JavaScript Callbacks

Summary: in this tutorial, you will learn about JavaScript callback functions including synchronous and asynchronous callbacks.

What are callbacks

In JavaScript, functions are first-class citizens. Therefore, you can pass a function to another function as an argument.

By definition, a callback is a function that you pass into another function as an argument for executing later.

The following defines a `filter()` function that accepts an array of numbers and returns a new array of odd numbers:

```
function filter(numbers) {
  let results = [];
  for (const number of numbers) {
    if (number % 2 != 0) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers));
```

How it works.

- First, define the `filter()` function that accepts an array of numbers and returns a new array of the odd numbers.
- Second, define the `numbers` array that has both odd and even numbers.

- Third, call the `filter()` function to get the odd numbers out of the numbers array and output the result.

If you want to return an array that contains even numbers, you need to modify the `filter()` function. To make the `filter()` function more generic and reusable, you can:

- First, extract the logic in the `if` block and wrap it in a separate function.
- Second, pass the function to the filter() function as an argument.

Here's the updated code:

```
function isOdd(number) {
  return number % 2 != 0;
}

function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers, isOdd));
```

The result is the same. However, you can pass any function that accepts an argument and returns a boolean value to the second argument of the `filter()` function.

For example, you can use the `filter()` function to return an array of even numbers like this:

```
function isOdd(number) {
  return number % 2 != 0;
}
function isEven(number) {
  return number % 2 == 0;
}
```

```
function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));
console.log(filter(numbers, isEven));
```

By definition, the `isOdd` and `isEven` are callback functions or callbacks. Because the `filter()` function accepts a function as an argument, it's called a *high-order function*.

A callback can be an anonymous function, which is a function without a name like this:

```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {
  return number % 2 != 0;
});

console.log(oddNumbers);
```

In this example, we pass an anonymous function to the `filter()` function instead of using a separate function.

In ES6, you can use an arrow function like this:

```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, (number) => number % 2 != 0);

console.log(oddNumbers);
```

There are two types of callbacks: synchronous and asynchronous callbacks.

Synchronous callbacks

A synchronous callback is executed during the execution of the high-order function that uses the callback. The `isOdd` and `isEven` are examples of synchronous callbacks because they execute during the execution of the `filter()` function.

Asynchronous callbacks

An asynchronous callback is executed after the execution of the high-order function that uses the callback.

Asynchronicity means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

Note that JavaScript is a single-threaded programming language. It carries asynchronous operations via the callback queue and event loop.

Suppose that you need to develop a script that downloads a picture from a remote server and processes it after the download completes:

```
function download(url) {
    // ...
}

function process(picture) {
    // ...
}

download(url);
process(picture);
```

However, downloading a picture from a remote server takes time depending on the network speed and the size of the picture.

The following `download()` function uses the `setTimeout()` function to simulate the network request:

```
function download(url) {
    setTimeout(() => {
        // script to download the picture here
        console.log(`Downloading ${url} ...`);
    },1000);
}
```

And this code emulates the `process()` function:

```
function process(picture) {
    console.log(`Processing ${picture}`);
}
```

When you execute the following code:

```
let url = 'https://www.javascripttutorial.net/pic.jpg';

download(url);
process(url);
```

you will get the following output:

```
Processing https://javascripttutorial.net/pic.jpg
Downloading https://javascripttutorial.net/pic.jpg ...
```

This is not what you expected because the `process()` function executes before the `download()` function. The correct sequence should be:

- Download the picture and wait for the download complete.
- Process the picture.

To resolve this issue, you can pass the `process()` function to the `download()` function and execute the `process()` function inside the `download()` function once the download completes, like this:

```
function download(url, callback) {
    setTimeout(() => {
        // script to download the picture here
        console.log(`Downloading ${url} ...`);

        // process the picture once it is completed
        callback(url);
    }, 1000);
}

function process(picture) {
    console.log(`Processing ${picture}`);
}
```

```
let url = 'https://wwww.javascripttutorial.net/pic.jpg';
download(url, process);
```

Output:

```
Downloading https://www.javascripttutorial.net/pic.jpg ...
Processing https://www.javascripttutorial.net/pic.jpg
```

Now, it works as expected.

In this example, the `process()` is a callback passed into an asynchronous function.

When you use a callback to continue code execution after an asynchronous operation, the callback is called an asynchronous callback.

To make the code more concise, you can define the `process()` function as an anonymous function:

```
function download(url, callback) {
    setTimeout(() => {
        // script to download the picture here
        console.log(`Downloading ${url} ...`);
        // process the picture once it is completed
        callback(url);

    }, 1000);
}

let url = 'https://www.javascripttutorial.net/pic.jpg';
download(url, function(picture) {
    console.log(`Processing ${picture}`);
});
```

Handling errors

The `download()` function assumes that everything works fine and does not consider any exceptions. The following code introduces two callbacks: `success` and `failure` to handle the success and failure cases respectively:

```
function download(url, success, failure) {
  setTimeout(() => {
    console.log(`Downloading the picture from ${url} ...`);
    !url ? failure(url) : success(url);
  }, 1000);
}

download(
  '',
  (url) => console.log(`Processing the picture ${url}`),
  (url) => console.log(`The '${url}' is not valid`)
);
```

Nesting callbacks and the Pyramid of Doom

How do you download three pictures and process them sequentially? A typical approach is to call the `download()` function inside the callback function, like this:

```
function download(url, callback) {
  setTimeout(() => {
    console.log(`Downloading ${url} ...`);
    callback(url);
  }, 1000);
}

const url1 = 'https://www.javascripttutorial.net/pic1.jpg';
const url2 = 'https://www.javascripttutorial.net/pic2.jpg';
const url3 = 'https://www.javascripttutorial.net/pic3.jpg';

download(url1, function (url) {
  console.log(`Processing ${url}`);
  download(url2, function (url) {
    console.log(`Processing ${url}`);
    download(url3, function (url) {
      console.log(`Processing ${url}`);
    });
  });
});
```

Output:

```
Downloading https://www.javascripttutorial.net/pic1.jpg ...
Processing https://www.javascripttutorial.net/pic1.jpg
Downloading https://www.javascripttutorial.net/pic2.jpg ...
Processing https://www.javascripttutorial.net/pic2.jpg
Downloading https://www.javascripttutorial.net/pic3.jpg ...
Processing https://www.javascripttutorial.net/pic3.jpg
```

The script works perfectly fine.

However, this callback strategy does not scale well when the complexity grows significantly.

Nesting many asynchronous functions inside callbacks is known as the **pyramid of doom** or the **callback hell**:

```
asyncFunction(function(){
    asyncFunction(function(){
        asyncFunction(function(){
            asyncFunction(function(){
                asyncFunction(function(){
                    ....
                });
            });
        });
    });
});
```

To avoid the pyramid of doom, you use promises or async/await functions.

Summary

- A callback is a function passed into another function as an argument to be executed later.
- A high-order function is a function that accepts another function as an argument.
- Callback functions can be synchronous or asynchronous.