Week 07 Readings

**Chapter 11**

**Further Functions: Firs-Class Objects.**

* they can be passed around in the same way as every other value
* have their own properties and methods
* accepting other functions as parameters and being returned by other functions

**Function Properties and Methods**

* function square(x) {
* return x\*x;
* }

**Call and Apply Methods**

The call() method can be used to set the value of this inside a function to an object that is provided as the first argument.

* function sayHello(){
* return `Hello, my name is ${ this.name }`;}

**Immediately Invoked Function Expressions**

Short for IIFE

* anonymous function that, as the name suggests, is invoked as soon as it’s defined
* This is easily achieved by placing parentheses at the end of the function definition
* (function(){
* const temp = 'World';
* console.log(`Hello ${temp}`);
* })();
* << 'Hello World'

**Temporary Variables**

* There is no way to remove a variable from a scope once it’s been declared
* let a = 1;
* let b = 2;
* (()=>{
* const temp = a;
* a = b;
* b = temp;
* })();
* a;
* << 2
* b;
* << 1
* console.log(temp);
* << Error: "temp is not defined"

**Functions that Define and Rewrite Themselves**

The dynamic nature of JavaScript means that a function is able to not only call itself, but define itself, and even redefine itself.

*function* party(){

console.log('Wow this is amazing!');

party = *function*(){

console.log('Been there, got the T-Shirt');

}}

*const* beachParty = party; *// note that the party function has not been invoked*

beachParty(); *// the party() function has now been redefined, even though it hasn't been called explicitly*

<< 'Wow this is amazing!'

party();

<< 'Been there, got the T-Shirt'

beachParty(); *// but this function hasn't been redefined*

<< 'Wow this is amazing!'

beachParty(); *// no matter how many times this is called it will remain the same*<< 'Wow this is amazing!'

**Recursive Functions**

A recursive function is one that invokes itself until a certain condition is met. It’s a useful tool to use when iterative processes are involved.

* function factorial(n) {
* if (n === 0) {
* return 1;
* } else {
* return n \* factorial(n - 1);
* }}

For example, if we start with the number 18, we would have the following sequence:

18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

**Callbacks**

Callbacks can be used to facilitate event-driven asynchronous programming. JavaScript is a single-threaded environment, which means only one piece of code will ever be processed at a time.

*function* wait(message, callback, seconds){

setTimeout(callback,seconds \* 1000);

console.log(message);}

*function* selfDestruct(){

console.log('BOOOOM!');

}

wait('This tape will self-destruct in five seconds ... ', selfDestruct, 5);

console.log('Hmmm, should I accept this mission or not ... ?');

<< 'This tape will self-destruct in five seconds ... '

<< 'Hmmm, should I accept this mission or not ... ? '

<< 'BOOOOM!'

**Closures:** are one of JavaScript’s most powerful features, but they can be difficult to get your head around initially.

**Functional Programming:** programming paradigm

**Chapter 13**

**Ajax:** is a technique that allows web pages to communicate asynchronously with a server, and it dynamically updates web pages without reloading. This enables data to be sent and received in the background, as well as portions of a page to be updated in response to user events, while the rest of the program continues to run.

* The use of Ajax revolutionized how websites worked, and ushered in a new age of web applications. Web pages were no longer static, but dynamic applications.

**The Fetch API**

* The XMLHttpRequest object was finally standardized by the WHATWG and W3C as part of the HTML5 specification, despite it originally being implemented by Microsoft many years earlier, and already available in most browsers.
* The Fetch API uses promises to avoid callback hell, and also streamlines a number of concepts that had become cumbersome when using the XMLHttpRequest object.

**Receiving Information**

<!doctype html>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Ajax Example</title>

</head><body>

<button id='number'>Number Fact</button>

<button id='chuck'>Chuck Norris Fact</button>

<div id='output'> Ajax response will appear here</div>

<script src='main.js'></script>

</body>

*const* textButton = document.getElementById('number');

*const* apiButton = document.getElementById('chuck');

*const* outputDiv = document.getElementById('output');

textButton.addEventListener('click', () => {

fetch(textURL)

.then( response => {

outputDiv.innerHTML = 'Waiting for response...';

*if*(response.ok) {

*return* response;

}

*throw* Error(response.statusText);

})

.then( response => response.text() )

.then( text => outputDiv.innerText = text )

.*catch*( error => console.log('There was an error:', error))

},false);

**FormData**

The Fetch API includes the FormData interface, which makes it much easier to submit information in forms using Ajax.

A FormData instance is created using a constructor function:

const data = new FormData();

Chapter 5, 5.1 Flattening,

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

console.log(arrays.reduce((flat, current) => flat.concat(current), []));

// → [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6]

Chapter 18, 18.2 A JavaScript Workbench

<!doctype html>

<script src="code/chapter/18\_http.js"></script>

<textarea id="code">return "hi";</textarea>

<button id="button">Run</button>

<pre id="output"></pre>

<script>

document.querySelector("#button").addEventListener("click", () => {

let code = document.querySelector("#code").value;

let outputNode = document.querySelector("#output");

try {

let result = Function(code)();

outputNode.innerText = String(result);

} catch (e) {

outputNode.innerText = "Error: " + e;

}

});

</script>

Text

Description automatically generated