TASK

Closures And Arrow Functions

Visit our website

Introduction

WELCOME TO THE CLOSURES AND ARROW FUNCTIONS TASK!

At this point, you've gained a high-level understanding of how to use and implement functions in JavaScript (JS). In this task, we're going to be going over two concepts based on functions: closures (functions within a function), and arrow functions (a different way of creating functions).

WHAT ARE CLOSURES?

Closures, in the simplest sense, are functions inside another function. What makes closures so unique and powerful is their ability to reference variables from the outer function scope but also allow you to create variables for them to be used at a later stage.

Something most people may not know about is that during your entire time programming in JavaScript so far, you've actually most likely made use of a closure in some way!

Take a look at the code below:

```
const myVar = "foo";

function printVar() {
   console.log(myVar);
}

printVar();
```

You may notice something familiar here, you have a global variable (this is a variable that can be accessed by any portion of your code after it's been defined) and then you have your function that calls that variable! This is a very basic example of a closure. The idea is that our JS file is one scope and has a variable, and then we have a function within the original scope that makes use of that variable!

In simple terms, closures are functions that are created within the scope of another function.

The more common use of closures

The above example is just a very basic example of what the idea of a closure is. However, we're going to discuss the more common way of creating a closure and how you can expect to see them created in the programming world once you land your dream tech job!

Let us take a look at the code example below:

```
function outerFunction(outerVariable){
   return function innerFunction(innerVariable) {
       console.log(outerVariable);
       console.log(innerVariable);
   }
}

const functionVariable = outerFunction('foo');
functionVariable('bar');
```

You may feel quite overwhelmed when looking at this, and that's completely understandable! We're going to review this slowly and one line at a time to help you fully understand the logic behind a closure.

Let us focus on our two functions, **outerFunction** and **InnerFunction**. You'll notice that our **outerFunction** returns another function inside itself. This can feel quite confusing, but we will come back to this when we get to our variable creation.

Finally, we print out the two parameters that we pass into our functions, and that's it. Closures are just functions that return a function inside themself. Now, let's get into the logic behind it and how they work

.

You'll notice that we do something different compared to what you're used to when it comes to using functions: we assign it to a variable. Why is that? Can't we just call the function?

While you can call the function on its own, your program wouldn't do anything. This is because just calling the function on its own without assigning a variable means that **innerFunction** will never actually be called!

The reason we used the variable **functionVariable** in the example (keep in mind that this can actually be any name) is that we need to have JS remember what parameter we used for the **outerFunction** function. So by assigning this to a variable, JS will remember the value of **outerFunction** and from there can move on to the **innerFunction**. You'll notice that after we create the variable, we simply write the variable name out to run **innerFunction** (that's all you need to remember is the variable you're using is actually just calling the closure!)

The technique of returning a function as output upon the application of one argument is generally known as **currying**, named after the logician, Haskell Curry. Currying was originally intended to simulate multi-argument functions in languages that only support functions with exactly one argument and must have a return value. The return value when currying is a... surprise surprise... a closure! This is by virtue of the function being returned closing over the argument that would need to be passed to the parent function for the closure to be returned. This technique of passing an incomplete number of arguments to a function to retrieve a closure is known as **partial application** because in languages where functions are carried by default, passing arguments to them is conventionally referred to as *applying the function to an argument*. Currying and partial application are often coupled together as features in functional programming languages such as F#, which have gone to have a direct influence on the current state of JavaScript. As you will see later on in this task, arrow functions make it uber-convenient to write curried functions and partially apply them. This use of functions as values has a long tradition dating back to the foundations of computer science with logicians such as Alan Turing's PhD supervisor, **Alonzo Church**. If you wish to learn more about functional programming, you can always request for resources from **quality@hyperiondev.com**, which you can gently work through in your spare time.

So now you may be wondering, where would we use this? Wouldn't it be easier to just have a function that has two parameters?

You are correct, but there are some situations where it can help to avoid repetitive code! Take a look at the example below:

```
function welcomeMessage(pronouns, name) {
```

```
  console.log(`Hello ${name}, your pronouns have been set to ${pronouns}`);
}

welcomeMessage('he/him','Logan');
welcomeMessage('he/him','Jack');
welcomeMessage('she/her','Kylie');
welcomeMessage('she/her','Sally');
```

As I'm sure you can notice, we repeat **he/him** and **she/her** a lot in this code! This breaks the DRY principle (don't repeat yourself). This is where closures come in really useful as we only need to call the function once with the pronouns we need to use, and any following function calls just need the name of the person! Take a look at the below code:

```
function userPronouns(pronouns) {
   return function userName(name) {
       console.log(`Hello ${name}, your pronouns have been set to
${pronouns}`);
   }
}

const heHim = userPronouns("he/him");
heHim('Logan');
heHim('Jack');

const sheHer = userPronouns("she/her");
sheHer('Kylie');
sheHer('Sally');
```

Notice how we just create one variable each, **heHim** and **sheHer**. These make use of the default values we provide being the users' pronouns. Now we just call that variable name with the name of the user! The example would produce this output!

Hello Logan, your pronouns have been set to he/him

Hello Jack, your pronouns have been set to he/him

Hello Kylie, your pronouns have been set to she/her

Hello Sally, your pronouns have been set to she/her

This saves us a lot of repetitive code as closures simply save the required data that we needed and we can reuse it where we use the closure. This is the beauty of closures!

Important information regarding closures.

While closures are amazing in many situations, it's important to remember that you should always attempt to limit how many you use in your program and only use them when absolutely necessary. The main reason for this is because of the performance and memory issues it can cause when overused.

Because JS is required to store the information in the original function, this means that it uses more resources to keep the information that was input. While it may not have a huge impact at this point in time, with your programs being relatively small, it's important to remember that in the programming world, efficiency is always something that should be in the back of your mind.

ARROW FUNCTIONS

Congratulations! You made it through the most challenging part of this task! As a reward, we're going to be showing you an awesome new way of creating functions.

Arrow functions get their name because of their unique "arrow" (**=>**) when creating a function. Take a look at the example below:

```
const myFunction = name => {
    return `Wow arrow! Very wow! Hi ${name}!`
}

// OR without braces
const myFunction = name => `Wow arrow! Very wow! Hi ${name}!`
```

You'll notice that this is a very unique way of creating a function: after the **()** you'll notice a **=>** sign which is your arrow! The reason it has this name is that... well... it's an arrow!

So let's break this code down to help you understand what on earth is happening.

You'll notice we make use of creating a "variable" (or how you are familiar with variables being created). This is how we create the name of our function. Whatever name you want your function to use would be where **myFunction** is. This is then followed by the equals sign and then your parameters. It's important to remember that this can take any number of values so long as they are separated by commas.

Finally, we have our arrow. This is how JS knows that this is a function rather than a variable. This is followed by the curly braces, and then you can start writing your own function code. It's exactly the same as the functions you've previously worked with - the only difference is the syntax to create it!

Also, important to note is that unlike non-arrow functions, arrow functions don't have their own **'this' binding**. They inherit their 'this' binding from the **parent *lexical scope*** - which is where the functions were defined (in your programme before you ran it). This means you can't define constructor functions as arrow functions as that depends on the functions having their inner own 'this binding. That also means that arrow functions are ***lexical closures*** because they *close* over their outer lexical scope by capturing/availing its 'this' binding. Generally speaking, all functions in JavaScript are closures because they close over their environment, but JavaScript didn't have lexical closures - it closures were dependent on the execution environment rather than the lexical environment. Don't be frightened by the big word 'lexical' - it simply means 'where they were defined in your programme before execution'.

Arrow functions are part of the evolution of JavaScript's programming style away from traditional object-oriented programming towards functional programming, where the emphasis is on the abstraction by means of manipulating functions the way you would any other value or object rather than abstracting using objects, prototype chains and classes. If you think of side effects as anything that a function does other than accepting an argument and returning a value (e.g. printing to a console), the functional programming paradigm also emphasises avoidance of side effects to reduce the complexity of reasoning about your programmes, especially when dealing with complex control flow, e.g. in asynchronous programmes, which you will write in a few tasks. This is why it's become quite popular in the JavaScript ecosystem via features such as arrow functions and frameworks such as React, which you will learn later in your bootcamp.

Why use arrow functions?

Arrow functions are universally used as the preferred mode of creating functions for many web developers who make use of JS. The reason for this is the clean and

easy-to-read syntax that is used to create these functions. It's easy to distinguish between the function name and the parameters and actually see where functions are (because of the unique arrow!).

Let's compare non-arrow functions and arrow functions as closures with similar behaviour below so you can see how simpler your code gets:

Non-arrow version

```
function outerFunction(outerVariable){
   return function (innerVariable) {
       console.log(outerVariable)
       console.log(innerVariable)
   }
}

const functionVariable =
outerFunction('foo');
functionVariable('bar');

function userPronouns(pronouns) {
   return function (name) {
       console.log(`Hello ${name}, your
pronouns have been set to ${pronouns}`);
   }
}
const heHim = userPronouns("he/him");
heHim('Logan');
heHim('Jack');

const sheHer = userPronouns("she/her");
sheHer('Kylie');
sheHer('Sally');

const nonBinary =
userPronouns("non-binary");
nonBinary("Vince");
nonBinary("Hannah");
```

Arrow version

```
const outerFunction = outerVariable
=> innerVariable => {
   console.log(outerVariable)
   console.log(innerVariable)
}

const functionVariable =
outerFunction('foo');
functionVariable('bar');

const userPronouns = pronouns =>
name => {
   console.log(`Hello ${name}, your
pronouns have been set to
${pronouns}`);
}
const heHim =
userPronouns("he/him");
heHim('Logan');
heHim('Jack');

const sheHer =
userPronouns("she/her");
sheHer('Kylie');
sheHer('Sally');

const nonBinary =
userPronouns("non-binary");
nonBinary("Vince");
```

```
// ####################

function multiply(a) {
   return function(b) {
       return a * b
   }
}

console.log(multiply(10)(32.4));

function map(mapper) {
   return function(array) {
       const mappedArray = []
       for (let i = 0; i < array.length;
i++) {
           mappedValue = mapper(array[i]);
           mappedArray.push(mappedValue);
       }
       return mappedArray
   }
}

const numbers1 =[1, 2, 3, 4, 5];
const numbers2 =[122, 230, 430, 140, 9.50];

const multiplyAllBy10 = map(multiply(10));

console.log(multiplyAllBy10(numbers1));
console.log(multiplyAllBy10(numbers2));

// ####################
```

```
nonBinary("Hannah");

// ####################

// No need for the 'return' keyword
const multiply = a => b => a * b

console.log(multiply(10)(32.4));

const map = mapper => array => {
   const mappedArray = []
   for (let i = 0; i <
array.length; i++) {
       mappedValue =
mapper(array[i])

mappedArray.push(mappedValue)
   }
   return mappedArray
}

const numbers1 = [1, 2, 3, 4, 5];
const numbers2 = [122, 230, 430,
140, 9.50];

const multiplyAllBy10 =
map(multiply(10));

console.log(multiplyAllBy10(numbers
1));
console.log(multiplyAllBy10(numbers
2));

// ####################
```

Overall, arrow functions are preferred because of the clean and simple design they have in comparison to regular functions, which is why it's recommended to get into the habit of making use of them in all tasks from this point forward.

Compulsory Task 1

Follow these steps:

- Create a new JS file called **closures.js**
- This project is going to have you make use of closures to create a very simple multiplication application.
- Your project should have one main number that will be used to multiply all other numbers (for example, 10).
- Your program should then loop through 10 numbers and push them into the inner scope of your callback function.
- Your output should look something like this.
    - 10 * 0 = 0
    - 10 * 1 = 10
    - 10 * 2 = 20
    - 10 * 3 = 30
    - 10 * 4 = 40
    - 10 * 5 = 50
    - 10 * 6 = 60
    - 10 * 7 = 70
    - 10 * 8 = 80
    - 10 * 9 = 90
    - 10 * 10 = 100

Compulsory Task 2

Follow these steps:

- For this task, you need to go back to the **Task 12** functions that you created for Compulsory Task 3. Copy the project into this folder.
- Now, simply convert all your normal functions into arrow functions.

Thing(s) to look out for:

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this, but **Visual Studio Code** may not be installed correctly.
2. If you are not using macOS, Linux or Windows, please ask a mentor for alternative instructions.

Rate us

Share your thoughts

Hyperion 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.