funcation b”)

}

function c(){

console.log(“I am funcation c”)

}

a();

console.log(“hi I am outside function”)

output in console is:-

I am funcation a

hi I am outside function

So this example proves javascript is single threaded and synchronous execution by behaviour, when a() function is invoked or called execution is moved to function a() body and execute the function a() commands then it came back to function a() invoke line and move to next command to print console.log(“hi I am outside function”)

Single threaded in above Example:- Javascript single controle which moves from function call ie a() to function body of function a() and same control came back to execute console.log(“hi I am outside function”), if javascript is multithreaded one control will move to a() body and another sill execute the console.log(“hi I am outside function”) then the output will be.

output in console is:-

hi I am outside function

I am funcation a

synchronous in above Example:- synchronous means one at a time, Javascript is single threaded language ie same control is responsible to execute our code flow.

In above example after function a() is called control moves to function a() body and executes all the commands in function a() body even if it takes couple of seconds and come back to print console.log(“hi I am outside function”)

1. What is function invocation?

Calling a function or executing a function is called function invocation.

1. **“let” and “var” difference?**
2. Hoisting

console.log(b) // undefined  
var b;   
  
console.log(a) // Uncaught ReferenceError: Cannot access 'a' before initialization  
let a;

ie Hoisting will happen for “var” but not “let”

1. Block scope

if(2>1){  
 var num = 10;  
}  
console.log(num) // 10

if(2>1){  
 let num1 = 10;  
}  
console.log(num1) // Uncaught ReferenceError: num is not defined  
  
ie “let” is block level scope we can not access let name outside any block might be if, for or function etc…

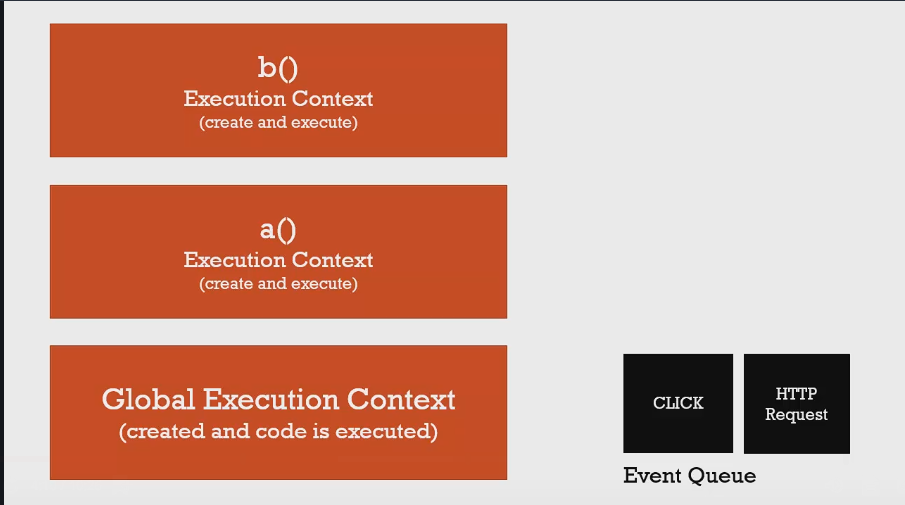
1. JS Asynchronous callback?

Javascript engine creates 2 stack

1. Execution stack
2. Event Queue

Execution stack holds “Global execution context” like global array, variables, function calls etc…  
javascript execute all the Global context ie memory will be created for all variables, all function will be executed if it is invoked at run time(calling function on page load) then javascript engine moves to Event queue stack if any event trigger by the user.

1. Event Queue, which holds the stack of events trigged ie if any click, drag, touchend etc…  
   javascript engine executes Event Queue once “Execution stack” is empty.



Example:-

// long running function

function waitThreeSeconds() {

// var ms = 3000 + new Date().getTime();  
 //while (new Date() < ms){}

for(var i=0;i<5000000000;i++){}  
 console.log('finished function');

}

function clickHandler() {  
 console.log('click event!');   
}

// listen for the click event  
document.addEventListener('click', clickHandler);  
waitThreeSeconds();  
console.log('finished execution');  
  
//output  
finished function  
finished execution  
4 click event!  
  
so in above example in side waitThreeSeconds() we are delaying the execution by using a loop, in between if user fires an event by click on document you can check the output that 4 events is mounted on Event Queue and javascript engine address this event after the Global context is executed completely.

Types and Operators

Javascript is Dynamic Typing language ie value of the variable changes on fly without any error ie   
  
var num =10;  
num = “bangalore”;  
console.log(num) // Bangalore  
  
Premitive types

1. undefined
2. null
3. string
4. boolean
5. symbol(new in ES6)
6. number

**Operators** are special functions, generally takes two parameters and return one results.

Example:- javascript behand the scene for + operator

var a = 3+4;  
console.log(a) // 7  
  
+(3,4);  
function +(3,4){  
 return 7; //predefined code by JS engine to add two numbers  
}

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

**Coercion:**

Converting a value from one type to another.

var a = 1 + “2”;  
console.log(a) // 12  
console.log(typeof a) //string  
  
so plus operator function convert result to number if both inputs are number, if any one of the input is string it convert both inputs to string and result is string.

function +(1,”2”){  
 return “1”+”2”;

}

**Weired parts:**

Example 1:-

console.log(1 < 2 < 3) //true  
step1 (1 < 2) return true  
step2 (true < 3) return true

Example2:-

console.log(3 < 2 < 1) //true  
step1 (3 < 2) return false  
step2 (false < 1) return true ie(0 < 1)

Number Coercion methods:-  
Number(2); //return number 2  
Number(“2”); //return number 2  
Number(true); //return number 1  
Number(undefined); //return NaN  
Number(null); //return number 0  
Number(“”); //return number 0  
Number("ddd"); //return NaN

Double Equals:-

3 == "3" //return true because it coercion number to string  
1 == true //return true  
0 == "" //return true  
0 == null //return false **weired**   
1 > null //return true **weired**null ==undefined //return true **weired**[1,2] == '1,2' //return true convert array to string

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness>

Boolean Coercion methods:-

var a;

if(a){

alert("hi")

}else{  
 alert(“hellow”)  
}

//output   
hellow  
  
so if method works like this  
if(a) converted to if(Boolean(a)) and a is undefined so Boolean(undefined) is false

Boolean(undefined) // return false  
Boolean(null) //return false  
Boolean(“”) //return false  
Boolean(“hi”) //return true  
Boolean(“1”) //return true  
Boolean(0) //return false  
Boolean(NaN) //return false

**|| (or) operator:-**(val1 || val2) or operator takes two parameter and return the true value  
  
(1 || 0) //return true value ie 1  
if(1 || 0){} // return 1 so its in if condition if(Boolena(1)){} //go inside true block condition

"hi" || "hellow" //return hi because jpreceding start from left to right when it finds true it return the true value  
  
undefined || "working" //return working  
undefined || null //return null  
0 || 0 //return 0  
2 || 1 //return 2  
undefined || undefined //return undefined  
undefined || "dddd" //return dddd  
null || “dddd” //return dddd  
“” || “dddd” // return dddd

Object and function

Object is the collection of key value pair, value can be Boolean, another object, method.

We can create JS object using new Object() constructor or object literal({})  
  
Ex:-

var person = new Object()

person.firstname = "Mohan";

person["lastname"] = "kumar";

//person.address.street = "dddd" //error Cannot set property 'street' of undefined

person.address = new Object()

person.address.street = "7th";

person.address.city = "bangalore"  
  
console.log(person) // return object  
console.log(person["address"]["street"]) //7th

Why error  
person.address.street = "dddd";  
dot operator function takes 2 parameter one is object called “person” and another is string called “address”.

Dot operator function check for address property in person object not found so it returns “undefined”.

**undefined.street** throw an error called // error Cannot set property 'street' of undefined

**working code:**  
person.address.street = "7th";  
  
dot operator takes 2 parameter both are object called person and address.  
So it return object({}).  
{}.street = “7th”;  
  
push street property to return object.

**1. Diffrence between JSON and Object literal?**JSON:- Javascript object notation. Strict syntax  
syntax:-   
var person= {“firstname” : ”mohan”, “pincode”:560086, “married”:true}

Object literal:-  
syntax:-   
var person= {firstname : ”mohan”, pincode:560086, married:true}

So when ever we send data to backend we need to send JSON not object literal, converting Objct literal to JSON is JSON.Stringify(person)  
  
converting JSON to Object literal is JSON.parse(person)

Example:-

JSON.stringify({firstname : "mohan", pincode:560086, married:true})

// return string "{"firstname":"mohan","pincode":560086,"married":true}"

JSON.parse('{"firstname":"mohan","pincode":560086,"married":true}')  
//return [object]

**Building Objects**

In javascript we can create objects using   
 1. Function constructors  
 2. Using object literal  
  
Object literal example:- var obj = {};  
  
function constructors:- Constructing a object using function or A normal fucntion that is used to construct objects is called function constructor.  
  
function Person(firstname, secname){  
 console.log(this); // Person{}   
 this.fname = firstname;  
 this.sname = secname;  
}

var obj1 = new Person("mohan","kumar")  
console.log(obj1); //Person {fname: "mohan", sname: "kumar"}

var obj2 = new Person("rakesh","kumar")  
console.log(obj2); //Person {fname: "rakesh", sname: "kumar"}

var obj1 = new Person()  
new operator in front of function call will create an empty object and call Person() function in Person() function “this” points to newly created object and function return object to variable obj1  
  
var obj1 = {} person fucntion call;

No “return” statement in side Person() object but still it returns an object with some property because of “new” keyword in front of function invoke.  
  
**if we return an object from function constructor**  
function Person(firstname, secname){  
 this.fname = firstname;  
 this.sname = secname;  
 return {greeting:"hi"}

}

var obj1 = new Person("mohan","kumar");  
console.log(obj1); // {greeting: "hi"}

if we use return keyword or return an object from function constructor return object overrides the function constructor object.

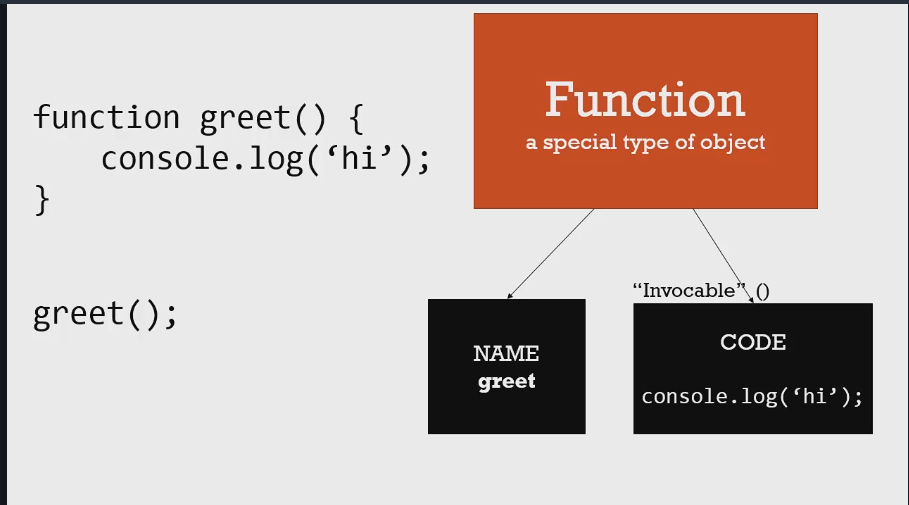
function Person(firstname, secname){  
 this.fname = firstname;  
 this.sname = secname;  
 return "abc"

}

var obj1 = new Person("mohan","kumar")  
console.log(obj1); //Person {fname: "mohan", sname: "kumar"}

if we return string or Boolean from function constructor this will be ignored and object created from function constructor will be assign to variable.   
  
**“this” keyword in side function constructor:-** it’s an empty object of type function constructor. console.log(this); // Person{}   
  
**function constructor always returns an object.**

**Functions:-**functions are special object in javascript.



Equal operator:-  
 ex: a = 3;   
equal operator function takes 2 parameter “a” and “3”, so it assign second parameter value to first parameter memory.

Two types of function

1. function Expression
2. function statement

Function statement:

function abc(){  
 console.log(“hi”);  
}  
abc();  
  
as soon as control see the function body memory will be allocated for function called abc() and once we invoke the function it search in the memory by name abc it fined trigger execute the function body.  
  
Function Expression:

var abc = function(){  
 console.log(“hi”)  
}

abc();  
  
as soon as control see the function expression variable “abc” is hoisted with value “undefined”, later once it hits this line of code it replace undefined value with function body so called anonymous function.  
  
once we invoke function using variable “abc”, variable abc holds the function body so it execute the function body.

Note:- we can pass variable, array, Boolean, object, function in the function parameter.  
  
function xyz(a){  
 a();   
}   
  
xyz( function(){ console.log(“hi”); } )

when we pass anonymous function, function will be stored in memory and variable a with undefined value.  
later anonymous function will be assign to variable a ie variable a holds the memory of anonymous function.

**Call By value and call by reference**Call by value:- holds value

var a= 2;  
var b= 3;

a = b;

console.log(a); //3  
console.log(b); //3

b = 10;  
console.log(a); // 3  
console.log(b); //10

Call by reference store value, in this case two variable will be created with respective values after a == b, value of b will be copied to a so a and b has same values.  
  
any change in b value variable a does not know about this change because both variable refer to different memory location.

Call by reference:- holds memory pointer of object

var obj1 = {greeting:"hi"};  
var obj2;

obj2 = obj1;

console.log(obj1); // {greeting: "hi"}  
console.log(obj2); // {greeting: "hi"}

obj1.greeting = "hellow";

console.log(obj1); // {greeting: "hellow"}  
console.log(obj2); // {greeting: "hellow"}  
  
obj1 = {greeting:"namaskara"};

console.log(obj1); // {greeting: "namaskara"}  
console.log(obj2); // {greeting: "hellow"}

obj2 = obj1;  
this line pass the memory reference or memory pointer of obj1 to obj2 variable ie both variables points to same memory address, so any change in the value(object) both will receive the same update from memory.

obj1 = {greeting:"namaskara"};  
if we assign new value to obj1, new memory will be created ie now obj1 holds two memory one for {greeting:"hi"}; and another for {greeting:"namaskara"};   
Still obj2 refer to old memory until we update to new memory.

if we do obj2 = obj1 again old memory will be destroyed ie old memory will be move to garbage collection.  
  
Why old memory still exist after updating obj1 = {greeting:"namaskara"};  
Garbage collection work on the concept that if any object does not holds any reference in memory that will be treated as garbage value or object and its deleted.   
  
but in this case obj2 still holds the reference of old object. So that object not move to garbage.

**“this” keyword**

“this” is an global object which is equal to window object.

if(this == window) //true

this // Window {postMessage: *ƒ*, blur: *ƒ*, focus: *ƒ*, close: *ƒ*, parent: Window, …}  
window // Window {postMessage: *ƒ*, blur: *ƒ*, focus: *ƒ*, close: *ƒ*, parent: Window, …}

“this” inside function expression or function declaration will refer to global object.   
  
function abc(){  
 console.log(this) ; //Window {postMessage: *ƒ*, blur: *ƒ*, focus: *ƒ*, close: *ƒ*, parent: Window, …}  
}  
  
**Creating a global variable in JS**  
window.initval = 20; or this.initval = 20;  
  
function abc(){  
 console.log(initval) //20  
}  
  
in above example no need to use window or this keyword to access the global variable value.

**“this” inside object method, anonymous function inside object is called method not function**

var obj = {  
 name:”mohan”,  
 getname: function(){  
 console.log(this) //refer to “obj”  
 console.log(this.name) //mohan  
 this.name = “kumar”;  
 console.log(this.name); //kumar  
  
  
 var abc = function(){  
 console.log(this);

/\*refer to global object not “obj” variable Window {postMessage: *ƒ*, blur: *ƒ*, focus: *ƒ*, close: *ƒ*, parent: Window, …}  
\*/   
  
 }  
 abc();  
  
 }  
}

How to access object “this” in side function expression  
  
var obj = {  
 name:”mohan”,  
 getname: function(){  
 var \_this = this;  
 var abc = function(){  
 \_this.name = “kumar”;  
 console.log(\_this.name); //kumar  
 }  
 abc();  
 }  
}

**WE can store function in an array like string, number, Boolean or object.**

var myarr= [false, “hi”, {name: “mohan”}, function(n){ console.log(n); }];  
  
console.log(myarr[3]( myarr[2].name )) // mohan

**function overloading**:- having a same function name with different parameter  
  
function greeting(){  
}

function greeting(name){  
}

function greeting(name, age){  
}

All above function has same name based on function parameter particular function will be called so its function overloading it’s supported in Java, C, C++ etc…

JavaScript does not support function overloading because functions are objects in JavaScript which overwrites the same function and invokes the latest function.

**IIFE(immediately inoved function expression)**IIFE using function expressionvar abc = function(name){  
 console.log(“hi ”+name); //hi mohan  
}(“mohan”);  
  
  
IIFE using function declaration   
(function(name){ 0r (function abc(name){  
 console.log(“hi ”+name); //hi mohan  
})(“mohan”)function(){ // gives an error “Unexpected token )”  
}  
  
(function(){  
}) // return function body  
  
(function(){  
})() // invokes this anonymous function  
  
function expression  
  
var abc = function(name){  
 return “hi ”+name;  
}(“mohan”);  
  
console.log(abc); //hi mohan  
return string to variable “abc” not the function body to invoke ie a string value is returned from the function and stored in a variable abc.

IIFE is a JavaScript pattern everything(variable, function) inside IIFE is private cannot be access out of it. Each variables and function inside the IIFE will have separate memory location ie if 2 IIFE has same variable for same variable 2 different memory will be created.   
  
var fname = "mohan";  
var lname = "kumar";  
var mname = "r";  
(function(name){  
 var mname = "R";  
 var city = "bangalore";  
 console.log(mname); //R  
 console.log(fname); //mohan  
 console.log(lname); // kumar  
})(fname)

console.log(city); //error

**Closures concepts**

**Concept 2:-**

function outerfun(fname){  
 return function(sname){  
 console.log(fname+" "+sname)  
 }

}

var ifun = outerfun("mohan");  
ifun("kumar") // mohan kumar  
  
the big question here is how anonymous function inside outerfun() holds the fname variable value.

First memory will be created for outerfun() when we invoke it variable fname is created and returns the anonymous function and outerfun() moved to garbage collection.

But in this case inner anonymous function holds memory pointer for outer function means outerfun() will not be moved to garbage collection so outerfun() will always holds its variable values init.

So when we call inner anonymous function its global context is outerfun() and it can and it will access global reference variable.

**Concept: - 3**  
  
function outfun(){  
var arr= [];  
for(var i=0; i<3; i++){  
 arr.push(function(){  
 console.log(i);  
 })

}

return arr;

}

outfun()[0](); //3

outfun()[1](); //3

outfun()[2](); //3

why all three invocation result is 3, this is closure inner anonymous function holds the result of outer function when we run each of inner anonymous function it will access var i value of outer function from memory so the result is always 3 from memory.

How to over come this problem  
we have 2 different approach to overcome

Approach 1:- Using “let” key work which is block specific, each time it executes it creates new variable of same kind in memory.

function outfun(){  
var arr= [];  
for(var i=0; i<3; i++){  
 let j = i;  
 arr.push(function(){  
 console.log(j);  
 })

}

return arr;

}

outfun()[0](); //0

outfun()[1](); //1

outfun()[2](); //2

In the above example each time for loop executes a new variable of name “j” is created, inside anonymous function will points to memory location of “j” which has been created in that loop.

outfun()[0](); points to j=0 in the memory so on.

Example:

if(a){ let a= 10; }   
console.log(a); //a is not defined error  
  
if(a){ var a= 10; }   
console.log(a); //10

Approach 2:-

function outfun(){  
var arr= [];  
for(var i=0; i<3; i++){  
 arr.push((function(j){  
 return function(){ console.log(j); }  
 })(i))  
 }

return arr;

}

outfun()[0](); //0

outfun()[1](); //1

outfun()[2](); //2  
  
In above example we have three level of closures first outfun(), second IIFE, third anonymous function.

While pushing to the array IIFE will be executed and updated value will be stored inside the variable “j” of each IIFE, so anonymous function will refer to its parent context closure concept.

Example:

function makeGreeting(language) {

return function(firstname, lastname) {

if (language === 'en') {

console.log('Hello ' + firstname + ' ' + lastname);

}

if (language === 'es') {

console.log('Hola ' + firstname + ' ' + lastname);

}

}

}

var greetEnglish = makeGreeting('en');

var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe'); // Hellow John Doe  
greetSpanish('John', 'Doe'); // Hola John Doe

When makeGreeting('en'); is called it return the anonymous function in which refer to the memory of language variable which created in this function invoke and so on.

**Javascript callback function:**

function fun(callback){  
 callback();  
}  
  
  
fun(function(){  
 console.log(“hi”)  
})  
  
//hi  
  
  
  
**call() apply() and bind()**

function is a special kind of object in JS, which has Name property optional and code property. Object can have properties and methods.

Code property can be invocable.

Function object has call(), apply() and bind() methods attached to it.

Is there any way to control “this” keyword in terms of global context, yes we can using bind(), call() and apply() methods.

**bind() method**

var gval = 20;  
var person = {

fname:"mohan",  
 sname:"kumar",  
 getname:function(){  
 return this.fname +" "+ this.sname;  
 }  
}

var greeting = function(){  
 console.log(this.getname()) //refer to person object “this”  
 console.log(this.gval) //error  
 }.bind(person)  
greeting(); //mohan kumar

bind() is an function object method. Which binds the object to the function so that function can refer the properties and methods of binded object.   
  
above example can be binded like below  
  
var greeting = function(){  
 console.log(this.getname()) //refer to person object “this”  
 }  
var newfun = greeting.bind(person);  
newfun(); // mohan kumar  
  
greeting.bind(person); greeting is an object which returns entire function body for which we are binding person object and return the function copy to newfun variable.  
  
  
**call() method**  
call() executes the function with parameter, while bind() wont execute but it just take a copy.

var gval = 20;  
var person = {

fname:"mohan",  
 sname:"kumar",  
 getname:function(){  
 return this.fname +" "+ this.sname;  
 }  
}

var greeting = function(lang1, lang2){  
 console.log(this.getname()) //refer to person object “this”  
 }

greeting.call(person, “eng”, “span”) // mohan kumar

**apply() method**

only difference between call() and apply() is in call we need to pass object and parameter but in apply() we need to pass object and array as second parameter.  
  
greeting.call( person, “eng”, “span” ) ;   
greeting.apply( person, [“eng”, “span”] );

example:-

(function(){  
 console.log(this.getname()) // mohan kumar  
}).apply(person,[“eng”,”span”])

Above example invoke IIFE using apply() method.

**Function borrowing**var person1 = {  
 fname:"mohan",  
 sname:"kumar",  
 getname:function(){  
 return this.fname +" "+ this.sname  
 }  
}

var person2 = {  
 fname:"Mahesh",  
 sname:"kumar",  
}

person1.getname.call(person2); // Mahesh kumar

**function currying**

Creating a copy of a function but with some preset parameters.

function multiple(a,b){  
 return a\*b;  
}  
  
var multipleoftwonumber = multiple.bind(this, 2);  
console.log(multipleoftwonumber(4,3)) // 8

var multipleoftwonumber = multiple.bind(this);  
console.log(multipleoftwonumber(4,3)) // 12

So if we pass any parameter to bind() method that remains preset ie always first parameter is 2 in case of first example, second parameter is 4 and third parameter is 3. “this” key word refer to global object.

This approach will be used in some mathematical functions in any case one or two parameter is fixed even user pass his own parameter.

**Functional programming**

**Example 1:**

var arr1 = [1,2,3];

function multiple(arr,callback){  
 var resultArr = [];  
 for(var i=0; i<arr.length; i++){  
 resultArr.push(callback(arr[i]))

}

return resultArr;  
}

var result = multiple(arr1, function(item){  
 return item\*2;  
})

console.log(result); // [2, 4, 6]

var result1 = multiple(arr1, function(item){  
 return item > 2;  
})

console.log(result); // [false, false, true]

**example 2:**

var arr1 = [1,2,3];

function multiple(arr,callback){  
 var resultArr = []

for(var i=0; i<arr.length; i++){  
 resultArr.push(callback(arr[i]))

}

return resultArr;

}

var setlimitval = function(limit){

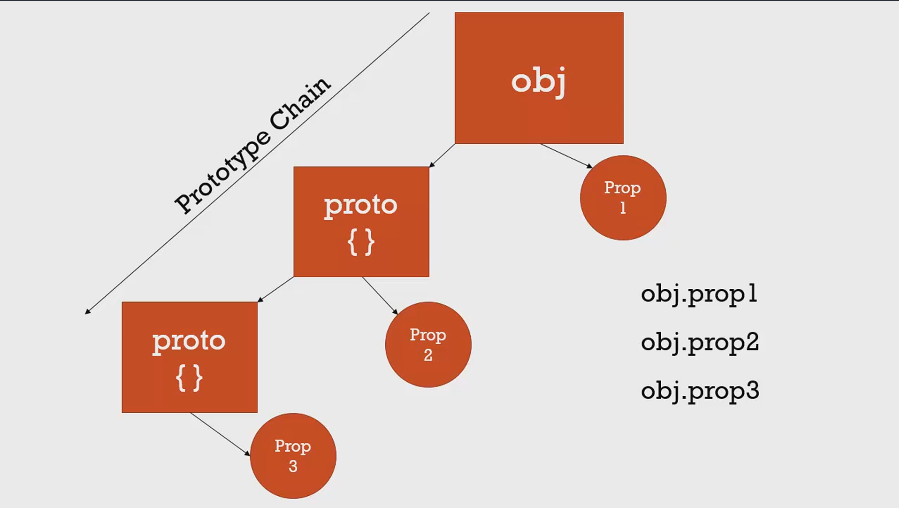
return function(limit, item){  
 return limit > item;  
}.bind(this,limit)

}

var result = multiple(arr1, setlimitval(2));

console.log(result); //  [true, false, false]

**Object oriented JavaScript and Prototypal Inheritance**



* Object refer to another object that refer object is called prototype object.
* Every object in JS will have prototype object.
* There is an hidden property in object called prop(\_\_prop\_\_) which holds the prototype object reference.
* If we try to access method or property in an object if it finds gets the value if not it goes to its prototype object and soo on until it complete the prototype chain. If it finds in prototype chain stop the search or it search till final prototype.

var person = {

firstname: 'Default',

lastname: 'Default',

getFullName: function() {

return this.firstname + ' ' + this.lastname;

}

}

var john = {

firstname: 'John',

lastname: 'Doe'

}

// don't do this EVER! for demo purposes only!!!

john.\_\_proto\_\_ = person;

console.log(john.getFullName()); // John Doe

console.log(john.firstname); //John

var jane = {

firstname: 'Jane'

}

jane.\_\_proto\_\_ = person;

console.log(jane.getFullName()); //Jane Default

person.getFormalFullName = function() {

return this.lastname + ', ' + this.firstname;

}

console.log(john.getFormalFullName()); //Doe John

console.log(jane.getFormalFullName()); // Default Jane

Everything in JS is Object, so all this objects will have prototype.   
Array is an object, function is an object etc…

“Object” is the base or core of all object in javascript.

Example1:-

var a = {};  
console.log(a.\_\_proto\_\_); // Object() { [native code] }  
console.log(a.\_\_proto\_\_. hasOwnProperty);   
  
a.\_proto\_\_ gives the prototype object of our “a” object, prototype property of “a” object is base object ie Object() { [native code] }

a.\_\_proto\_\_.hasOwnProperty; “hasOwnProperty” is the property of base object

Example2:-

var b = [];  
console.log(b.\_\_proto\_\_.constructor) // Array() { [native code] }  
console.log(b.\_\_proto\_\_.\_\_proto\_\_.constructor); // Object() { [native code] }

Any array prototype is Array(){} object and Array(){} object prototype is base prototype

Example3:-

var c = function(){};  
console.log(c.\_\_proto\_\_); // f(){}  
console.log(c.\_\_proto\_\_.\_\_proto\_\_); // Object() { [native code] }

Any function prototype is f(){} object and f(){} object prototype is base prototype

Example3:-   
var person1 = {fname:"mohan"}  
var person2 = {lname:"kumar"}  
var person3 = {getname:function(){ return this.fname+" "+this.lname}}

var getallkey = function(obj){  
 if(!isobj(obj)) return [];  
 var keyarr = [];

for(var myobj in obj){  
 if(obj.hasOwnProperty(myobj)){  
 keyarr.push(myobj);  
 }  
 }

return keyarr;

}

var isobj = function(obj){  
 var type = typeof obj;  
 return type === 'function' || type === 'object' && !!obj;  
}

var getextended = function(getallkey){

return function(obj){

//obj gives first parameter

var length = arguments.length;

if (length < 2 || obj == null) return obj;

for(var i=1; i<arguments.length; i++){  
 var source = arguments[i],  
 keys = getallkey(source)

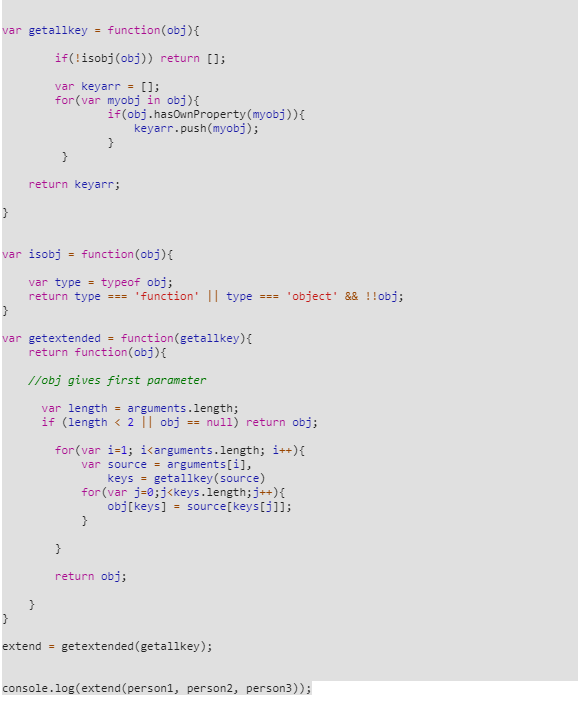
for(var j=0;j<keys.length;j++){  
 obj[keys] = source[keys[j]];

}   
 }

return obj;  
 }  
}

extend = getextended(getallkey);

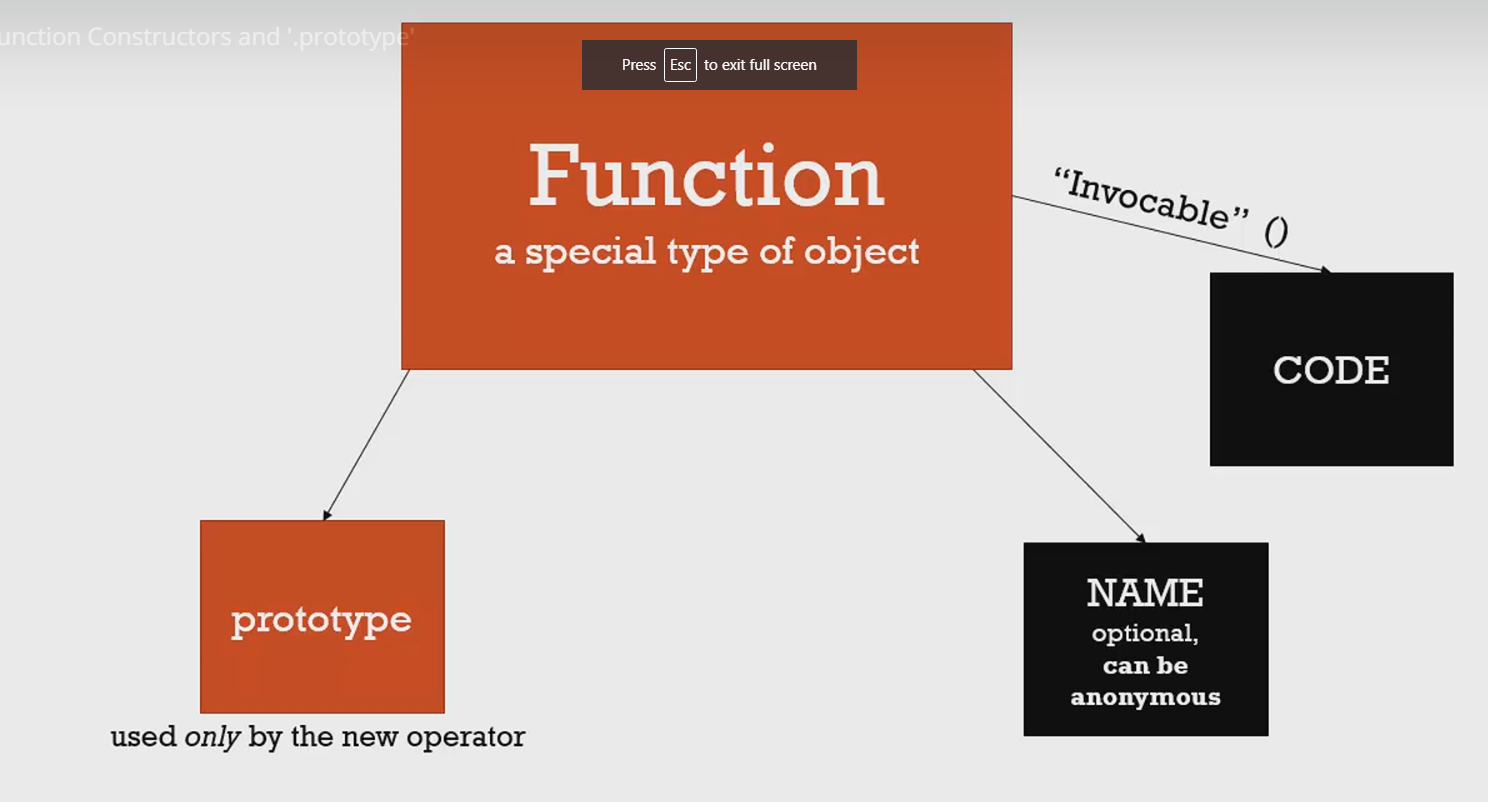
console.log(extend(person1, person2, person3));



Example: ajax call  
  
var ajaxcall = function(url, type, callback){  
 setTimeout(function(){ callback("success"); },5000); //$.ajax return  
}

ajaxcall("/login", ”post”, function(result){  
 console.log(result); //we will get result after 5 sec   
})

**Difference between \_\_proto\_\_ and prototype**\_\_proto\_\_ is a hidden property in all object which holds the reference of its prototype object.



Function is a special type of object, every function has Name property and code which can be invocable. When we use “new” operator before a function call “prototype” property will be enabled.  
  
**Prototype property of a function is not prototype of the function, it a prototype of any objects.**

Ie prototype is an property of an object but not reference to its prototype constructor.

function Person(fn,sn) {  
 this.fname = fn,  
 this.sname = sn,  
 this.getname = function(){  
 return this.fname +" "+ this.sname;  
 }

}

var obj1 = new Person("mohan","kumar")  
console.log(obj1); // Person {fname: "mohan", sname: "kumar", getname: ƒ}  
console.log(obj1.\_\_proto\_\_.constructor); // Person(){}  
console.log(obj1.\_\_proto\_\_.\_\_proto\_\_.constructor); // Object() { [native code] }



Whenever we create an object using “new” operator that object holds the copy of properties and methods of function constructor along with that each and every object will have prototype property enabled to hold or refer prototype of on function constructor.

**In above example obj1 prototype points to Person.prototype**

**obj1.getpname();** it search for getpname() method in its own properties and methods if not found it search in prototype object of obj1. prototype object of obj1 is Person(){} so on upto native object.

**Good practise:-**always declare methods in prototype not in function constructor, declare only property inside function constructor. Declare prototype method before object creation.  
  
**if we Declaring a prototype function after object create.**   
  
var obj1 = new Person("mohan","kumar")  
Person.prototype.getpname = function(){  
 return this.fname +" "+ this.sname;  
}

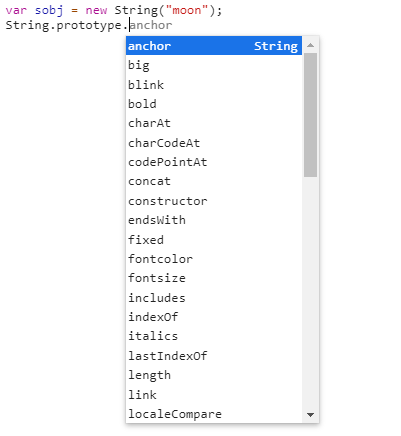
Console.log(obj1.getpname()); // mohan kumar  
  
still obj1 can access prototype method because method is not a property of an object it a prototype property.   
  
**why to declare property inside function constructor and methods using prototype property?  
  
Object is the collection of property and methods, property value keep in changing for each object eg:- this.fname;   
methods are used for operations, if we put methods in function constructor each and every object will have a copy of this method big memory waste or utilization for common operations.  
  
if we declare methods has prototype property it’s a single copy in prototype and all objects will refer to it, memory saved.**

**Built in function constructor**

var s = "moon"; //string literal return string  
console.log(s); // moon   
  
var sobj = new String("moon"); // string function constructor return object  
console.log(sobj); // String {"moon"}

(s === sobj) // false

Difference between string literal and string function constructor is, string literal returns string and string function constructor returns object.  
  
var sobj = new String("moon");   
console.log(sobj.\_\_proto\_\_.constructor); // ƒ String() { [native code] }  
console.log(sobj.\_\_proto\_\_.\_\_proto\_\_.constructor); // Object() { [native code] }



anchor, bold, concat etc… are String constructor prototype methods.

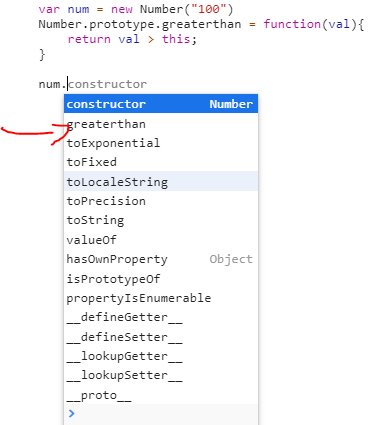
var n = new Number(123);  
var b = new Boolean();  
var date = new Date(); //return object

**Override or add new methods to build in constructor function**

It’s not good idea to override existing constructor function or prototype function.  
we can add new prototype methods but we need to handle with care.   
  
var num = new Number("100");  
Number.prototype.greaterthan = function(val){  
 return val > this;  
}

num.greaterthan(20); //false

num is the object and greaterthan() is the custom prototype method, this inside the greaterthan() refer to Number(){ } function constructor “this” holds 100 value passed in parameter.



var num = new Number("200");  
num.toFixed(2); // 200.00

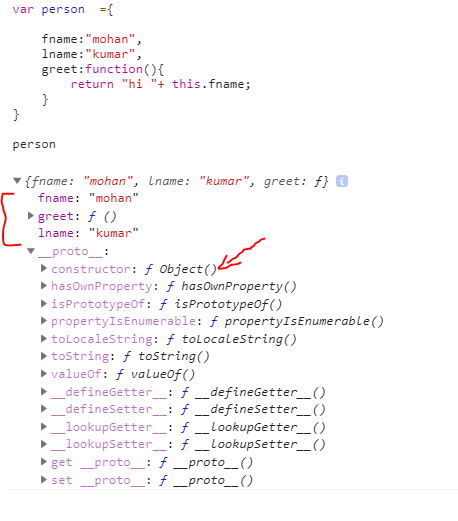
var num = new Number("500");  
Number.prototype. toFixed = function(val){  
 return this \* 2;  
}  
  
num.toFixed(2); // 1000  
  
so it not good practise to override existing prototype methods.

**Array is an object in Javascript**var namearr = ["ram","ravi","rakesh"];  
for(var obj in namearr){  
 console.log(obj+" "+namearr[obj])  
}  
namearr[0]; // ram

//output

0 ram  
1 ravi  
2 rakesh  
  
so at backend it stores like this  
var namearr = { 0:ram, 1:ravi, 2:rakesh };  
namearr[0]; // ram  
  
Array.prototype.custommethod = function(){  
}  
  
if we add custom method to Array constructor for in loop lists customemethod as well.

**Object.create() and pure prototypal Inheritance**

Javascript uses prototypal inheritance not classic inheritance in java, c, c++.  
  


When every we create an object here fname, lname and greet() are property and methods of person object. Object() is the prototype of person object.



Object.create(person) creates an empty object and its prototype property is person object.   
Object is base object, create() is the prototype method of base object.   
  
So later if **jhon.greet() //hi mohan** it search greet() in jhon object if not found it go to prototype object.



In above example jhon object has its own fname and lname properties and greet() method from prototype object.   
  
**jhon.greet()** // hi Ramesh

Using Object.create() we can create pure prototypal inheritance.

If your browser not support Object.create() use polufill, create our own polyfill for Object.create();  
  
// polyfill

if (!Object.create) {

Object.create = function (o) {  
 if (arguments.length > 1) {  
 throw new Error('Object.create implementation' + ' only accepts the first parameter.');  
 }  
 function F() {}  
 F.prototype = o;  
 return new F();

};

}

If Object.create() is not supported by browser, create a prototype method called ”create” in base object Object(), pass object as argument.  
create a empty function F() and its prototype is argument object and return the function constructor object.   
  
return new F() returns an object and its prototype is argument object which is o.