**Q1. Create a hierarchy of person, employee and developers.**

function Person() {

this.dept = 'general';

}

function Employee() {

Person.call(this);

this.projects = [];

}

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

Employee.prototype.constructor = Employee;

function Developer() {

Employee.call(this);

this.quota = 100;

}

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

Developer.prototype.constructor = Developer;

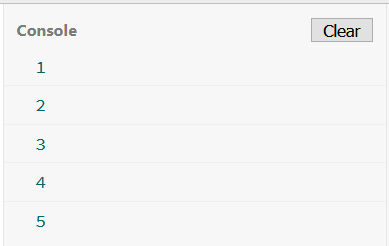
**Q2. Given an array, say [1,2,3,4,5]. Print each element of an array after 3 secs.**

var array = [1, 2, 3, 4, 5]

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

setTimeout(() => {console.log(array[i])}, 3000);

}

****

**Q3. Explain difference between Bind and Call (example).**

const person = {

firstName: 'John',

lastName: 'Doe',

printName: function() {

console.log(this.firstName + ' ' + this.lastName); }

};

person.printName();

John Doe

const printFullName = person.printName;

printFullName();

undefined undefined

Here, we are storing a reference of person.printName to printFullName variable. After that, we are calling it without an object reference, so this will now refer to the window (global) object or undefined (in strict mode).

We use call, bind and apply methods to set the this keyword independent of how the function is called.

# **Bind( )**

The bind method creates a new function and sets the this keyword to the specified object.

const john = {

name: 'John',

age: 24,

};

function greeting() {

console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);

}

const greetingJohn = greeting.bind(john);

greetingJohn();

Hi, I am John and I am 24 years old

# **Call ( )**

The call method sets the this inside the function and immediately executes that function.

The difference between call() and bind() is that the call() sets the this keyword and executes the function immediately and it does not create a new copy of the function, while the bind() creates a copy of that function and sets the this keyword.

function greeting() {

console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

name: 'John',

age: 24,

};

greeting.call(john);

Hi, I am John and I am 24 years old

**Q4. Explain 3 properties of argument object.**

The arguments object is a local variable available within all non-[arrow](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) functions. You can refer to a function's arguments inside that function by using its arguments object. It has entries for each argument the function was called with, with the first entry's index at 0.

The arguments object is not an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). It is similar, but lacks all Array properties except [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length).

The [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) operator returns 'object' when used with arguments. The type of individual arguments can be determined by indexing arguments.

[arguments.callee](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/callee)

Reference to the currently executing function that the arguments belong to. Forbidden in strict mode.

[arguments.length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/length)

The number of arguments that were passed to the function.

[arguments[@@iterator]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/@@iterator)

Returns a new [Array iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/@@iterator) object that contains the values for each index in arguments.

**Q5. Create a function which returns number of invocations and number of instances of a function.**

var myFuncCalls = 0;

function myFunction()

{

myFuncCalls++;

}

myFunction();

myFunction();

myFunction();

myFunction();

console.log( "I have been called " + myFuncCalls + " times" );

"I have been called 4 times"

**Q6. Create a counter using closures.**

JavaScript variables can belong to the local or global scope. Global variables can be made local with closures.

var add = (function () {

var counter = 0;

return function () {counter += 1; console.log(counter);}

})();

add();

add();

add();

**1**

**2**

**3**

**Q7. Explain 5 array methods with example.**

1. **The slice() method slices out a piece of an array into a new array.**

var city = new Array("delhi", "agra", "akot", "aligarh","palampur");

console.log(city.slice(2));

["akot", "aligarh", "palampur"]

1. **The shift() method removes the first array element and "shifts" all other elements to a lower index.**

var city = new Array("delhi", "agra", "akot", "aligarh","palampur");

console.log(city.shift());

"delhi"

1. **The pop() method removes the last element from an array**

var city = new Array("delhi", "agra", "akot", "aligarh","palampur");

console.log(city.pop());

"palampur"

1. **The sort() method sorts an array alphabetically.**

var city = ["delhi", "agra", "akot", "aligarh"];

console.log(city.sort());

["agra", "akot", "aligarh", "delhi"]

1. **The concat() method creates a new array by merging (concatenating) existing arrays**

var city = ["delhi", "agra", "akot", "aligarh"];

var city2 = ["kolkata", "chennai", "pune"];

console.log(city.concat(city2));

["delhi", "agra", "akot", "aligarh", "kolkata", "chennai", "pune"]