Lecture 22 | Higher Order Functions

Quick Revision of `map()` and `filter()`

- map() iterates over each element of the array and returns a new array with the results of the callback function
- filter() returns true false for each element of the array and returns a new array with the elements that return true

```
// map filter example
```

Question 01

- Get the number of products whose price is atleast 100.

```
const products = [
  { name: "T-Shirt", price: 25 },
  { name: "Headphones", price: 125 },
  { name: "Keyboard", price: 75 },
  { name: "Monitor", price: 200 },
];
// hint : chaining map and filter
```

> Answer:

```
let productsAbove100 = products
  .map(function (product) {
    return product.price;
  })
  .filter(function (price) {
    return price >= 100;
  });
console.log(productsAbove100.length); // 2
```

Question 02

- Get the movie Names from this Array of Objects method and only get the movie name which has rating higher than or equal to 8
- Hint : Use Filter and map chaining to achaive the objective

```
var newReleases = [
  {
```

```
    id: 1,
    title: "Die Hard",
    rating: 9,
  },
  {
    id: 2,
    title: "Bad Boys",
    rating: 7,
  },
  {
    id: 3,
    title: "The Chamber",
    rating: 10,
  },
];
```

> Solution using filter and map chaining

```
let highRatedMovies = newReleases
  .filter(function (newRelease) {
    return newRelease.rating >= 8;
  })
  .map(function (movie) {
    return movie.title;
  });
console.log(highRatedMovies);
```

> Output:

```
[ 'Die Hard', 'The Chamber' ]
```

Question 03

- Map Question
    - 1. You have to use map function and have to get all the students name in upperCase
- Use filter method to approach the problem
    - 2. Retrieve the details of students who scored more than 50 marks and have id greater than 120 from studentRecord

```
let studentRecords = [
  { name: "Abhishek", id: 123, marks: 98 },
  { name: "Udai", id: 101, marks: 90 },
  { name: "Himanshu", id: 200, marks: 96 },
  { name: "Mrinal", id: 115, marks: 75 },
];
```

- Solution 1

```
let upperStudents = studentRecords.map(function (student) {
  return student.name.toUpperCase();
});
console.log(upperStudents);
```

> Output :

```
[ 'ABHISHEK', 'UDAI', 'HIMANSHU', 'MRINAL' ]
```

- Solution 2

```
let highScoreStudents = studentRecords.filter(function (student) {
  return student.marks > 50 && student.id > 120;
});
console.log(highScoreStudents);
```

> Output :

```
[
  { name: 'Abhishek', id: 123, marks: 98 },
  { name: 'Himanshu', id: 200, marks: 96 }
]
```

Next Higer Order Function | `reduce()`

- `reduce()` reduces the array to a single value

> 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. The final result of running the reducer across all elements of the array is a single value. The first time that the callback is run there is no "return value of the previous calculation". If supplied, an initial value may be used in its place. Otherwise the array element at index 0 is used as the initial value and iteration starts from the next element (index 1 instead of index 0).

- Perhaps the easiest-to-understand case for `reduce()` is to return the sum of all the elements in an array:

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
```

```
const initialValue = 0;
const sumWithInitial = array1.reduce(
  (previousValue, currentValue) => previousValue + currentValue,
  initialValue
);

console.log(sumWithInitial);
// expected output: 10
```

> The reducer walks through the array element-by-element, at each step adding the current array value to the result from the previous step (this result is the running sum of all the previous steps) — until there are no more elements to add.

Syntax

```
// Arrow function
reduce((previousValue, currentValue) => {
  /* ... */
});
reduce((previousValue, currentValue, currentIndex) => {
  /* ... */
});
reduce((previousValue, currentValue, currentIndex, array) => {
  /* ... */
});
reduce((previousValue, currentValue, currentIndex, array) => {
  /* ... */
}, initialValue);

// Callback function
reduce(callbackFn);
reduce(callbackFn, initialValue);

// Inline callback function
reduce(function (previousValue, currentValue) {
  /* ... */
});
reduce(function (previousValue, currentValue, currentIndex) {
  /* ... */
});
reduce(function (previousValue, currentValue, currentIndex, array) {
  /* ... */
});
reduce(function (previousValue, currentValue, currentIndex, array) {
  /* ... */
}, initialValue);
```

Example

```
const array1 = [1, 2, 3, 4];

let addition = array1.reduce(function (sum, value) {
  let updatedSum = sum + value;
  return updatedSum;
});
console.log(addition); // expected output: 10

let multiplication = array1.reduce(function (product, value) {
  let updatedProduct = product * value;
  return updatedProduct;
});
console.log(multiplication); // expected output: 24
```

> Output :

```
> node reduce.js
10
24
```

Question 01 - Reduce

- From the Transactions Array filter out positive Elements and Calculate the total amount
- Use filter and Reduce to Achieve this

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

let totalPositiveTransactions = transactions
  .filter(function (transaction) {
    return transaction > 0;
  })
  .reduce(function (sum, value) {
    return sum + value;
  });
console.log(totalPositiveTransactions); // expected output: 13800
```

> Output :

```
> node reduce.js
13800
```

Reduce Right | `reduceRight()`

- `reduceRight()` - it just like `reduce()` but it goes from right to left

```
const arr = [1, 2, 3, 4];
let sum = arr.reduceRight(function (sum, value) {
  let updatedSum = sum + value;
  console.log(updatedSum); // 7, 9, 10
  return updatedSum;
});
console.log(sum); // expected output: 10
```

> Output :

```
> node reduceRight.js
7
9
10
10
```

`forEach()` and `findIndex()`

- `forEach()` - it is used to iterate over the array and execute a callback function on each element of the array
- `findIndex()` - it is used to find the index of the first element in the array that satisfies the provided testing function

Object Oriented JavaScript | `this` keyword

- `this` keyword is used to access the current object in the context of the function it is used in.

> Js has two modes strict and non-strict

```
"use strict"; // strict mode
a = 10; // error
console.log(a); // not defined
```

> In non-strict mode, the `this` keyword refers to the global object. In strict mode, the `this` keyword refers to the object that is currently executing the function.

- Js has two environments:
    - Browser
    - Node

Context : `this` in `Node` Environment with `non strict mode`

- `this` in global area returns empty object {}

```
console.log(this); // {} empty object
```

Output :

```
> node this.js
{}
```

- `this` in function area returns global object

```
function f() {
  console.log(this); // function object : {}
}
f();
```

Output :

```
> node this.js
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: Performance {
    nodeTiming: PerformanceNodeTiming {
      name: 'node',
      entryType: 'node',
      startTime: 0,
      duration: 231.9817570000887,
      nodeStart: 50.90881999954581,
      v8Start: 82.18204599991441,
      bootstrapComplete: 200.12776999920607,
      environment: 133.09282599948347,
      loopStart: -1,
      loopExit: -1,
      idleTime: 0
    },
    timeOrigin: 1644687300184.629
  },
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
```

```
  }
}
```

- `this` in object in function area returns object itself

```
let obj = {
  a: 10,
  f: function () {
    console.log(this); // obj
  },
};
obj.f();
```

> Output :

```
{ a: 10, f: [Function: f] }
```

- Function inside Function inside Object | `this` keyword returns the global object

```
let obj = {
  a: 10,
  f: function () {
    let f = function () {
      console.log(this); // obj
    };
    f();
  },
};
obj.f();
```

```
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: Performance {
    nodeTiming: PerformanceNodeTiming {
      name: 'node',
      entryType: 'node',
      startTime: 0,
      duration: 81.35366600006819,
```

```
      nodeStart: 4.461749998852611,
      v8Start: 8.080905999988317,
      bootstrapComplete: 42.207240998744965,
      environment: 26.03384899906814,
      loopStart: -1,
      loopExit: -1,
      idleTime: 0
    },
    timeOrigin: 1644687676750.476
  },
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}
```

Node Environment | `this` keyword in `Node` Environment with `strict mode`

```
// Node + strict mode
"use strict";

// global context
console.log(this); // {} empty object

// function context
function f() {
  console.log(this); // function object : {}
}
f(); // undefined

// object context
let obj = {
  a: 10,
  f: function () {
    console.log(this); // obj
  },
};
obj.f(); // obj itself

// object context 2
let obj2 = {
  a: 20,
  f: function () {
    function g() {
      console.log(this);
    }
    g();
  },
};
obj2.f(); // undefined
```

Browser Environment | `this` keyword in `Browser` Environment with `non strict mode`

```
// Browser + Non Strict Mode

// Global Context
console.log(this); // Window object gets returned

// Function Context
function f() {
  console.log(this); // Window object gets returned
}
f();

// Object Context
let obj = {
  a: 2,
  b: function () {
    console.log(this); // Object gets returned
  },
};
obj.b(); // Object itself gets returned

// Object Function Function Context
let obj2 = {
  a: 2,
  b: function () {
    function f() {
      console.log(this); // Window object gets returned
    }
    f();
  },
};
obj2.b(); // Window object gets returned
```

Browser Environment | `this` keyword in `Browser` Environment with `strict mode`

```
// Browser + Strict Mode

"use strict";

// Global Context
console.log(this); // Window object gets returned

// Function Context
function f() {
  console.log(this); // undefined gets returned
```

```
}
f();

// Object Context
let obj = {
  a: 2,
  b: function () {
    console.log(this); // Object gets returned
  },
};
obj.b(); // Object itself gets returned

// Object Function Function Context
let obj2 = {
  a: 2,
  b: function () {
    function f() {
      console.log(this);
    }
    f();
  },
};
obj2.b(); // undefined gets returned
```

Table of values of `this` keyword in different environments and different modes

- Node Environment

`this` Context	Non Strict Mode	Strict Mode
Global Context	Empty Object {}	Empty Object {}
Function Context	Global Object {}	undefined
Object Context	Object itself	Object itself
Object Function Function Context	Global Object	undefined

- Browser Environment

`this` Context	Non Strict Mode	Strict Mode
Global Context	Window Object	Window Object
Function Context	Window Object	undefined
Object Context	Object itself	Object itself
Object Function Function Context	Window Object	undefined