TASK

Higher-Order Functions and Callbacks

Visit our website

Introduction

WELCOME TO THE HIGHER-ORDER FUNCTIONS AND CALLBACKS TASK!

In this task, you will learn about two fundamental concepts in JavaScript - **Higher-order functions (HOF)** and **callbacks**. These concepts are closely tied together, as you cannot have a callback function without a higher-order function. Both concepts are essential to understanding more advanced concepts in JavaScript, which will be introduced later in this bootcamp.

WHAT ARE HIGHER-ORDER FUNCTIONS?

Higher-order functions (HOF) are functions that take *a function* as *an argument* and/or *return* a function. They are a way to define reusable functionality and a less tedious alternative to class inheritance. Higher-order functions enable functional programming in JavaScript. In order to understand this concept well, it is beneficial to understand what functional programming is, and importantly the concept of first-class functions.

WHAT IS FUNCTIONAL PROGRAMMING?

Functional programming is a programming paradigm in which you operate on functions as you would with any other kind of value. This makes functions in functional programming "first-class", meaning they are just like any other value you work with. Higher-order functions and callbacks are great examples of how you can treat functions as values.

First-class functions

As stated above, JavaScript treats functions as first-class objects. Functions in JavaScript are values. That means that they can be assigned to variables and they can also, therefore, be passed as arguments because they are values (object). This is the most important thing to remember when working with higher-order functions.

In JavaScript, functions are a special kind of object called Function objects. We can assign functions as values to variables *because* they are objects.

Let us look at the following example:

```
// This is a higher order function (HOF)
function higherOrder() {
  // Arrow function returns the greeting "Hello, World!"
  return (greeting = () => "Hello, World!"); /* return the function called
  greeting (N.B. the function itself, NOT the result of calling the function)
  */
}

// Call (HOF) and store the greeting function in a variable
let useGreeting = higherOrder();

/* Call the function stored in useGreeting and log the output to the console.
This is effectively the same as having called the greeting function */
console.log(useGreeting());
// Output: Hello, World!
```

Note, in the example, how the `useGreeting` variable is assigned the function called `higherOrder()`. Because it is a function, when you use `useGreeting()` you call it with parentheses as you would call `higherOrder()`. Also note that in the example above, we use an arrow function *without* the `let` keyword, as was demonstrated in the introduction to arrow functions. If you're at all confused by this, revise arrow functions briefly at this point.

BUILT-IN HIGHER-ORDER FUNCTIONS

JavaScript actually already has a bunch of built-in higher-order functions that make our lives easy. The Array class has built-in methods that take functions as arguments such as the `map()`, `filter()`, and `reduce()` methods.

Let's look at an example of how the `map()` method can be used. Let's say we have an array of numbers. We want to create a new array that will contain the doubled values of the first one. Let's see how we can solve this problem with and without a higher-order function.

Without a HOF

Conventionally, we could solve the problem as illustrated overleaf with an empty array and a loop that does the calculation on each element and pushes the values to the empty array:

```
function doubleAllValues() {
  // Define the first array to store the single values
  const firstArray = [1, 2, 3, 4];

  // Define the second array (empty array) to store the doubled values
  const secondArray = [];

  // Loop through the length of the first array
  for (let i = 0; i < firstArray.length; i++) {
    /* Multiply each value in the first array by 2 and push it into the
    second array */
    secondArray.push(firstArray[i] * 2);
  }
  // Return the doubled values stored in the second array
  return secondArray;
}
// Call the function and log the results in the console
console.log(doubleAllValues());
// Output: [ 2, 4, 6, 8 ]
```

With a HOF

Alternatively, we can use the `map()` method to create a much cleaner solution by passing a function as an argument to the map method:

```
function doubleAllValues() {
  // Define the first array to store the single values
  const firstArray = [1, 2, 3, 4];

   /* Arrow function takes the argument "item". Each item is an element in the
first array that is multiplied by 2 */
```

```
  const multiplyByTwo = (item) => item * 2;

   /* Map each element in the first array by applying the multiplyByTwo
function. Define the second array to store the doubled values. */
  const secondArray = firstArray.map(multiplyByTwo);

  // Return the new array
  return secondArray;
}
// Call the function and log the results in the console
console.log(doubleAllValues());
// Output: [ 2, 4, 6, 8 ]
```

CREATING A HIGHER-ORDER FUNCTION

Let's create our own higher-order function that mimics the behaviour of the built-in map method so that we get a better understanding of how this works. Our higher-order function – let's call it `myMapper()` – will take an array of strings and a function as an argument. `myMapper()` will then apply the function to every element in the array passed as an argument, and return a new array. **When we pass a function as an argument we call that a callback function**. Also note: a function without a name is called an anonymous function. Read more about anonymous functions **here**.

Take a look at the code for `myMapper()`:

```
const wordsArray = ["Express", "JavaScript", "React", "Next"];

/* myMapper(HOF) takes an array (arr) and hypothetical function (fn) as
arguments */
let myMapper = (arr) => (fn) => {
  const arrayAfterMapping = [];
  for (let i = 0; i < arr.length; i++) {
    // The callback function fn() is applied to each element
    arrayAfterMapping.push(fn(arr[i]));
  }
  return arrayAfterMapping; // Return a new array
};
```

```
/* myMapper(HOF) is used with an anonymous function to return the length of
each word followed by a space */
const outputArray = myMapper(wordsArray)((item) => item.length + " ");

// Output the result to the console
console.log("Length of words: " + outputArray);
// Output: Length of words: 7, 10, 5, 4
```

WHAT ARE CALLBACK FUNCTIONS?

Now that you understand what higher-order functions are, let’s explore callback functions. We will first be looking at the definition of the term. The authorities on the matter, **MDN**, explain: “A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.”

In other words, a callback function is a function definition that is passed in as the argument of another function’s invocation. The callback will never run until the encompassing function executes it when the time comes. Therefore, the callback is at the mercy of the function receiving it.

Let’s revise the concepts covered earlier to ensure they are fresh in your mind before we move on to callbacks. In JavaScript, functions – just like everything else – are objects. Because of this, functions can take other functions as arguments and can be returned by other functions. Functions that do this are called higher-order functions. **Any function that is passed as an argument** is called a **callback function**.

WHAT DO WE NEED CALLBACKS FOR?

We need callbacks for one very important reason – JavaScript is an event-driven language. This means that instead of waiting for a response before moving on, JavaScript will keep executing while listening for other events. Examples of such events could be button clicks or even responses from external servers that are serving data to be used by your application.

JavaScript code runs from top to bottom, meaning that code runs sequentially. There are times, however, when we do not want code to run sequentially, but rather have certain code executed only when another task is completed or an event takes place. This is called asynchronous execution.

In the code below, you can see the behaviour that you are familiar with up to this point. We have two functions defined, and when we call these functions they execute sequentially from top to bottom as they are called by the program. That means that `first()` will be executed before `second()` and generate the expected output.

```
// Define two functions which execute sequentially
function first() {
  console.log(1);
}
function second() {
  console.log(2);
}

// Call the functions
first();
second();
// Output:
// 1
// 2
```

Let's now look at a very simple example of how we can use callbacks to change code so that it does not execute sequentially. What if we put code in the function called `first()` that can't be executed immediately? For example, we could be making a request to an external server for data to be used in our application, but the server will take a while to respond with the data, delaying the execution of `first()`.

To simulate this scenario without actually making a request from an external source, we are going to use `setTimeout()` which is a JavaScript function that calls another function after a set amount of time. We'll delay our function for 5000 milliseconds (i.e., 5 seconds; we work in milliseconds because that is what the `setTimeout()` function expects as input) to simulate an API request. Our new code will look like this:

```
// Define first function with a delay
function first() {
  // Use an anonymous function as a callback for the setTimeout
  setTimeout(function () {
    console.log(1); // Body of the anonymous function
  }, 5000); // Delay is 5000ms for the second argument
}

// Define a second function without a delay
function second() {
  console.log(2);
}

// Call the functions in the same order as before
first();
second();
// Output:
// 2
// 1
```

When we execute the code above, we will get a different result than before because of the delay in executing the code in the first function.

This does not, however, mean that all callback functions are asynchronous. Callbacks can also be used synchronously, i.e. the code is executed immediately. This is often referred to as "blocking", where the higher-order function doesn't complete its execution until the callback is done executing.

WAYS OF CREATING AND USING CALLBACK FUNCTIONS

There are multiple ways to create and use callback functions that we can look at briefly. As you will come across callbacks being used in each of these ways in online resources and existing codebases that you may contribute to in the future, it is a good idea to review them.

There are three ways that we can use callbacks. Let's take a look at an example of each of these ways.

Anonymous functions

Remember the earlier definition: a function without a name is called an anonymous function. Let's take a look at an example.

```
/* This is an anonymous function (function without a name) created and passed
as an argument at the same time */
setTimeout(function () {
  console.log("This line is returned after 5000ms"); /*Anon. function body */
}, 5000);
```

Arrow functions

```
/* This is an arrow function expression of an anonymous function as this also
has no name. It is used similarly to the anonymous function as it is created
and passed as an argument at the same time */
setTimeout(() => {
  console.log("This line is returned after 5000ms");
}, 5000);
```

Defined functions passed as an argument

```
// Assign a function to a variable
let callback = function () {
  console.log("This line is returned after 5000ms");
};

// Pass the callback variable as the callback function to setTimeout
setTimeout(callback, 5000);
```

Instructions

Compulsory Task 1

For this Compulsory Task, you will be making your own higher-order function that mimics the behaviour of a built-in method that already exists (similar to what we did earlier when we created our own implementation of the built-in function `map()`, which we called `myMapper()`).

Follow these steps:

- You're going to be creating your own filter function that is similar to the built-in `filter()` method in JavaScript. The first thing you need to do is to have a look **here** to familiarise yourself with the built-in `filter()` method. Once you have the basic idea, you can move on to the next step.
- Create a higher-order function named `myFilterFunction()` in a file named `higherOrder.js` (do not use the built-in function at all - the objective is to create a different filter function of your own).
- Your filter function should take the following two arguments:
  - An array of strings with 10 words, where at least three of the words have six letters.
  - A callback function that returns a boolean based on whether or not a word has six letters.
- `myFilterFunction()` should return a new array that contains only the words that are six letters long and no other words.

Remember: you **may not use** the **built-in filter** method to complete this task!

Compulsory Task 2

For this Compulsory Task, you will need a comprehensive understanding of both the `setInterval()` and the `clearInterval()` functions, and how they can be used.

Resources for `setInterval()`:

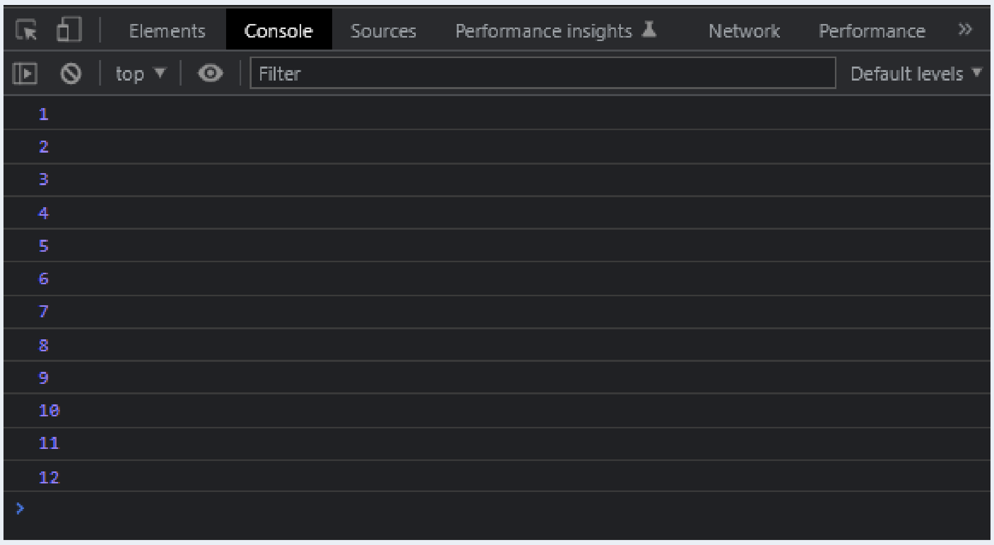
- **https://developer.mozilla.org/en-US/docs/Web/API/setInterval**
- **https://www.w3schools.com/jsref/met_win_setinterval.asp**

Resources for `clearInterval()`:

- **https://developer.mozilla.org/en-US/docs/Web/API/clearInterval**
- **https://www.w3schools.com/jsref/met_win_clearinterval.asp**

Follow these steps:

- Follow the instructions in the **callBack.js** file that can be found in the task folder.
- When the start button is clicked, your program should output a number to the console every 1000ms, starting from 1 and incrementing the output by 1 every time. The output should look as follows:



```
1
2
3
4
5
6
7
8
9
10
11
12
>
```

- When the stop button is clicked, your program should stop all output to the console.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.
Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.