**Chapter 1**

**Javascript functions**

**Javascript for Server side Development**

In this chapter, we will be reviewing functions, which will be used frequently in our node js code. Besides recapping the conventional function definition you have learnt last semester, we will also be looking at First-class functions, closures and callbacks, concepts which are important when you create your node js applications.

**Functions**

Functions are really powerful in JavaScript. We will examine functions in more detail in the examples below.

A normal function structure in JavaScript is defined below:

|  |
| --- |
| function name(arg1) {  //code  } |

Functions can accept any number of arguments and also return a value in JavaScript. In the absence of a return statement, a function returns undefined. Consider the below code example:

function test(num){

console.log(num); //prints num

}

console.log(test(1));

What is the output?

Graphical user interface, text

Description automatically generated

**First class Function**

A function without a name is called an anonymous function. In JavaScript, functions are objects. You can assign functions to variables, to array elements, and to other objects. They can also be passed around as arguments to other functions or be returned from those functions.

Javascript supports first-class functions and a function that is defined can be treated the same way as any other variable in javascript. Consider the below example:

|  |
| --- |
| var funct1 = function () { // anonymous function  console.log(“Printing!!”);  }  funct1(); |

When the code is run, “Printing!!” will be printed in the console.

**Higher-Order Functions**

In Javascript, we can pass functions to other functions. Functions that take functions as arguments are called higher-order functions. An example of a higher order function in javascript is setInterval(function, milliseconds) which takes in a function and runs the function every milliseconds milliseconds for the input milliseconds value. In the below example:

|  |
| --- |
| setInterval(function () {  console.log(‘Running….’);  }, 1000); |

If you run this application in Node.js, you will see the console.log message every 1s.

Note that we can also create a function and passing it into the setInterval() function.

|  |
| --- |
| function run() {  console.log(‘Running….’);  }  setInterval(run, 1000); |

**Closures**

A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function’s variables.

The closure function has 3 key properties:

1. It has access to variables defined in itself
2. It has access to the outer function’s variables
3. The inner function will continue to have access to the variables from the outer scope even after the outer function has returned

Let’s look at the below example:

|  |
| --- |
| function outerFunction(x) {  var variableInOuterFunction = x;  inner=function innerFunction() {  console.log(“In inner function…”);  console.log(variableInOuterFunction); // Access a variable from the   //outer scope  }  return inner;  }  inner=outerFunction('hello world!');  inner() |

**JavaScript and Callbacks**

JavaScript is synchronous by default and is single threaded.

Code is executed one line after another.

Let’s look at a basic example:

|  |
| --- |
| function firstFn(){  console.log(1);  }  function secondFn(){  console.log(2);  }  firstFn();  secondFn(); |

The function firstFn is executed first, followed by the function secondFn, deriving 2 lines of output 1 followed by 2.

But JavaScript was born inside the browser, its main job, in the beginning, was to respond to user actions, like onClick, onMouseOver, onChange, onSubmit and so on. How could it do this with a synchronous programming model?

The answer was in its environment. The browser provides a way to do it by providing a set of APIs that can handle this kind of functionality, making javascript event driven.

Node.js also introduced a non-blocking I/O environment to extend this concept to file access, network calls and so on.

You can’t know when a user is going to click a button, so what you do is, you define an event handler for the click event. This event handler accepts a function, which will be called when the event is triggered:

|  |
| --- |
| document.getElementById('button').addEventListener('click', function() {  //item clicked  }) |

This is the so-called callback.

A callback is a simple function that’s passed as a value to another function, and will only be executed when the parent function wants it to be executed. We can do this because JavaScript has first-class functions, which can be assigned to variables and passed around to other functions (called higher-order functions)

One common example of use of callback is the setTimeout() asynchronous function. setTimeout() is similar to setTimeInterval() except it runs only once:

|  |
| --- |
| setTimeout(function() => {  ….  }, 2000) |

To illustrate the concept of callback functions, let’s consider a situation whereby function firstFn() runs some code that needs to take a period of time, for example a file I/O operation to download some file? In such a situation, assuming the next few lines of code can proceed without waiting for the I/O operation to complete, it would make sense for the code instructions to continue to execute without waiting for the long I/O operation. To simulate this action, we are going to use setTimeout which is a JavaScript asynchronous function that calls a function after a set amount of time. We’ll delay our firstFn() function code for 2000 milliseconds to simulate a file download.

|  |
| --- |
| function firstFn(){  // Simulate a code delay  setTimeout( function(){  console.log(1);  }, 2000 );  }  function secondFn(){  console.log(2);  }  firstFn();  secondFn(); |

Let’s run the new code and observe the output. What do you get now?

Even though we invoked the firstFn() function first, result of secondFn() actually gets printed first before output from firstFn().

It’s not that JavaScript didn’t execute our functions in the order of the code, but it didn’t wait for a response from firstFn() before moving on to execute secondFn() since setTimeout() is an asynchronous function. The support of callbacks enable support of non-blocking I/O environment in node js.

|  |
| --- |
| Computation  1  3  Long I/O  Callback  2  Computation |

**Handling errors in callbacks**

How can we handle errors in callbacks, such as when file reading fails when supplied file name is wrong? We will use what Node.js adopted: the first parameter in any callback function is the error object.

If there is no error, the object is null. If there is an error, it contains some description of the error. An example is shown below:

|  |
| --- |
| const fs = require('fs');  fs.readFile('/file.txt',’utf-8’, function(err, data){  if (err !== null) {  //handle error  console.log(err)  }else  //no errors, process data  console.log(data)  }) |

**Arrow Functions**

Lastly, introduced in ES6, is the support of arrow functions.

Arrow functions make our code more concise and simplify function scoping. By using arrow functions, we avoid having to type the function keyword and return keyword (it’s implicit in arrow functions).

Let’s compare the versions of the callback error handling code with and without arrow functions.

|  |
| --- |
| const fs = require('fs');  fs.readFile('/file.txt', (err, data)=>  {  if (err !== null) {  //handle error  console.log(err)  }else    //no errors, process data  console.log(data)  }); |

Another example on creating a function which returns the multiplication of 2 values:

|  |
| --- |
| multiplyFunct = (x, y) => x \* y;  console.log(multiplyFunct(2,3)); |