FullStackJS Period 1

Contents

Explain and Reflect:

Differences between Java and Javascript

- Java is an object orientated programming (OOP) language, while Javascript is an OOP **scripting** language. Java creates an application which runs on JVM or browser. Javascript is run on browser only. Java can run multithreads, Javascript can't.

Two strategies for improving Javascript

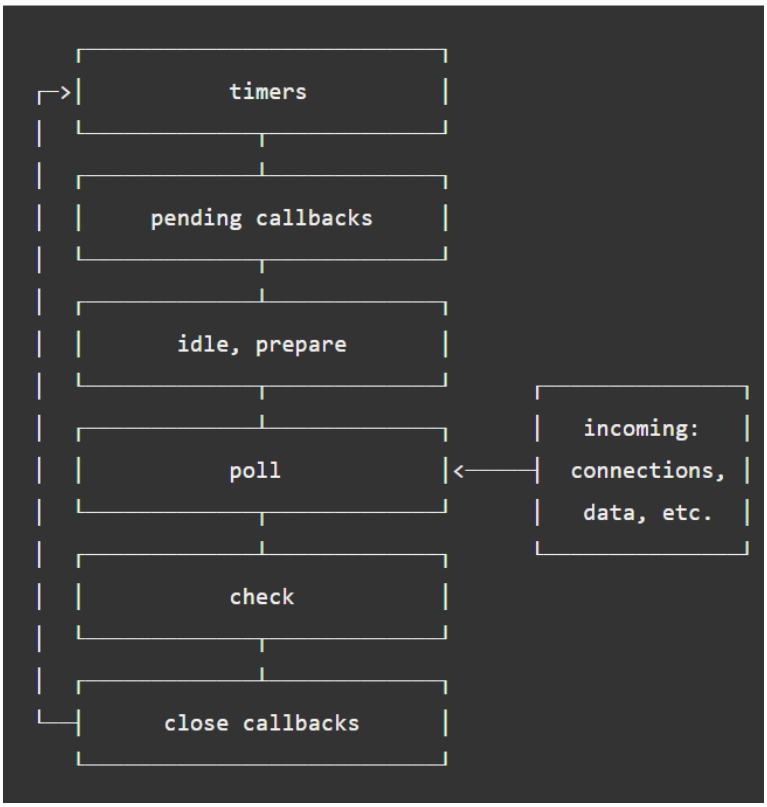
- ES6 and ES7 are a newer version of Javascript, which means not all browsers, (or not browser friendly), support them ***yet***. To make use of code made in ES6 and ES7, both TypeScript and Babel can be used to transpile code to ES5. **TypeScript** both transpiles and extends Javascript through static typing and class based OOP. **Babel** simply converts ES6 to browser friendly ES5

Generally about Node.js

- Node.js is a platform for building fast and scalable server applications using javascript
- Npm is the package manager for node.js
- Node.js is an open source project designed to help you write javascript programs that communicates to networks, file systems or other I/O sources
- Node is neither a web framework or a programming language
- Node does I/O in an asynchronous way
- Node handles I/O with callbacks, events, streams and modules
- Node.js is best, or specially, suited for applications where one would like to maintain a persistent connection from browser back to the server
- Also good for situations where code can be reused a lot across client/server

Event loop in node.js

- The event loop is what allows Node.js to perform non-blocking I/O operations
- They can handle multiple operations working in the background, and once an operation is completed, Node.js is notified that the appropriate callback can, or may, be added to the queue poll, that can be executed later.



- Each of the boxes above has a FIFO, first in first out, queue and callbacks to be executed. When an event loop enters any of the boxes above, it performs operations specific to that box, and executes the callbacks in that queue until it has reached maximum callbacks execution. And at that point the event loop moves on to the next box

Purpose of the tools Babel and Webpack

- **Webpack** is a static module bundler for modern javascript applications, when webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles
- **Webpack** has different properties:
  - **Entry** point, which indicates which module, webpack should use to begin building out its internal dependency graph
  - **Output** property tells webpack where to emit the bundles it creates and how to name the files
  - **Loaders** property allows webpack to process process other types of files and convert them into valid modules that the application can consume and add to the dependency graph. (Out of the box, webpack only understands jacvascript and json files, hence **loaders** can be used).

    Loaders has two properties:
    - **Test** property identifies which file(s) should be transformed
    - **Use** property indicates which **loader** should be used to do the transforming

- **Plugin** property can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.
- **Babel** is a transpiler which transpiles ES6 to browser friendly ES5.

Purpose of "use strict"

- Defines that the javascript code to be executed in "strict" mode
- It is a literal expression
- With "use strict" you **can't** have undeclared variables
- It makes it easier to write "secure" javascript code, and gives errors on bad syntax'
- One example could be using an object without declaring it.

Explain using sufficient code examples, using these features

Variable/function hoisting

- Hoisting means raising/lifting something up, hoisting is understood as you have defined a variable, but you have not assigned any value to it. Whenever you define a variable within the body of a function, the value is available from where it is defined to the end of the function, but because of hoisting the variable is lifted to the start of the function scope
- ```
  function a(){
      console.log(foo);
      var foo = "hello";
  }
  a();
  ```

this

- If you are inside a specific scope, object or function, "this" means whichever object you are in
- ```
  function GreetMe(name) {
      this.greeting = 'Hello ';
      this.name = name;
      this.speak = function() {
          console.log(this.greeting + this.name);
          console.log(this);
      }
  };
  ```
- "this" in a global scope means we are in window scope, when you do this.*arg* you are creating a public property that is accessible from outside.

- If you want to access an object within a local scope you can't use "this" from outside the scope to access it, because it can't read the property because it is not accessible, because it is a private variable/object.

Function closures

- In javascript you can get by without passing any variables/parameter in the function
- Closures are nothing but functions with preserved data
- Difference between a normal function behavior and closure is, that after a normal function is executed, the variables and data resets, with closure the variables and data remains, it will be stored in memory
-

```
var x = 1;
function a() {
    var y = 2;
    function b() {
        var z = 3;
        console.log(x+y+z);
    }
    b();
}
a(); //displays 6
```

Javascript prototype

- Javascript is a prototype based language
- By default every function has a property called prototype
-

```
function Person() {}
Person.prototype.name = "A";
Person.prototype.age = 20;

var person1 = new Person();
console.log(person1.name); //A

var person2 = new Person();
person2.name = "B";
console.log(person2.name); //B
```

Callbacks

- Any function that is passed as an argument is a callback function

```
const filterDire = require("./MagicOfCallbacks");
const pathToFile = process.argv[2];
const ext = "." + process.argv[3];

filterDire(pathToFile, ext, (err, data) => {
    if(err){
        throw new Error("WHOOPSIE " + err);
    }
    console.log(data.join("\n"));
});
```

map(), filter() and reduce()

- **map()** does not only filter, but also creates new arrays of anything from a current array, map() takes functions

```
//Mapped()
// The map() method creates a new array with the results of calling a
const mappedArrToUppercase = names.map((name) => {
    return name.toUpperCase();
});
// console.log(mappedArrToUppercase);
document.getElementById("mapped").innerHTML = mappedArrToUppercase;
```

- **filter()** allows you to filter things out of the array

```
// filter()
// The filter() method creates a new array with all elements that pa
let names = ["Lars", "Jan", "Peter", "Bo", "Frederik", "Malik"];
document.getElementById("original").innerHTML = names;
// console.log(names);
const lessThanOrEqual = names.filter((name) => name.length <= 3);
// console.log(lessThanOrEqual);
document.getElementById("filter").innerHTML = lessThanOrEqual;
```

- **reduce()** reduces the array to a single value

```
let numbers = [2, 3, 67, 33].reduce((sum, value) => sum + value, 0);
// console.log(numbers);
function red() {
    document.getElementById("reduced").innerHTML = numbers;
};
let button2 = document.getElementById("btn").onclick = red;
```

Examples of user-defined reusable modules implemented in Node.js

```
const fs = require('fs');
const path = require("path");
const pathToFile = process.argv[2];
const ext = "." + process.argv[3];
//returns array of data, but as a callback, so it is unused after data is acquired
fs.readdir(pathToFile, (err, data) => {
    if(err){
        throw new Exception("Whoops " + err);
    }
    //arrow functions has an explicit return, so no need to type return
    const filteredArr = data.filter((filename) => path.extname(filename) === ext);
    console.log(filteredArr.join("\n"));
});
```

ES6, 7, 8… and TypeScript

- **Let**

```
let user = new Student("Malik", "Sharfo");
```

- **Arrow**

```
let reusable = (setAge, setName) => {
    const res = "Name is ";
    let getInfo = () => {
        //return res + (" " + setName + ", age is " + setAge);
        return res + (`${setName}` + ", age is " + `${setAge}`);
    }
    return getInfo()
}
let finalRes = reusable(22, "Malik");
//console.log("Check: ", finalRes);
```

- **this**

```
//method that gets detail
p1.details = function() {
    //return this.firstName + " " + this.lastName + ", " + this.age;
    return `${this.firstName}` + " " + `${this.lastName}` + ", " + `${this.age}`;
};
//console.log("check ", p1.details());
document.getElementById("person").innerHTML = p1.details();
```

Examples of TypeScript

```
let message: string = "Hello World";
console.log(message);

class Student {
    fullName: string;
    constructor(public firstName: string, public lastName: string) {
        this.fullName = firstName + " " + lastName;
    }
}

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Malik", "Sharfo");

//document.body.innerHTML = greeter(user);
console.log(greeter(user));
```

- **ES5 support of modules**

```
const fs = require('fs');
const path = require('path');
module.exports = function (pathToFiles, ext, callback) {
    fs.readdir(pathToFiles, function (err, files) {
        if (err) {
            return callback(err);
        }
        files = files.filter(function (file) {
            return path.extname(file) === "." + ext;
        });
        callback(null, files);
    });
};
```

- **Inheritance**

-
```
class Repository extends React.Component {

  constructor(props) {
    super(props);
    this.state = { repo: {} }
  }
```

Designers of ECMAScript

- The ECMA TC39 committee is responsible for evolving ECMAScript programming language and authoring the specification
- TC39 meets regularly, its meetings are attended by delegates that members send and by invited experts.
- It has 5 stages, strawman, proposal, draft, candidate, finished

Callback, promises and async/await

- **Callback hell**
-
```
const verifyUser = function(username, password, callback){
    dataBase.verifyUser(username, password, (error, userInfo) => {
        if (error) {
            callback(error)
        }else{
            dataBase.getRoles(username, (error, roles) => {
                if (error){
                    callback(error)
                }else {
                    dataBase.logAccess(username, (error) => {
                        if (error){
                            callback(error);
                        }else{
                            callback(null, userInfo, roles);
                        }
                    })
                }
            })
        }
    })
};
```
- To avoid callback hells, using .then and catch is recommended

```
const verifyUser = function(username, password) {
    database.verifyUser(username, password)
        .then(userInfo => dataBase.getRoles(userInfo))
        .then(rolesInfo => dataBase.logAccess(rolesInfo))
        .then(finalResult => {
            //do whatever the 'callback' would do
        })
        .catch((err) => {
            //do whatever the error handler needs
        });
};
```

- **Errorhandling with promises**

```
const fetch = require("node-fetch");
async function getSwapi() {
    try {
        await fetch("https://swapi.co/apie/people/1")
            .then(res => res.json())
            .then(data => console.log(data.name));
        await fetch("https://swapi.co/api/people/2")
            .then(res => res.json())
            .then(data => console.log(data.name));
        await fetch("https://swapi.co/api/people/3")
            .then(res => res.json())
            .then(data => console.log(data.name));
    } catch (e) {
        console.log("Error: ", e);
    }
}
getSwapi();
```