WEB230: JavaScript 1

Module 3: Higher-Order Functions

Abstraction

- Hide programming details
- Talk about problems at a higher (or more abstract) level

Abstracting Repetition

- Let's log each element in a sequence:

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

- With abstraction, this becomes:

```
repeatLog(numbers);
```

- But then we have to define the `repeatLog()` function:

```
function repeatLog(n) {
    for (let i = 0; i < n; i++) {
        console.log(i);
    }
}
```

- This makes the code more readable
- Maybe we need to do other things than just logging
- We can update our function to "do something"
- In this case "doing something" can be represented as a function
    - Remember: functions are just values
- Let's pass our action as a function value

```
function repeat(n, action) {
    for (let i = 0; i < n; i++) {
        action(i);
    }
}

repeat(3, console.log);
```

- We can also create a function on the spot
- Pass it instead of using a predefined function

```
let labels = [];
repeat(5, i => labels.push(`Unit ${i + 1}`) );

console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

Higher-Order Functions

- Functions that operate on other functions
    - By taking them as arguments OR
    - By returning them
- When we pass a function to another function, the function we pass is known as a **Callback Function** or just a **Callback**
- We can have functions that create new functions.

```
function greaterThan(n) {
   return function(m) { return m > n; };
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
```

Built-in Array Method

- Arrays have a built-in method `.forEach()` that will act on each element of the array

```
let letters = ["A", "B"];
letters.forEach(el => console.log(el));
```

Filtering an Array

- In JavaScript we can use an array method called `.filter()`.
- `.filter()` takes the array and builds up a new array with only the elements that pass the test
- It is a "**pure**" function, it does not modify the array it is given

```
let nums = [3,5,7,9];

console.log( nums.filter( num => num%3 === 0 ) );
```

- Filter will loop through the array
    - use our function on each item in it
    - taking the array element as the argument
- It will check each item to make sure it meets our condition
- If the function returns `true` the element is included in the new array

Transforming With Map

- The `.map()` method transforms an array by applying a function to all of its elements and building a new array
- The new array will have the same length as the input array

- It's content will be “mapped” to a new form
- The function takes one element as a parameter and returns a new element
- Suppose I wanted an array of the square of each number in an array
- We use the map function to create this new array

```
let nums = [3,5,7,9];

console.log( nums.map( num => num*num) );
```

- .map() is a standard method on arrays
- the new array will always be the same size as the original array

Summarizing With Reduce

- Another common pattern is to compute a single value from an array
- Let's find the sum of all the numbers in an array
- The higher-order operation that represents this pattern is called "reduce"
- When summing numbers, you’d start with the number zero and, for each element, add the current number
- .reduce() takes a function to act on each element and a starting value
- If your array contains at least one element, and the first element can be used as the initial value, you can leave off the start value

```
let nums = [3,5,7,9];

console.log(nums.reduce( (sum, num) => sum + num, 0 ) );
console.log(nums.reduce( (sum, num) => sum + num ) );
```

Sorting (not in book)

- We can sort arrays **in place** using the .sort() method
- By default, it will sort elements lexicographically (dictionary order)

```
let pets = ["dog", "hamster", "cat"];

console.log( pets.sort() );
```

- by default, .sort() sorts strings
- it doesn't work with numbers (or other data types)

```
let nums = [7,3,9,5,11,31];

console.log( nums.sort() ) );
```

- you can pass a function that will determine order - useful for sorting values that are not strings

```
let nums = [7,3,9,5,11];

console.log( nums.sort((a,b) => a-b) );
```

- The function takes two arguments which will be two elements of the array
- Your function should return: - a negative number (or 0) if the first comes first - a positive number if the second comes first

Composability

- Higher-order functions start to shine when you need to compose functions
- You can combine the output of one method as the input of another

```
let nums = [3,5,7,9];

console.log( nums.filter( num => num%3 === 0 )
        .reduce( (sum, num) => sum + num ) );
```

Summary

- Being able to pass function values to other functions is a very useful aspect of JavaScript
- It allows us to write computations with “gaps” in them
- Functions fill in these gaps by providing functionality
- Arrays provide a number of useful higher-order methods
    - `.forEach()` to do something with each element in an array
    - `.filter()` to build a new array with some elements filtered out
    - `.map()` to build a new array where each element has been put through a function
    - `.reduce()` to combine all of an array’s elements into a single value
    - `.sort()` sort the array in place