Functional JS

Created	@June 8, 2022 4:50 PM
Class	
Type	
Materials	
Reviewed	☐

First-Class Functions

A language has a feature called as `FirstClass Functions` if we can pass the functions around as values, i.e. we can pass a function as an argument to another function. In Javascript we have the concept of first-class functions as well that we commonly refer as `callbacks`.

Higher Order Functions

These are those functions that take other functions as input or return a function as output.

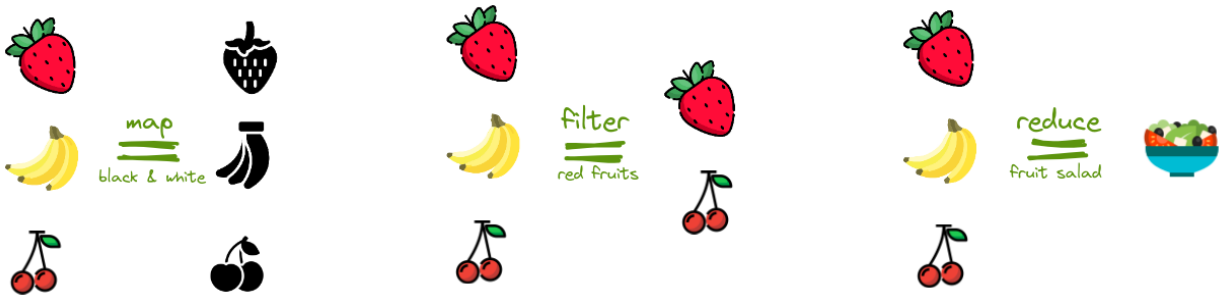
That means, the functions in which you're passing your callback is a higher order function.

```
function fun(x, fn) {
  console.log(x);
  fn(x*x);
}

fun(10, function gun(y) {
  console.log(y);
})
```

Here `gun` is a first-class function and `fun` is a higher order function.

There are some already in built higher order functions as well, like `forEach`, `map`, `reduce`, `filter` etc. that helps us in many ways like iterating over an array.



The above image gives a very good example of map, reduce and filter.

Filter Function

- It is a higher order function that means, it takes another function as an argument.
- Filter function is used to filter out values that is select few items of the array and reject the other.
- For example: In an array of integers, if we just want to return the even numbers so we can use filter function on the array.
- So, filter function takes a function as argument, the argument function is also called as `predicate` function.
- The main thing that `predicate` does is, it is applied one by one on every elements, and based on the logic written inside predicate, we have to return true/false.
- True denotes we have to keep the value in the final list, False denotes we have to reject the value.

```
let arr = [1,2,3,4,5,6,7,8,9];
const result = arr.filter(function logic(element) {
  //console.log(element);
  return element%2 == 0;
});
console.log(arr);
console.log(result);
```

Map function

- If we want to apply some logic to every element of the array and based on that logic get a new element, then we use map.
- This is a higher order function.

```
let arr = [0,1,2,3,4,5,6,7,8,9];
const result = arr.map(function logic(element) {
  //console.log(element);
  return element*10;
});
console.log(arr);
console.log(result);
```

- It always returns a new array.
- The logic callback we are passing will be applied on every single element of the array.

Reduce function

- This is also a higher order function, and the function that we pass as the callback to reduce is called as `reducer` function.
- This reducer function is applied on each element of the array, in the original order, passing in the return value form the calculation on preceding elements
- The reducer function takes two arguments, prev and curr. Prev is the value returned in the prev iteration, and curr is the value of current iteration.

Task: Try to implement your own filter function.

```
function myFilter(arr, predicate) {
  if(arr.length == 0) return [];
  const firstElement = arr[0];
  const filterOutput = predicate(firstElement) ? [firstElement] : [];
  const recursiveOutput = myFilter(arr.slice(1), predicate);
  return filterOutput.concat(recursiveOutput);
}

const newArr = myFilter([1,2,3,4,5], function pred(x) {
```

```
  return x%2 == 0;
})
```

Currying

Currying as a concept is closely related to closures. It introduces a new way of writing functions so that we can create more reusable functions. This concept applied directly on multi-argument functions.

Currying in a nutshell is like partially applying a function, by saying partially applying then we mean that we don’t use all the arguments at once, but instead lock up few of them for later use.

Currying breaks your multi-argument function into a series of single argument function.

```
function greet(greeting, name) {
  return greeting + name;
}
console.log(greet("hello", "sanket"));
console.log(greet("namaste", "sanket"));
console.log(greet("ciao", "sanket"));
console.log(greet("ciao", "karan"));

function curryGreet(greeting) {
  return function (name) {
    return greeting + name;
  }
}
const helloGreeting = curryGreet("hello");
console.log(helloGreeting("sanket"));
console.log(helloGreeting("karan"));
console.log(curryGreet("ciao")("sanket"));
```

So the greet function has been refactored into a curryGreet function, where we are returning a function from it. The internal function due to closure, remembers the variable greeting which is in the scope of outer function, and then whenever we have to call someone with hello, we can just call the helloGreeting variable.

Due to the fact that we can return function, we can make a reusable unit of code with the concept of currying. It is more or less like a code writing pattern, that based on the use case, help us to write better code.

function composition

In simple words, function composition refers to the concept where we circuit multiple functions one after another where one functions output will be used as other functions input.

Ex: f(g(x))

```
fun(gun(x));
```

Pure vs Impure Functions

A pure function is a function whose output is predictable, in the sense that it does only the expected intended task with the passed arguments, otherwise it is an impure function.

```
function greet(name) { // pure function
  return "hello " + name;
}

let givenName = "sanket";
function greet2() { // impure function
  return "hello " + givenName;
}
givenName = "";
greet2();
```