**Exercise 3.2:-**

**Create 3 simple functions where call, bind and apply are used. The intention of this exercise is to understand how they work and their differences.(0.5 hours)**

Solution:

1. **Call() function:-**

We can call any function, and *explicitly specify what* ***this*** *should reference* within the calling function.

code

let getName = function(hobby1, hobby2) {

//here ‘this’ will be ‘person’ object if we call function and individual arguments need to be provided

console.log(this.name + ' likes ' + hobby1 + ' , ' + hobby2 + ' and lives in ' + this.city);

}

let person = {

name: 'Yogesh',

city: 'Siliguri'

};

let hobbies = ['football', 'music'];

getName.**call**(person, hobbies[0], hobbies[1]);

**Output:**“Yogesh likes football, music and lives in Siliguri”

1. **bind() function:-**

The bind() method is similar to the call() method but with one difference. Unlike the call() method of calling the function directly, bind() returns a brand new function and we can invoke that instead.

const person = {

firstName:"John",

lastName: "Doe",

fullName: function () {

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

}

}

const member = {

firstName:"Hege",

lastName: "Nilsen",

}

let fullName = person.fullName.bind(member);

fullName(); // output is ‘Hege Nilsen’

1. **apply() function:-**

It is exactly the same as call() but allows you to pass the arguments more conveniently.

let getName = function(hobby1, hobby2) {

//here ‘this’ will be ‘person’ object or any other object by which we have called using apply() function.

console.log(this.name + ' likes ' + hobby1 + ' , ' + hobby2 + ' and lives in ' + this.city);

}

let person = {

name: 'Yogesh',

city: 'Siliguri'

};

let hobbies = ['football', 'music'];

getName.**apply**(person, hobbies);

//same code we are able to pass the arguments using the array.

**Output:**“Yogesh likes football, music and lives in Siliguri”

**Exercise 3.3**

**What is the output of the below problem and why:[30 min]**

**function createIncrement(){**

**let count=0;**

**function increment(){**

**count++;**

**}**

**let message='Count is ${count}’;**

**function log(){**

**console.log(message);**

**}**

**return[increment,log];**

**}**

**const [increment,log]=createIncrement();**

**increment();**

**increment();**

**increment();**

**log();**

Solution:

1. Initially a global execution context will be created where function **createIncrement** will be provided a space.
2. Next new execution context will be created for function **createIncrement().** This execution will provide a space for the “count” variable as undefined.
3. Once function **increment()** is called a new execution context is created where locally count will have value as undefined. The local variables within a function exist for just the duration of that function's execution. Thus calling the increment() function will only increment the count three times and will be changed in the increment function execution context and the count outside will not be affected.
4. But when we call **log()** function, there is no variable defined as “count” inside the function. In javascript the functions are bundled with its lexical scope. So value will be taken from its lexical parent where count =0.
5. Thus the final output will be

**‘Count is 0’**