JAVASCRIPT INTERVIEW Q&A:

1. **What is a closure?**

A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

function showName (firstName, lastName) {

var nameIntro = "Your name is ";

// this inner function has access to the outer function's variables, including the parameter

function makeFullName () {

return nameIntro + firstName + " " + lastName;

}

return makeFullName ();

}

showName ("Michael", "Jackson"); // Your name is Michael Jackson

1. **Explain Prototype?**

First, every JavaScript function has a prototype property (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance. This prototype property is not enumerable; that is, it isn’t accessible in a for/in loop. But Firefox and most versions of Safari and Chrome have a \_\_proto\_\_ “pseudo” property (an alternative syntax) that allows you to access an object’s prototype property. You will likely never use this \_\_proto\_\_ pseudo property, but you should know that it exists and it is simply a way to access an object’s prototype property in some browsers.

The prototype property is used primarily for inheritance; you add methods and properties on a function’s prototype property to make those methods and properties available to instances of that function.

Consider this simple example of inheritance with the prototype property (more on inheritance later):

var Person = function(name) {

this.name = name;

this.canTalk = true;

};

Person.prototype.greet = function() {

if (this.canTalk) {

console.log('Hi, I am ' + this.name);

}

};

var Employee = function(name, title) {

Person.call(this, name);

this.title = title;

};

Employee.prototype = Object.create(Person.prototype);

Employee.prototype.greet = function() {

if (this.canTalk) {

console.log('Hi, I am ' + this.name + ', the ' + this.title);

}

};

var Customer = function(name) {

Person.call(this, name);

};

Customer.prototype = Object.create(Person.prototype);

var Mime = function(name) {

Person.call(this, name);

this.canTalk = false;

};

Mime.prototype = Object.create(Person.prototype);

var bob = new Employee('Bob', 'Builder');

var joe = new Customer('Joe');

var rg = new Employee('Red Green', 'Handyman');

var mike = new Customer('Mike');

var mime = new Mime('Mime');

bob.greet();

// Hi, I am Bob, the Builder

joe.greet();

// Hi, I am Joe

rg.greet();

// Hi, I am Red Green, the Handyman

mike.greet();

// Hi, I am Mike

mime.greet();

1. **Describe call, bind, apply method in javascript?**

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the fullName method of person, using it on person1:

var person = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person1 = {

firstName:"John",

lastName: "Doe",

}

var person2 = {

firstName:"Mary",

lastName: "Doe",

}

person.fullName.call(person1); // Will return "John Doe"

**apply()**

var person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}

var person1 = {

firstName:"John",

lastName: "Doe",

}

person.fullName.apply(person1, ["Oslo", "Norway"]); // will return John Doe,Oslo,Norway

The difference is:

The call() method takes arguments separately.

The apply() method takes arguments as an array.

**Bind()**

//Use .bind() javascript

var obj = {name:"Niladri"};

var greeting = function(a,b,c){

return "welcome "+this.name+" to "+a+" "+b+" in "+c;

};

//creates a bound function that has same body and parameters

var bound = greeting.bind(obj);

console.dir(bound); ///returns a function

console.log("Output using .bind() below ");

console.log(bound("Newtown","KOLKATA","WB")); //call the bound function

/\* the output will be

Output using .bind() below

welcome Niladri to Newtown KOLKATA in WB \*/

1. **What is Anonymous function?**

The function above is actually an anonymous function (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

var x = function (a, b) {return a \* b};

var z = x(4, 3); // will return 12

1. **Method overriding and overloading**

function A(){} /\* constructor \*/

A.prototype = { /\* especification of prop and methods of the class A \*/

num : 5,

name : 'test',

action : function(){ /\*base implementation\*/},

alert

};

function B(){ /\* constructor \*/

/\* called base constructor:\*/

A.call(this);

}

B.prototype = Object.create(A.prototype); /\* Object.create is suported by: Chrome5+, Firefox4+, IE9+, Safari5+ \*/

B.prototype.num = 20; /\* override prop\*/

B.prototype.action = function(base){ /\* overload method \*/

return function() {

/\* own implementation \*/

base();

};

}(A.prototype.action);

B.prototype.name = function(base){ /\* overload prop \*/

return base + " overload";

}(A.prototype.name);

/\* using \*/

var a = new A();

var b = new B();

console.log(a); /\*show this: {num: 5 , name: "test", action: function() {/\* ... \*/}} \*/

console.log(b); /\*show this: {num: 20, name: "test overload", action: function() {/\* ... \*/}} \*/

1. **Hoisting**

JavaScript Hoisting

Hoisting is a concept in JavaScript, not a feature. In other scripting or server side languages, variables or functions must be declared before using it.

In JavaScript, variable and function names can be used before declaring it. The JavaScript compiler moves all the declarations of variables and functions at the top so that there will not be any error. This is called hoisting.

x = 1;

alert('x = ' + x); // display x = 1

var x; // return 1

alert(Sum(5, 5)); // 10

function Sum(val1, val2)

{

return val1 + val2;

}

1. **For- of and for-in loop**

The for...of statement creates a loop iterating over iterable objects, including: built-in String, Array, Array-like objects (e.g., arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

let iterable = [10, 20, 30];

for (let value of iterable) {

value += 1;

console.log(value);

}

// 11

// 21

The for...in statement iterates over all non-Symbol, enumerable properties of an object.

var obj = {a: 1, b: 2, c: 3};

for (const prop in obj) {

console.log(`obj.${prop} = ${obj[prop]}`);

}

// Output:

// "obj.a = 1"

// "obj.b = 2"

// "obj.c = 3"

1. **Map**

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

var array1 = [1, 4, 9, 16];

// pass a function to map

const map1 = array1.map(x => x \* 2);

console.log(map1);

// expected output: Array [2, 8, 18, 32]

1. **Arrow function**

An arrow function expression has a shorter syntax than a function expression and does not have its own this, arguments, super, or new.target. These function expressions are best suited for non-method functions, and they cannot be used as constructors.

var materials = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

console.log(materials.map(material => material.length));

// expected output: Array [8, 6, 7, 9]

1. **Promise**

The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

var promise1 = new Promise(function(resolve, reject) {

setTimeout(function() {

resolve('foo');

}, 300);

});

promise1.then(function(value) {

console.log(value);

// expected output: "foo"

});

console.log(promise1);

// expected output: [object Promise]

1. **Asyn/Await function**

The async function declaration defines an asynchronous function, which returns an AsyncFunction object. An asynchronous function is a function which operates asynchronously via the event loop, using an implicit Promise to return its result. But the syntax and structure of your code using async functions is much more like using standard synchronous functions.

function resolveAfter2Seconds() {

return new Promise(resolve => {

setTimeout(() => {

resolve('resolved');

}, 2000);

});

}

async function asyncCall() {

console.log('calling');

var result = await resolveAfter2Seconds();

console.log(result);

// expected output: 'resolved'

}

asyncCall();

1. **Try/Catch**

function myFunction() {

var message, x;

message = document.getElementById("p01");

message.innerHTML = "";

x = document.getElementById("demo").value;

try {

if(x == "") throw "is empty";

if(isNaN(x)) throw "is not a number";

x = Number(x);

if(x > 10) throw "is too high";

if(x < 5) throw "is too low";

}

catch(err) {

message.innerHTML = "Error: " + err + ".";

}

finally {

document.getElementById("demo").value = "";

}

}