Hyperiondev

TASK

Advanced JavaScript

Visit our website

Introduction

Welcome to The 'Advanced JavaScript' Task!

You have already become fairly proficient at writing code using JavaScript. When creating big, distributed, scalable full stack web applications using JavaScript however, it becomes much more important to understand not just how to code with JavaScript but how JavaScript works too. This is what you will learn in this task. Understanding the concepts covered in this task will be vital as you learn to use the MERN stack. To understand how JavaScript works, we first need to understand some simple computer science data structures (queues, trees and stacks) and the concept of asynchronous programming.

Get in touch

Connect for support

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

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor 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!

A note from the
Hyperion Team

Understanding some basic computer science data structures is important because:

1. Understanding how data structures work will help you understand how programming platforms work. This, in turn, will help you understand what happens with the code you write once it is executed, giving you valuable insight that can help you more easily debug and optimise code.
2. Questions about data structures are often used in software developer job interviews. Understanding the basics of how the most common data structures work, helps prepare you for technical interviews and for starting a career as a developer.

SOME BASIC COMPUTER SCIENCE DATA STRUCTURES

A data structure is an object that organises and stores data. You have already used various kinds of data structures, such as arrays and lists, in your code so far. There are many other data structures in the field of computer science. These include stacks, trees, queues, heaps etc.

An in-depth discussion of all of these data structures is beyond the scope of this bootcamp. However, we will briefly discuss the basics of three important data structures used in JavaScript: trees, queues and stacks.

Queue

A queue is a concept every one of us is familiar with. We have all had to wait in a queue at a bank, bathroom or checkout line at the grocery store. Queues are also often encountered in IT, e.g. print queues. In computer science, a queue is a simple data structure where data has to enter the queue at the back and leave the queue from the front. It is therefore known as, a first in, first out (FIFO) data structure.

Queue terminology:

- Enqueue - is an operation that adds an item to the back of a queue.
- Dequeue - is an operation that removes an item from the front of a queue.

Stack

A stack is a data structure in which items are added to the top of the stack and removed from the top of the stack. It is therefore known as a last in, first out (LIFO) data structure.

Stack terminology:

- Push - is an operation that adds an item to the top of a stack.
- Pop - is an operation that removes an item from the top of a stack.

Tree

A tree is a data structure that stores data in a hierarchy made up of a collection of nodes connected by edges. Nodes are shown in the image below as squares (e.g. html, head, body, etc) and edges are the lines that connect them. The HTML DOM (Document Object Model) is an example of an implementation of a tree data structure.

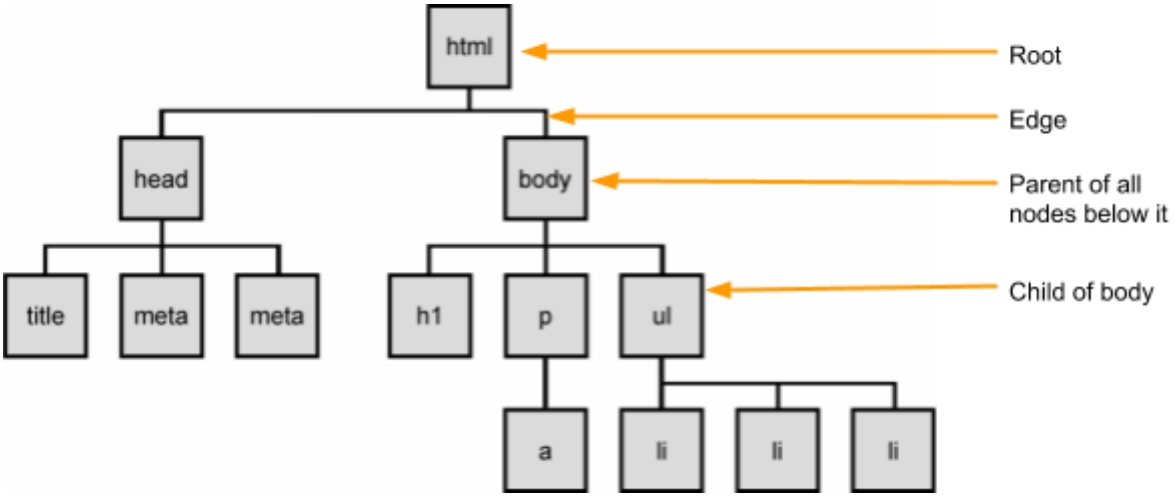

Image modified from https://programmingfordigitalart.wordpress.com/2015/03/05/w6-html-and-js/

ASYNCHRONOUS PROGRAMMING

When you learn to code it is natural that you expect that the instructions you code are carried out one after the other in the order in which you have specified them, i.e. in sequence. However, code can be executed either synchronously or asynchronously.

Synchronous code is when lines of code are executed in order, one after the other. In other words, line 2 of your code will not be executed before line 1 is finished executing.

With asynchronous code, instructions are not necessarily carried out in the order in which they are given. The order to execute each line of code is given as it is encountered BUT your program doesn't stop and wait for an instruction to be completely executed before moving on to the next instruction. Therefore, line 2 of your code may finish executing before line 1 is done. This could obviously cause some problems if you don't handle it properly!

In full stack web development, asynchronous code is important since it makes everything a lot quicker. This is because you can move on to executing other tasks while a task that is going to take a longer runs in the background. E.g. if a user writes something to a database you don't want your server to sit around and do absolutely nothing while it waits for the connection the database to be established and the data to be written to the database. Rather you want the server to issue the instruction to add data to the database but then carry on with other tasks until it hears that the data has been added and then do whatever needs to be done in that case. I.e you don't want a task that is going to take a long time to complete to block other tasks from being executed in the meantime.

However, some tasks cannot be executed until others have completed. E.g. you wouldn't be able to display the contents of a file until you had finished opening and reading the file.

To get JavaScript to work asynchronously but also ensure that tasks are executed in the correct order you can use some of the following tools:

1. **Callback functions** - A callback is a function that is passed as an argument to another function. The callback isn't executed right away. It is executed later, somewhere within the calling function.

```
function a(s, callback) {
        callback(s);
}
```

```
function b(s) {
        console.log(s + "! It's me!");
}

//call function 'a' with a predefined callback 'b'
a("Hello world", b);

//call function 'a' with a custom callback.
a("Hello world", function(s) {
        console.log(s + ", I can use callbacks");
});
```

The code above shows an example of using callback functions.

An example of a very useful method that requires a callback function to work is the map() function. The map() method creates a new array with the results of calling a provided function on every element in this array. See an example of the map function from here below:

```
// create an array
var numbers = [1, 4, 9, 16];

// call map(), passing a function
var mapped = numbers.map(function(x) { return x * 2 });

// log the result
console.log(mapped);

//Output would be: 2,8,18,32
```

2. **Timing events** - with JavaScript you can make certain functions of code wait for a certain amount of time before being executed. You can do this using timing events like:
    a. setTimeout: E.g.: `setTimeout(aFunction, 3000);`
    This code would wait 3000 milliseconds before executing the function called 'aFunction.' Notice that 'aFunction' is an example of a callback function since it 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.

b. setInterval: E.g.: `setInterval(aFunction, 3000);`
This code would repeatedly execute 'aFunction' every 3000 milliseconds.

A note from our coding mentor
Ridhaa

ES6 Update: Arrow functions

With ES2015 arrow functions are used. ***Arrow functions*** *are basically a different way of declaring and calling a normal JavaScript function. An example of an arrow function is shown below where an arrow function (highlighted in yellow) is used to create the anonymous callback function passed as an argument to the* ***`setInterval()`*** *method. An* ***anonymous function*** *is a function that was declared without any named identifier to refer to it. As such, an anonymous function is usually not accessible after its initial creation. Arrow functions have two key advantages over traditional functions:*

- *You use less code to do the same thing*
- *There is no separate* ***`this`*** *object.* ***`this`*** *has the same value as the enclosing function.*

ES2015	ES6

`function Person() {`
`  var _this = this;`
`  this.age = 0;`
 	
`  setInterval(function () {`
`  _this.age++;`
`  }, 1000);`
`}`
`var p = new Person();`
	`function Person(){`
	`  this.age = 0;`
	
	`  setInterval(() => {`
	`  this.age++;`
	`  }, 1000);`
	`}`
	`var p = new Person();`

The basic syntax for an arrow function is shown below. In the example above, there are no parameters for the arrow function, therefore, empty parentheses are used.

`(param1, param2, ..., paramN) => { statements }`

To learn more about using arrow functions, see here.

3. **Promises** - A promise basically allows the interpreter to carry on with other tasks while a function is executed without waiting for that function to finish. This is because the promise, 'promises' to let you know the outcome of the function once it is finished. The basic structure for consuming a promise is:

   ```
   doSomethingThatReturnsAPromise().then(successCallback,
   failureCallback);
   ```

   You consume the promise using **`.then()`** which accepts 2 optional arguments: a callback function for successful execution and a callback function for failure.

   An example of an API that returns a promise is the Fetch API. As the name suggests, the Fetch API provides an interface for asynchronously fetching resources. The fetch() method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response of that request, whether it is successful or not. An example of code using the fetch() method is shown below:

   ```
   require('isomorphic-fetch');

   fetch("https://www.example_API.com/")
     .then(res => res.json())
     .then((result) => {
         console.log(result);
       }),    (error) => {
       console.log(error);
      }
   ```

   Notice that to use the fetch method we first need to install (**`npm install --save isomorphic-fetch es6-promise`**) and then require 'isomorphic-fetch' (Line 1 of code above).

   In the example above, the **`fetch()`** method returns a promise that resolves to the Response of the request to fetch the resource using the URL "https://www.example_API.com/". Notice that the promise returned by the **`fetch()`** method call is consumed by the first **`.then`** statement. Here the response received from fetching the data from the API is passed as an argument to the **res.json()** method. This method returns a promise that

resolves with the result of parsing the res text as JSON. The result of calling **`res.json`** is consumed by the second **`.then()`** statement. Notice that this second **`.then`** statement has two arguments: a callback for successful execution and a callback for failure.

You could also create your own promise object. An example of a constructor for a promise is shown below.

```
let promise = new Promise(function(resolve, reject) {
  // do something, possibly async, then...

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
//Code example src: //https://developers.google.com/web/fundamentals/primers/promises
```

MDN's definition of a promise object: "The promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value."

4. **Async functions** - more recent revisions of JavaScript have made it easier to consume promises using async functions (sometimes called async/await functions). An async function can contain an **`await`** expression that pauses the execution of the async function and waits for the passed Promise's resolution, and then resumes the async function's execution and returns the resolved value. Notice that the code in the example below does exactly the same as the previous example that consumes the promise returned by the fetch() method but uses an async function instead:

```
require('isomorphic-fetch');

const request = async () => {
    const response = await fetch("https://www.example_API.com/");
    const json = await response.json();
    console.log(json);
}
request();
```

HOW DOES JAVASCRIPT WORK?

Now that we have some background information, let's put all the pieces of the puzzle together and see how all the concepts we've learned apply to JavaScript and how it works.

JavaScript is a hosted language in that it never runs on its own; rather it runs in a container. The container can be the browser (on the client-side) or in Node.js (on the server-side). As seen in the previous task, JavaScript is executed on the client-side by a JavaScript Engine. Google's V8 engine is a high-performance JavaScript engine that "compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs." On the server side, Node.js is the runtime environment that executes the JavaScript source code.

JavaScript uses an event-driven model with a single thread of execution. JavaScript's event-loop is illustrated below. Notice how *stacks* (the call stack) and *queues* (the task queue) are involved with the event loop. Keep the theory about these data structures that you have just read about in mind as we analyse the event loop.

1. When your source code is compiled, the functions are dequeued onto the **call stack** and executed by the interpreter. The call stack is a mechanism that the interpreter uses to keep track of the order in which functions are being executed. Using a single call stack means that the interpreter will only execute one function at a time, i.e. JavaScript is single threaded. If a function is slow, the interpreter has to wait for it to complete before moving on to the next function. Therefore, these slow functions could, potentially, slow down or block the whole system. Callbacks are used to prevent this. *Callbacks are non-blocking functions.*

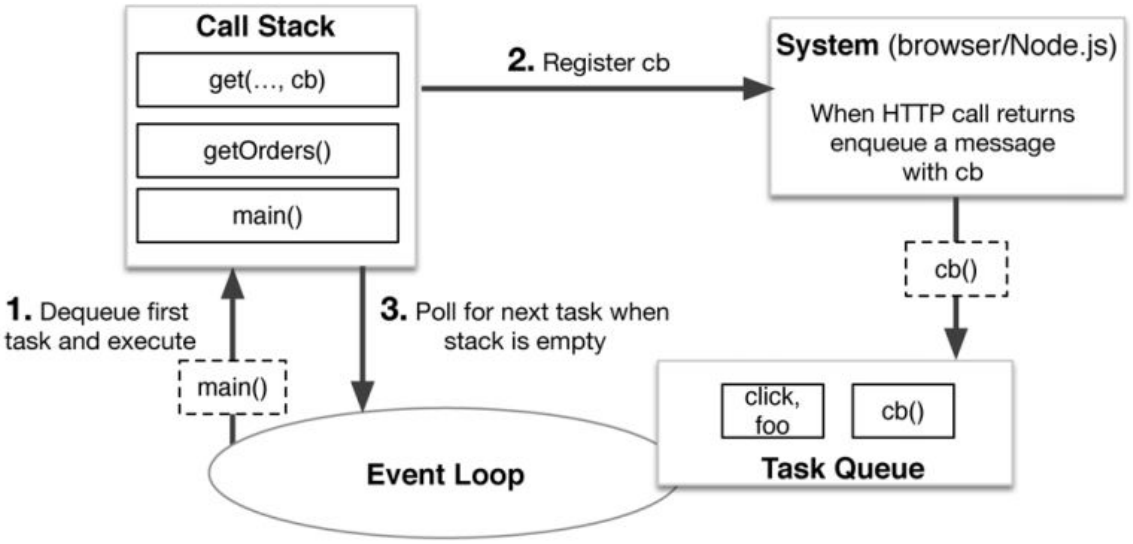

Image from article by Gallaba, Mesbah & Beschastnikh

2. When a function with a callback is encountered, the callback is sent to an API (web APIs on the client-side and C++ APIs with Node.js on the server-side). Since the API is dealing with the callback function, the interpreter no longer has to worry about it and it is removed from the call stack. This means that the other functions on the call stack can continue being executed by the interpreter. Once the API is finished processing the callback function, it pushes the callback onto the **task queue**.

3. The **event loop** sits between the call stack and the task queue. When the call stack is empty, the event loop will dequeue a function from the task queue onto the call stack where it will be executed. Therefore, callbacks on the task queue are put onto the call stack once all the other functions have been removed from the call stack.

The above should make it clear that, although JavaScript is single threaded, it is fast because it allows functions to be executed concurrently (at the same time) by letting callbacks be handled by APIs which act like multiple threads.

Use this tool and watch the accompanying video to make sure that you can visualise exactly how JavaScript's event loop works. The concepts discussed here are some of the most important but hardest to understand in this bootcamp. You will be using these concepts more and more as you work with the MERN stack in the following tasks. It is vital that you do understand how JavaScript supports asynchronous, concurrent programming and this video is well worth watching, as it will help you to gain a better understanding of these concepts.

A note from our coding mentor

Nkosi

Did you know? A stack is a data structure with a specific size. If more functions are placed onto the call stack than it was defined to be able to carry, you will get a stack overflow error.

Instructions

- Read through and run all the examples in the 'examples' folder that accompanies this task.

Compulsory Task 1

Follow these steps:

- Copy the 'Example 2' directory that accompanies this task (in the 'examples' folder) to your local computer. Open 'fetch-example.js'. Complete all the steps outlined in the comments of this file. Once you have completed all the steps in this file, copy the 'Example 2' folder back to Dropbox.
- Create a website that will display three random Chuck Norris jokes. Use the API found here: http://www.icndb.com/api/. You can use this URL to fetch the jokes: http://api.icndb.com/jokes/random/3.
- Create a webpage that always displays the current time. The time should be updated every second. Hint: use setInterval.

Once you have completed the task in line with the instructions above, click the button below to request your mentor to review your work and provide feedback. If

you have any questions while attempting the task, leave your mentor a note on the comments.txt file in your Student Dropbox folder.

Completed the task(s)?

Ask your mentor review your work!

Review work

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think 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.