COURSEWARE

Higher Order Functions

Contents

Overview

A **higher order function** is a function that takes a function as an argument, or returns a function.

Tutorial

Higher order functions is in contrast to first order functions, which do not take a function as an argument or return a function as output.

The following example of a first-order function filters all 4-letter words from a list:

```
const censor = words => {
    const filtered = [];
    for(let i = 0; i < words.length; i++) {
        const word = words[i];
        if(word.length !== 4) filtered.push(word);
    }
    return filtered;
};

censor(['zonk','army','shout','sun']);
// Output: ['shout','sun']
```

Now, what if we want to select all the words that begin with 's'? We could create another function...

```
const startsWithS = words => {
    const filtered = [];
    for( let i = 0; i < words.length; i++ ) {
        const word = words[i];
        if(word.startsWith('s')) filtered.push(word);
    }
    return filtered;
};

startsWithS(['zonk','army','shout','sun']);
// Output: ['shout' , 'sun']
```

However, this results in significant code repetition.

There's a pattern forming here which we can identify and abstract into a more generalised solution.

Luckily, JavaScript has first class functions. Just like numbers, strings, or objects, functions can be:

- Assigned to an identifier (variable) value
- Assigned to object property values
- Passed as arguments
- Returned from functions

All in all, we can use functions just like any other bits of data in our programs, which in turn makes abstraction a lot easier.

For instance, we can create a function that abstracts the process of iterating over a list and accumulating a return value by passing in a function that handles *the bits that are different*.

We'll call that function the *reducer*:

```
const reduce = (reducer, initial, arr) =>{
    //shared stuff
    let acc = initial;

    for(let i=0; i<arr.length; i++) {
        //unique stuff in reducer() call
        acc = reducer(acc,arr[i]);

        // more shared stuff
    }
    return acc;
}

reduce((acc, curr) => acc + curr, 0, [1,2,3]);
// Output : 6
```

This `reduce()` implementation takes a reducer function, an initial value for the accumulator, and an array for the data to iterate over.

For each item in the array, the reducer is called, passing it in the accumulator and the current array element. The return value is assigned to the accumulator. When it's finished applying the reducer to all of the values in the list, the accumulated value is returned.

With the iteration and value accumulation abstracted, now we can implement a more generalised `filter()` function:

```
const filter = (fn,arr) => reduce((acc,curr) => fn(curr) ? acc.concat([curr]) :
acc, [], arr);
```

In the `filter()` function, everything is shared except the `fn()` function that gets passed in as argument. That `fn()` argument is called a **predicate**. A predicate is a function that returns a boolean value.

We call `fn()` with the current value, and if the `fn(curr)` test returns true, we concat the `curr` value to the accumulator array. Otherwise, we just return the current accumulator value.

Now we can implement `censor()` with `filter` to filter out 4-letter words:

```
const censor = words => filter(word => word.length !== 4, words);
```

Pretty neat right? We can apply the same to `startsWithS()`:

```
const startsWithS = words => filter(word => word.startsWith('s'), words);
```

Let's have a look at another example.

Assume we have a list of grades from a classroom. Our classroom has 5 girls, and 5 boys, and each of them has a grade between 0 and 20:

```
let grades = [
    {name: 'John', grade: 8, sex: 'M'},
    {name: 'Sarah', grade: 12, sex: 'F'},
    {name: 'Bob', grade: 16, sex: 'M'},
    {name: 'Johnny', grade: 2, sex: 'M'},
    {name: 'Cyrus', grade: 4, sex: 'M'},
    {name: 'Paula', grade: 18, sex: 'F'},
    {name: 'Jeff', grade: 5, sex: 'M'},
    {name: 'Jennifer', grade: 13, sex: 'F'},
    {name: 'Courtney', grade: 15, sex: 'F'},
    {name: 'Jane', grade: 9, sex: 'F'}
]
```

Lets say we want to find out:

- The average grade of the classroom
- The average grade of the boys
- The average grade of the girls
- The higher note among the boys
- The higher note among the girls.

We will try to use higher-order functions to get a program that is simple and easy to read.

```
let isBoy = student => student.sex === "M";
let isGirl = student => student.sex === "F";

let getBoys = grades => (grades.filter(isBoy));
let getGirls = grades => (grades.filter(isGirl));

let average = grades => (grades.reduce((acc,curr) => (acc + curr.grade), 0) /
grades.length);

let maxGrade = grades => (Math.max(...grades.map(student => student.grade)));
let minGrade = grades => (Math.min(...grades.map(student => student.grade)));
```

Here we have 7 functions, and each of them only has one job.
Now we have what we need to write higher-order functions:

```
let classRoomAverage = average(grades);
let boysAverage = average(getBoys(grades));
let girlsAverage = average(getGirls(grades));
```

Exercises

1. Following on from the previous example, write the code to calculate - ensuring that higher-order functions are used:
    - Highest Grade
    - Lowest Grade
    - Highest Grade of Boys
    - Highest Grade of Girls
    - Lowest Grade of Boys
    - Lowest Grade of Girls

▶ Solution