
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 (or HOFs) and callbacks. These concepts are closely tied together, as you cannot have a callback function without a higher-order function. By mastering HOFs and callbacks, you will be able to write dynamic and responsive code, which is key to becoming proficient in advanced JavaScript programming.

## WHAT ARE HIGHER-ORDER FUNCTIONS?

Higher-order functions are functions that take a function as an argument and/or return a function. They are a way to define reusable functionality and are fundamental to 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 way of writing code where functions are treated like any other value. This means they can be assigned to variables, passed as arguments, and returned from other functions. Higher-order functions and callbacks are key examples of this concept in action.

### First-class functions

JavaScript treats functions as **first-class citizens**. Functions are special values that can be used like any other value, and as stated above, this means you can assign them to variables, pass them as arguments to other functions, and even return them from functions. This is a very powerful concept that allows you to create more flexible and reusable code.

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() {
  // Regular function returns the greeting "Hello, World!"
  function greet() {
    return "Hello, World!";
  }

  // Return the function itself
  return greet;
}

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

// Call the function stored in useGreeting and log the output to the console
console.log(useGreeting()); // Output: Hello, World!
```

Note in the example above how the `useGreeting` variable is assigned the function `higherOrder()`. Because it is a function, when you use `useGreeting()` you call it with parentheses as you would call `higherOrder()`. Also note that we define a regular function within the `higherOrder` function, which is then returned. This nested function is not explicitly declared with `let` or `const` because it's a local variable within the `higherOrder` function.

## BUILT-IN HIGHER-ORDER FUNCTIONS

JavaScript actually already has a bunch of built-in higher-order functions that make our lives easier. 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 an HOF**

Conventionally, we could solve the problem as illustrated below 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 an 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** (more on this in the following section). Also, note that a function without a name is called an **anonymous** function. Read here to learn more about **anonymous functions**.

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

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

// Higher-order function that takes an array and a callback function
const myMapper = (arr) => (fn) => {
  return arr.map(fn); // Use map to apply the callback to each element
};

// Use myMapper with an anonymous function to get the length of each word
const outputArray = myMapper(wordsArray)((item) => " " + item.length);

// Output the result
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. The authorities on the matter, **MDN**, describe it as: "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.

In JavaScript, functions are objects, which means they can have properties and methods like any other object. Because of this, functions can be passed around in the code – taking other functions as arguments, being returned from other functions, or even having properties of their own. This is why functions can serve as higher-order functions, which can take or return other functions as arguments, and also why functions passed as arguments are known as callback functions.

## 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 that 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 the amount of time it might take to request data from an external server. 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 both create and use callback functions, which you'll often encounter in online resources and existing codebases. There are three ways that we can use callbacks. Let's take a look at an example of each of them.

### Anonymous functions

```
/* 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");
}, 5000);
```

### Arrow functions

```
// This is an arrow function expression used as an anonymous callback
function
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);
```

# Practical task 1

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 **at this website** 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 ten 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!

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

# Practical task 2

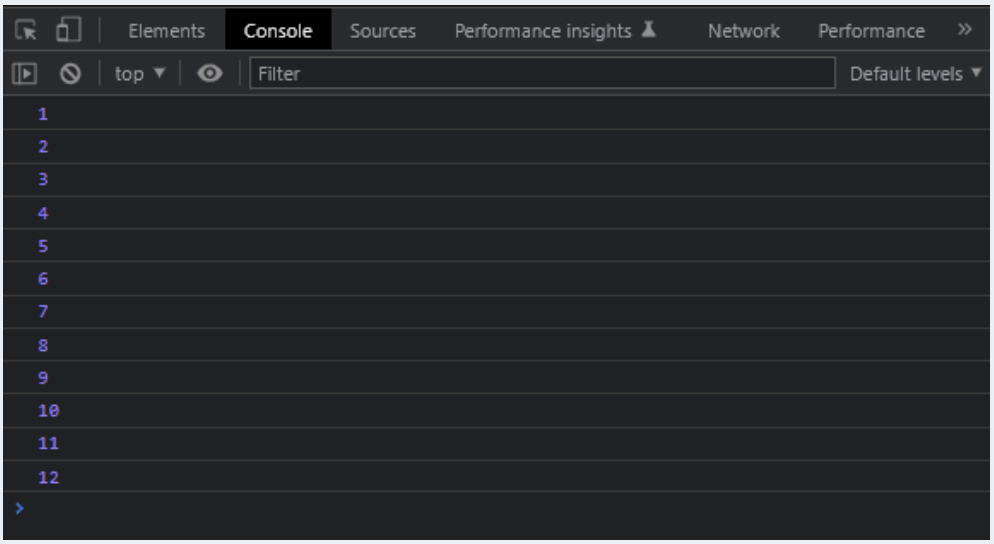
For this practical task, you will need a comprehensive understanding of both the `setInterval()` and the `clearInterval()` functions, and how they can be used.

Resources for `setInterval()` can be found on **this website**.

Resources for `clearInterval()` can be found on **this website**.

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 1000 ms, starting from 1 and incrementing the output by 1 every time. The output should look as follows:




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

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

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?

Share your thoughts anonymously using this **form**.