Module 3

Exercise 3(OOJS JavaScript)

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

A1.)

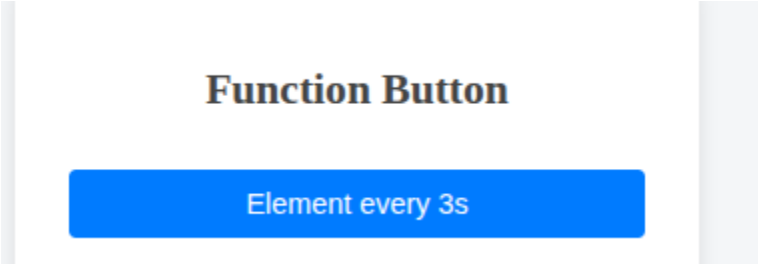
```
function Person(name,address){
    this.name=name;
    this.address=address;
}

function Employee(designation,compentency){
    this.designation=designation;
    this.compentency=compentency;
}

function Developer(canCode){
    this.canCode=canCode;
}


Employee.prototype=new Person("Dhruv","Delhi");

Developer.prototype=new Employee("Software Engineer Trainee","Js");

var developer=new Developer(true);

console.log(developer.name);
console.log(developer.designation);
console.log(developer.compentency);
console.log(developer.canCode);
```

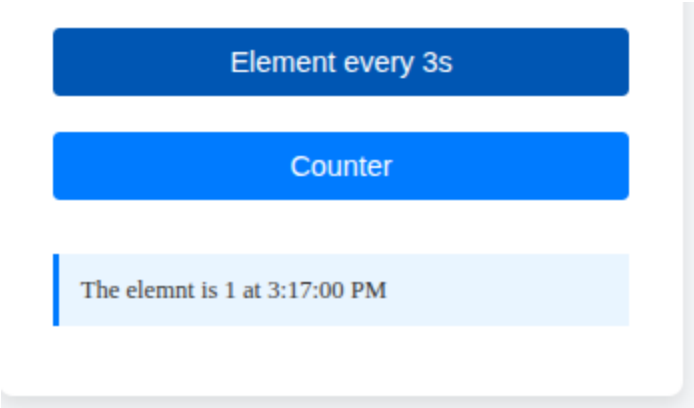
```
Dhruv
Software Engineer Trainee
Js
true
```

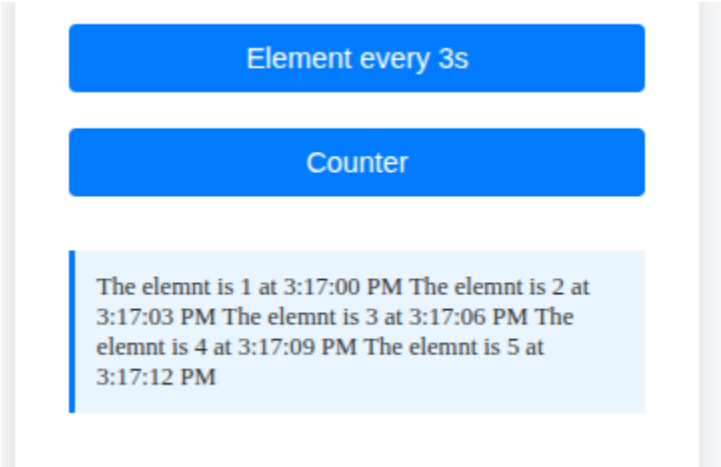
Q2. Given an array, say [1,2,3,4,5]. Print each element of an array after 3 secs.

A2.)

```
function printElement(){
    let array=[1,2,3,4,5];
    let i=0;

    let intervalID=setInterval(() => {
        const now = new Date();
        const timeWithSeconds = now.toLocaleTimeString('en-US', {
            hour: 'numeric',
            minute: 'numeric',
            second: 'numeric',
            hour12: true,
        });
        document.getElementById("result").innerHTML+=`\n
        The elemnt is ${array[i++]} at ${timeWithSeconds}`;
        if(i===array.length){
            clearInterval(intervalID);
        }
    }, 3000);
}
```







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

A3.)

Feature	Bind	Call
Execution	The Function.bind does not execute the function immediately.	The Function.call will execute the function immediately.
Return Value	Returns a new function with the **this** context and (optional) argument permanently bound.	Returns the result of the function call
Usage	Useful for Callbacks (e.g. eventListeners and setTimeout). Where you	Useful for immediately invoking the function with a specific **this** value or

Feature	Bind	Call
	need to preserve **this** context for future execution	for method borrowing
Arguments	Accepts a **this** value and optional initial arguments subsequent arguments are passed to the return function during call.	Accepts the **this** value as the first argument, and all other arguments are passed individually, separated by commas.

Example Bind:

```
const person = {
  name: "John",
  greet: function(city) {
    console.log(`Hello, my name is ${this.name} from ${city}.`);
  }
};

const anotherPerson = {
  name: "Jane"
};
```

```
const greetJane = person.greet.bind(anotherPerson, "London");

greetJane();
```

```
Hello, my name is Jane from London.
```

Call

```
const person = {
  name: "John",
  greet: function(city) {
    console.log(`Hello, my name is ${this.name} from ${city}.`);
  }
};

const anotherPerson = {
  name: "Jane"
};
```

```
person.greet.call(anotherPerson, "New York");
```

```
Hello, my name is Jane from New York.
```

Q4. Explain 3 properties of argument object.

A4.)

- **arguments.length** : This property contains the count of the actual number of arguments passed to the function when it was called. This can be different from the number of formally declared parameters in the function definition.

```
//Q4 Argument object

function test(a, b) {
  console.log(arguments.length);
}

test(1);
test(1, 2, 3);
```

```
1
3
```

- **arguments.callee** : This property holds a reference to the currently executing function. This was historically useful for anonymous functions to call themselves recursively, but it is deprecated and its use is discouraged in modern JavaScript, especially in strict mode. Rest parameters and named function expressions are preferred for recursion.

```
function factorial(n) {
  if (n <= 1) {
    return 1;
  }

  return n * arguments.callee(n - 1);
}

console.log(factorial(4));
```

```
24
```

- **arguments[@@iterator]** : This property returns an iterator for the arguments object's values. The arguments object itself is iterable, which allows it to be used with loops like for...of or converted into a true Array using methods like Array.from() or the spread syntax (...)

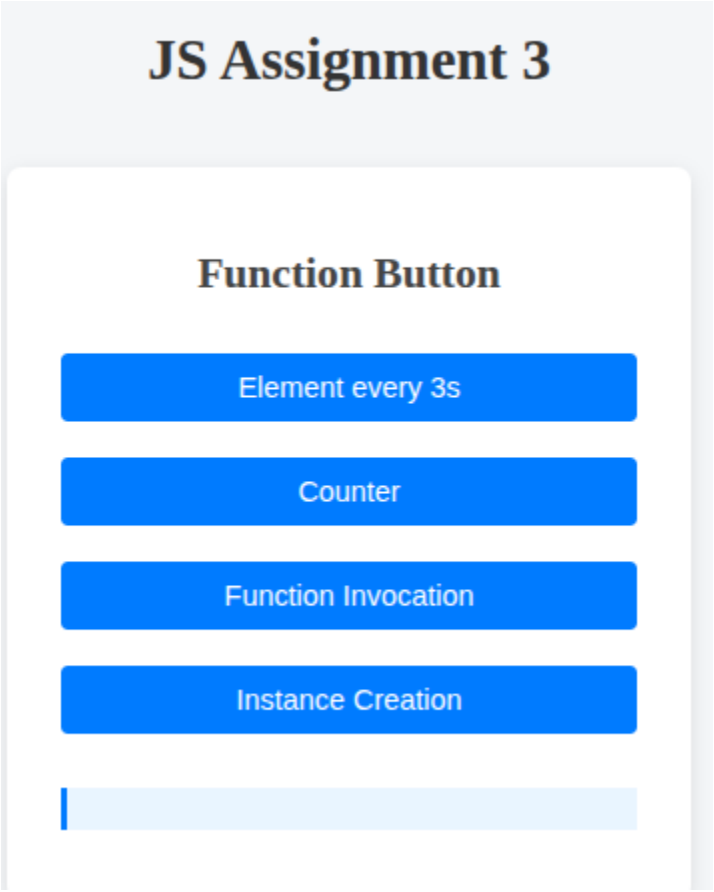
```
function sum() {
  let total = 0;
  for (const arg of arguments) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3, 4));
function getArgsAsArray() {
    const argsArray = [...arguments];
    console.log(Array.isArray(argsArray));
    return argsArray;
}

getArgsAsArray(5, 6, 7);
```

```
10
true
```

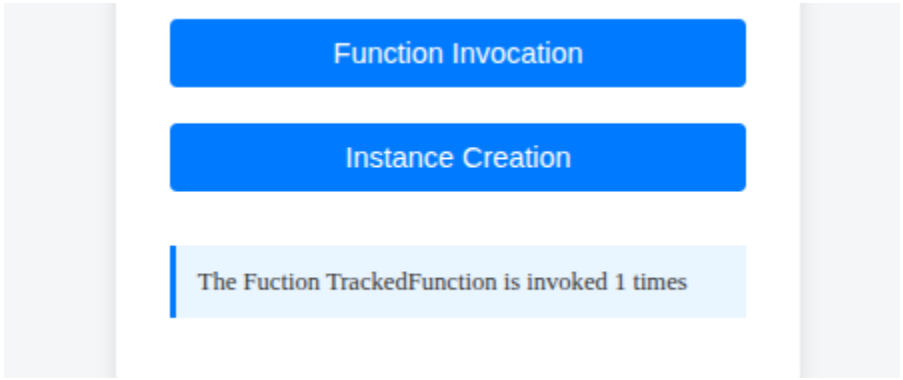
Q5. Create a function which returns number of invocations and number of instances of a function.

A5.)

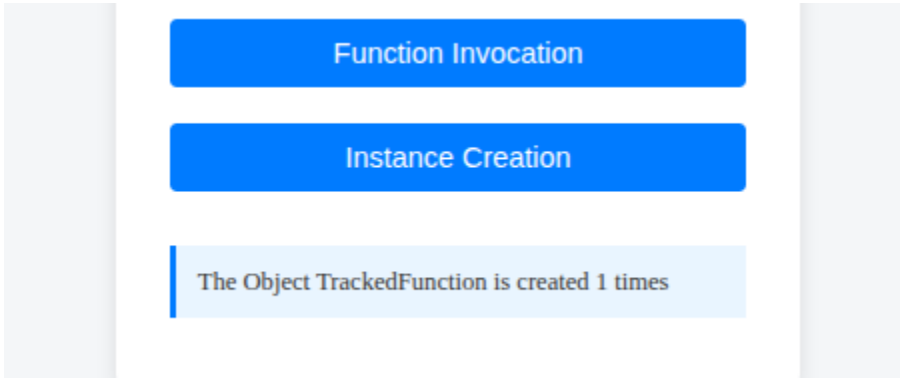
```
function TrackedFunction() {
  if (this instanceof TrackedFunction) {
    TrackedFunction.instanceCount++;
  } else {
    TrackedFunction.invocationCount++;
  }
}

TrackedFunction.invocationCount = 0;
TrackedFunction.instanceCount = 0;

document.getElementById("function-btn").addEventListener("click",()=>{
    TrackedFunction();
    document.getElementById("result").innerHTML=`The Fuction TrackedFunction is invoked ${TrackedFunc
})

document.getElementById("instance-btn").addEventListener("click",()=>{
    new TrackedFunction();
    document.getElementById("result").innerHTML=`The Object TrackedFunction is created ${TrackedFunct
})
```



Function Invocation



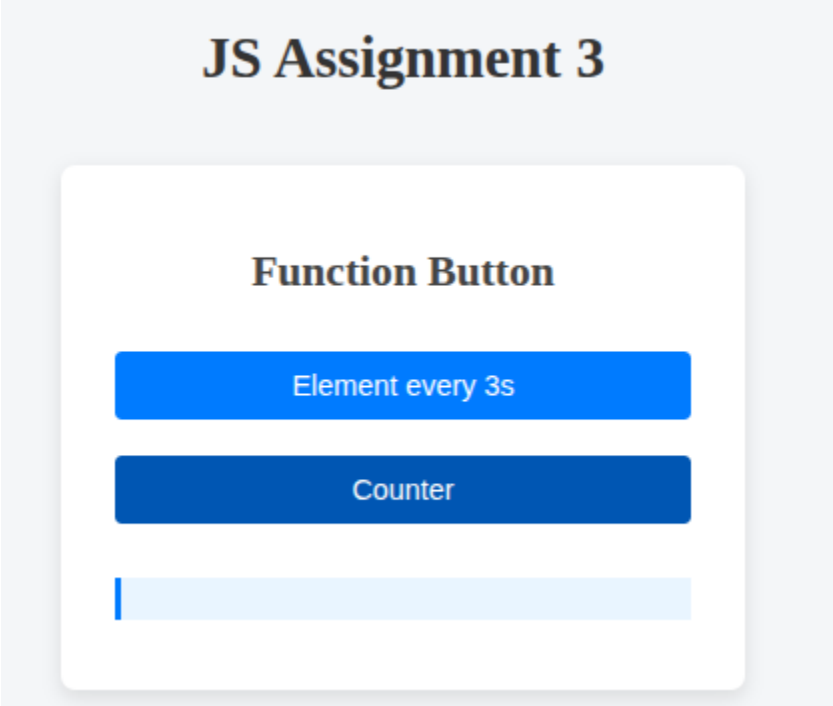
Object creation



Q6. Create a counter using closures.

A6.)

```
function counterClosure(){
    let count=0;
    return function(){
        count++;
        return count;
    }

}


var counter=counterClosure();

document.getElementById("counter-btn").addEventListener("click",()=>{
    document.getElementById("result").innerHTML=`The count is ${counter()}`;
});
```



Q7. Explain 5 array methods with example.

A7.)

- **map()** : The map() method creates a **new array** populated with the results of calling a provided function on every element in the calling array. It is ideal for transforming data without mutating the original array.

```
const prices = [100, 200, 300];

const withTax = prices.map(price => price * 1.2);
console.log(withTax);
console.log(prices);
```

```
▸ (3) [120, 240, 360]
▸ (3) [100, 200, 300]
```

- **filter()** : The filter() method creates a **new array** with all elements that pass the test implemented by the provided function. This is useful for selecting a subset of data based on specific criteria.

```
const scores = [45, 80, 90, 30];
const passed = scores.filter(score => score >= 50);
console.log(passed);
console.log(scores);
```

```
▸ (2) [80, 90]
▸ (4) [45, 80, 90, 30]
```

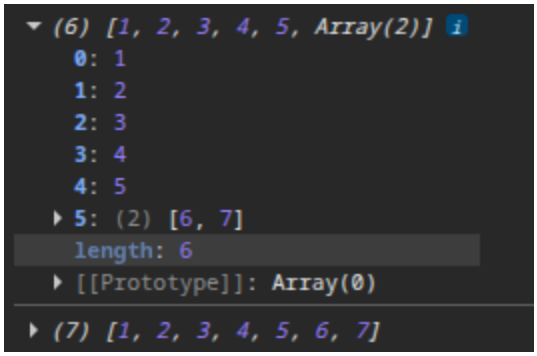
- **reduce()** : The reduce() method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. When the function is complete, it results in a **single value**.

```
const expenses = [10, 20, 5, 15];
const total = expenses.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(total);
```

```
50
```

- **flat()** : The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth. It's useful for working with nested arrays.

```
const nestedArray = [1, 2, [3, 4], [5, [6, 7]]];

const flatOnce = nestedArray.flat();
console.log(flatOnce);
const flatDeep = nestedArray.flat(Infinity);
console.log(flatDeep);
```



- **find()** : The find() method returns the **value of the first element** in the array that satisfies the provided testing function. If no elements satisfy the condition, undefined is returned. It stops iterating as soon as a match is found.

```
const users = [
  { name: 'Ada', age: 28 },
  { name: 'Tobi', age: 23 },
  { name: 'Tobi', age: 30 }
];

const user = users.find(u => u.name === 'Tobi');
console.log(user);
```