CLASS 11 - FUNCTIONAL PROGRAMMING

Agenda
Callback functions
Pure and Impure Functions
Higher Order Functions
Writing clean code with higher-order functions.
Array Methods(Map, Filter, Reduce, etc.)

Functional Programming

Following are some of common types of programming paradigms:

1. Procedural Programming Paradigm
    a. In procedural programming, the focus is on writing procedures or functions that operate on data. The program is structured as a sequence of procedures or function calls. The emphasis is on the procedure to perform an action. Languages like C are often used to illustrate this paradigm.
2. Object Oriented Paradigms(eg:- Java, C++)
    a. In contrast, object-oriented programming (OOP), as seen in languages like Java, structures a program around objects and their interactions. An object is an instance of a class, and it encapsulates data and the methods that operate on that data. I

    b. While both paradigms use functions or methods, procedural programming treats data and procedures as separate entities, whereas OOP binds data and the procedures (methods) that operate on the data together in objects.
3. Functional Programming Paradigm(eg:- javascript)
    a. Functional programming is a way of writing code in JavaScript where we focus on using functions in a special way.
    b. Imagine each function as a special unit: you give it some information (these are called 'inputs'), and it gives you back something new without changing the original information.
    c. This makes your code like a series of small, predictable steps. In functional programming, we try to avoid changing values once they're set, which makes our code easier to understand and less likely to have unexpected errors.
    d. each piece does one thing well and can be combined in many ways to create something larger.

Callback Functions

1. These are the functions that can be passed to another function as an argument.

```
function printName(cb){
   console.log(Shubham)
```

```
    // calling received callback function
    cb()
}
function printLastName(){
    console.log('Verma')
}
function printAge(){
    console.log(25)
}
printName(printLastName)
printName(printAge)
```

Output:
Shubham
Verma
Shubham
24

Explanation:
In printName(printLastName) statement, we are passing printLastName as a callback, and it is accepted by the printName as a cb argument. And when cb() is executed then the callback function is called.

A callback function is a function that is passed into another function as an argument and is expected to be executed at a certain point within the containing function's body.

We can also pass multiple callback arguments

```
function printName(cb1, cb2, cb3){
   console.log('Shubham')
   cb1()
   cb2()
   cb3()
}
function printLastName(){
   console.log('Verma')
}
function printAge(){
   console.log(25)
}
function printAddress(){
   console.log('Delhi')
}
printName(printLastName, printAge, printAddress)
```

Lets see one more example

```
function greet(name) {
 return `Hello, ${name}!`;
}
function farewell(name) {
 return `Goodbye, ${name}!`;
}

function createSalutation(name, fn) {
 console.log(fn(name));
}

createSalutation("Shubham", greet);
createSalutation("Shubham", farewell);
```

What are some benefits of using this pattern ?

1. Reusability and Abstraction: The createSalutation function is an abstraction that can be reused with different greeting functions.

By passing different callbacks (greet and farewell), the same function can perform different operations. This reusability reduces code duplication.

2. Flexibility: The callback pattern provides flexibility in the code. You can easily extend the code by introducing new behavior without modifying the createSalutation function itself. For example, you could write another function like inform and pass it to createSalutation without any changes needed in the createSalutation implementation.

Pure / Impure functions

1. Impure Functions in JavaScript
    a. **An impure function is a function that contains one or more side effects. It mutates data outside of its scope (a global variable for example) and does not predictably produce the same output for the same input.**
2. For example, consider the following code snippet:

```
var c = 0;
function sum(a, b){
 return a + b + c++;
}

console.log(sum(2, 3)); // 5
console.log(sum(2, 3)); // 6
```

3. In this, even if we call the function with the same arguments 2, 3 the output values change because of an external variable

Pure functions

1. What is a Pure Function?
2. A pure function is a function without any side effects, it will provide the same output for the same inputs/arguments provided.

```
function sum(a, b){
   return a + b;
 }
  console.log(sum(2, 3)); // 5
 console.log(sum(2, 3)); // 5
```

Is this pure ?

```
function calculateAge(birthYear) {
 const currentYear = new Date().getFullYear();
 return currentYear - birthYear;
}

console.log(calculateAge(1988));
```

Is this pure ?

```
function getRandomNumber() {
   return Math.random(); // Returns a new random number each time
}

console.log(getRandomNumber());
console.log(getRandomNumber());
```

Is this pure ?

```
function updateProfile(profile) {
   profile.lastUpdated = new Date(); // Modifies the object passed in
   return profile;
}
```

```
const userProfile = { name: "Jane Doe", lastUpdated: new Date('2024-03-07') };
console.log(updateProfile(userProfile));
```

Ans - no, because we mutate the passed param

Coding question (context for higher order function)

We are given an array, which has the radius of different circles, we need to find the area, circumference and diameter for all the radiuses in a result array.

```
let myRadiusArray = [2, 3, 4, 5, 8];

function calculateArea(radiusArr) {
 let result = [];
 for (let i = 0; i < radiusArr.length; i++) {
   result.push(3.14 * radiusArr[i] * radiusArr[i]);
 }
 return result;
}
let finalAreas = calculateArea(myRadiusArray);
console.log("This is area array => ", finalAreas);

function calculateCircumference(radiusArr) {
 let result = [];
 for (let i = 0; i < radiusArr.length; i++) {
   result.push(2 * Math.PI * radiusArr[i]);
 }
```

```
 return result;
}
let finalCircumferences =
calculateCircumference(myRadiusArray);
console.log("This is Circumference array =>",
finalCircumferences);

function calculateDiameter(radiusArr) {
 let result = [];
 for (let i = 0; i < radiusArr.length; i++) {
   result.push(radiusArr[i] * 2);
 }
 return result;
}
let finalDiameters = calculateDiameter(myRadiusArray);
console.log("This is Diameter array =>", finalDiameters);
```

What is wrong or inefficient here

Here we can see that every function has the same structure, and here we are violating the dry principle.

Dry Principle says that do not repeat yourself.

While writing a code just try to write generic code wherever possible , so that you do not need to repeat the same structure again and again. Now we will try to generalize this code. Let us try to solve this problem using a higher-order function

Higher Order Function

A higher order function is a function that can take one or more functions as arguments, or returns a function as its result or both.

Solution of coding question Using Higher Order Function

Below is the javascript for the above coding question using a higher-order function.

```
let myRadiusArray = [2, 3, 4, 5, 8]

function circleArea(radius){
   return Math.PI * radius * radius;
}
function circleCircumference(radius){
   return 2 * Math.PI * radius;
}
function circleDiameter(radius){
   return 2 * radius;
}

function calculate(radiusArr, logic){
   let result = []
```

```
    for(let i = 0 ; i < radiusArr.length ; i ++ ){
        result.push(logic(radiusArr[i]))
    }
    return result
}

let finalAreas = calculate(myRadiusArray, circleArea)
console.log('This is area array => ', finalAreas)


let finalCircumferences = calculate(myRadiusArray,
circleCircumference)
console.log('This is Circumference array =>',
finalCircumferences)


let finalDiameter = calculate(myRadiusArray,
circleDiameter)
console.log('This is Diameter array =>', finalDiameter)
```

Is the function which we have created a pure function?

Yes , Because everytime we are creating a new array to store updated values and returning the new array and we are not mutating the original radius array!

So , Higher order functions or in general the concept of functional programming follows the pure function rule

Functional Programming methods for arrays

1. map, filter, and reduce methods for arrays in languages like JavaScript are examples of functional programming methods applied to arrays. These methods encapsulate common operations on arrays in a way that aligns with the principles of functional programming (immutability, pure functions, higher order functions) .

Map

1. Purpose: **The map method is used to transform each element in an array. It applies a function to each element and returns a new array containing the results of those function calls**.
2. Functional Aspect: It embodies the principle of immutability and transformation in functional programming. Instead of modifying the original array, it creates a new one with transformed values.

```
let arr = [1, 2, 3, 4, 5]
let SquareArr = []
for(let i = 0 ; i < arr.length ; i ++ ){
   SquareArr.push(arr[i] * arr[i])
}
console.log(SquareArr)
```

3. We can also write this code for finding squares within the function.

```
et arr = [1, 2, 3, 4, 5]
function squareArrFn(arr){
   let squareArr = []
   for(let i = 0 ; i < arr.length ; i ++ ){
       squareArr.push(arr[i] * arr[i])
   }
   return squareArr;
}
let squareArrFinal = squareArrFn(arr)
console.log(squareArrFinal)
```

4. For these questions, where we want to apply operations on every element of an array, then we can use the map method. ,map is a higher-order function which will not change the original array. There is an inbuilt loop in a map which will take array elements one by one.

```
let arr = [1, 2, 3, 4, 5]
let squaredValues = arr.map(function(num){
   return num * num;
})
console.log(squaredValues)
```

5. Using a map to find an area for the given radiuses in the form of an array

```
let radiusArr = [1, 2, 3, 4]
let areaArr = radiusArr.map(function(num){
   return Math.PI * num * num;
})
console.log(areaArr)
```

Question
You are given a transaction array treat the transaction amount in rupees, and convert those amounts into dollars

Solution:

We can use the map to apply similar operations to all the elements of an array.

```
const transactions = [1000, 3000, 4000, 2000, - 898, 3800, -
4500];
const inrtToUsd = 80;

let conversionToDollars = transactions.map(function(amount){
   return amount / inrtToUsd;
})
console.log(conversionToDollars)
```

Filter

1. Purpose: **The filter method is used to select elements from an array that meet a certain condition. It applies a function to each element and returns a new array containing only those elements**

that pass the test or meets the conditions mentioned in the function.

2. Functional Aspect: It promotes the concept of pure functions and immutability. The original array remains unchanged, and a new array is returned based on the condition.
3. Problem statement

   We are given an array of numbers that contains both even and odd numbers and we need an array which only contains the even numbers of the input array.

   Solution: If the remainder of the number on dividing it by 2 is zero(element % 2 == 0), then it is an even number. Here we can use the filter. Here we have created evenArray and used a filter on myArr, so the elements that satisfy the condition will only be added to the evenArray.

```
let myArr = [1, 2, 5, 7, 8, 2, 6, 9, 13, 17]

let evenArray = myArr.filter(function(num){
   return num % 2 == 0;
})
console.log(evenArray)
```

   Problem statement: You are given a transaction array, and use a filter to find the positive transaction amounts

```
const transactions = [1000, 3000, 4000, 2000, - 898, 3800, - 4500];
```

```
const transactions = [1000, 3000, 4000, 2000, - 898, 3800, -
4500];

let positiveValue = transactions.filter(function(amount){
   return amount > 0;
})
console.log(positiveValue)
```

Reduce

1. it's a powerful tool for transforming lists into single values, embracing the concepts of immutability and higher-order functions.
2. Problem statement - You are given an array of numbers and you need to calculate the sum of all the elements of an array.
3. Solution - We will define a variable sum, initialize it with 0, iterate over the array and add elements one by one to sum the variable.

```
let arr = [1, 2, 3, 4, 5]
let sum = 0
for(let  i = 0 ; i < arr.length ; i ++ ){
   sum = sum + arr[i]
}
console.log(sum)
```

4. Just like above we have reduced all the array elements into one value i.e. sum, so basically **reduce is used to reduce multiple elements into a single one.**

```
let arr = [1, 2, 3, 4, 5]
const totalSum = arr.reduce(function(accumulator,
currentValue){
   accumulator = accumulator + currentValue
   return accumulator
},0)

console.log("total using reduce",totalSum)
```

5. Here 0 written after } is the initialising value of accumalator, this means accumalator will be initiated with 0, and accumalator is used to store the sum and num is the current element of an array at every iteration and at every iteration, num is added to acc.