TASK

Programming with Callback Functions

Visit our website

Introduction

WELCOME TO THE CALLBACK FUNCTIONS TASK!

In this task we will take a closer look at callback functions which we have already touched on in the previous task about higher-order functions. The two concepts are closely tied together, as you cannot have a callback function without a higher-order function.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHAT ARE CALLBACK FUNCTIONS?

For us to get a good understanding of what callback functions are, we will first be looking at the definition of the term.

Let's quote the authorities on the matter, **MDN**: "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 - 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 execute only once another task is completed or an event takes place. This is called asynchronous execution which we will take a closer look at later in the bootcamp.

Let's look at a very simple example of how we can use callbacks to change code so that it does not execute sequentially.

```
function first(){
    console.log(1);
  }
  function second(){
    console.log(2);
  }
  first();
  second();
```

In the code above, we can see the behaviour that you are familiar with up to this point. We have 2 functions defined, and when we call these functions they execute sequentially from top to bottom when they are called. That means that **first()** would execute before **second()** and generate the expected output below:

```
// 1
// 2
```

But what if we had code in function **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.

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 a function after a set amount of time. We'll delay our function for 5000 milliseconds to simulate an API request. Our new code will look like this:

```
// define first function
function first(){
    /* Simulate a code delay
     use an anonymous function as a callback function
     passed as one of the arguments to the setTimeout function */

    setTimeout( function(){
      console.log(1);/*This is the entire body of the anonymous function - see
                       more about anonymous functions later*/
    }, 5000 );// the second argument is the delay i.e. 5000ms
  }

// define second function
function second(){
    console.log(2);
  }

/* we call the functions in the same order as before, but the
output will differ because of the delay*/
first();
second();
```

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.

```
// 2
// 1
```

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. It is good to be able to identify these different ways as you will come across callbacks being used in each of these ways in resources online, and also existing codebases that you may contribute to in the future.

The three ways that we can use callbacks are as follows:

- Anonymous functions
- Arrow functions

- Defined functions passed as an argument.

Let's take a look at an example of each of these ways:

```
/* This is an anonymous function (function with no name)
 created and passed as an argument at the same time */
setTimeout(function() {
    console.log("This is outputted after 5000ms") /* Remember this is the body
                                                   of the anonymous function*/
   }, 5000)
```

```
/* This is an arrow function. 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 output after 5000ms")
   }, 5000)
```

```
//assigning a function to a variable
let callbackToBePassedAsArgument = function() {
    console.log("This line is output after 5000ms");
}

/* we can now pass the variable callbackToBePassedAsArgument
as the callback function*/
setTimeout(callbackToBePassedAsArgument, 5000)
```

Compulsory Task 1

This task will need you to become familiar with both the setInterval() and the clearInterval() functions, and how they can be used.

Resources for setInterval:

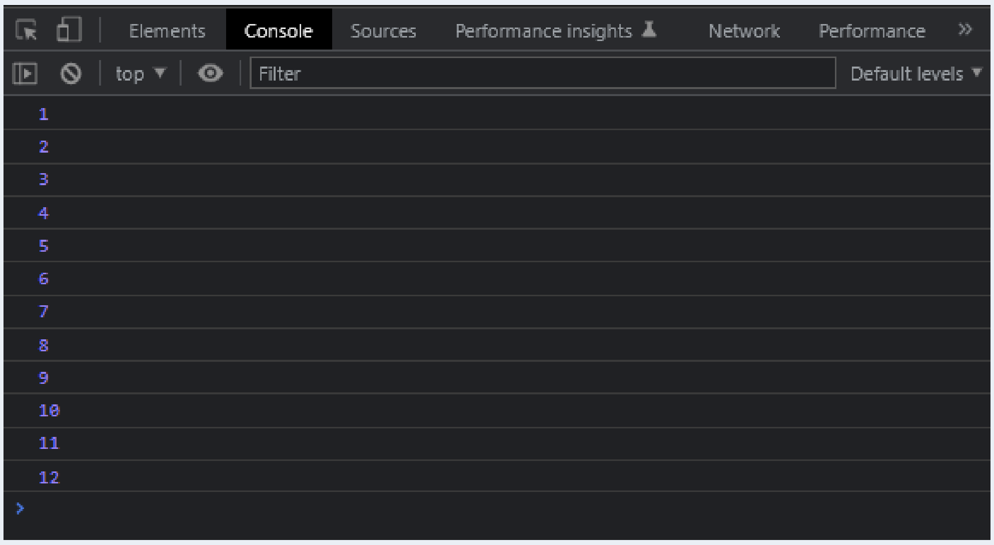
- **https://developer.mozilla.org/en-US/docs/Web/API/setInterval**
- **https://www.w3schools.com/jsref/met_win_setinterval.asp**

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 CompulsoryTask1 folder.
- When the start button is clicked, your program should output a number, starting from 1, to the console every 1000ms, 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.