CS 4220

- Current Trends in Web Design & Development -

Cydney Auman

AGENDA

Synchronous vs Asynchronous

In a **synchronous** programming model, things happen one at a time. When you call a function that performs a long-running action, it returns *only* when the action has finished and it can return the result. This stops your program for the time the action takes.

An **asynchronous** model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result.

JavaScript code ***runs single threaded***. There is just one thing happening at a time. So, both JavaScript platforms - browsers and **Node.js relys on the Event Loop** for asynchronous programming for anything that takes awhile to run.

Event Loop, Call Stack & Queue

The **Event Loop** is what allows JavaScript to use callbacks and promises. It works with the Call Stack, Node.js Interfaces, and the Event Queue.

JavaScript has a single **Call Stack** in which it keeps track of what functions are currently executing and what function needs to be executed next.

If an asynchronous function is added to the Call Stack then it gets moved to be handled by **Node.js background processes**.

When a process has finished, such as a network request or setTimeout. It is then placed into the **Callback/Event Queue**. Only after the Call Stack is empty, then items in the Event Queue are processed by the Event Loop - it is popped from the Queue and pushed to Stack.

JavaScript Timers

Timer methods in JavaScript are **higher order** functions that can be used to delay or repeat the execution of some other function.

`setTimeout(fn, number)`

Calls a function or executes a code after specified delay.

`setInterval(fn, number)`

Calls a function or executes a code repeatedly, with a fixed time delay between each call to that function. Returns an Interval ID

`clearInterval(intervalID)`

Cancels repeated action which was set up using `setInterval()`.

**mocking delays that we will see next week from making HTTP reqs or File Reads*

Visualizing the Event Loop

The **Event Loop** is a constantly running process that checks if the Call Stack is empty. It looks at the Call Stack and ***if*** it is empty it looks into the Event Queue. If there is something in the Event Queue that is waiting, then it is moved to the Call Stack. And if nothing is in the Event Queue, then nothing happens.

Visualizing the Event Loop

Event Loop JSConf YouTube

Callbacks

One approach to asynchronous programming is to make functions that perform slow actions take an extra argument, a **callback** function. The action is started, and when it finishes, the callback function is then called with the result.

Callbacks are an important feature of asynchronous programming. It enables the function that receives the callback to call our code when it finishes a long running task, while allowing us to continue the execution of other code.

TL;DR - A callback is a function to be executed after another function is done executing.

Promises

A **Promise** is an object which takes a callback. A promise specifies some code to be executed later (as with callbacks) and also explicitly indicates whether the code succeeded or failed at its job. You can chain promises together based on success or failure.

A Promise is in one of these states:

- *pending*: initial state, not fulfilled or rejected.
- *fulfilled*:   meaning that the operation completed successfully.
- *rejected*:   meaning that the operation failed.

Promises vs. Callbacks

Promises provide a clearer way of handling asynchronous operations. Promises are really just a different syntax for achieving the exact same effect as Callbacks.

The **advantages to Promises** are increased readability and providing a way of chaining functions. Then with the release of ES6 the ability to apply syntactic sugar using async/await.

It is important to note that Promises ***do not*** remove the use of callbacks. In fact, Promises utilize callbacks in a smarter way - the *resolve* and *reject* functions are callbacks that are just simply mapped to `Promise.then` and `Promise.catch`

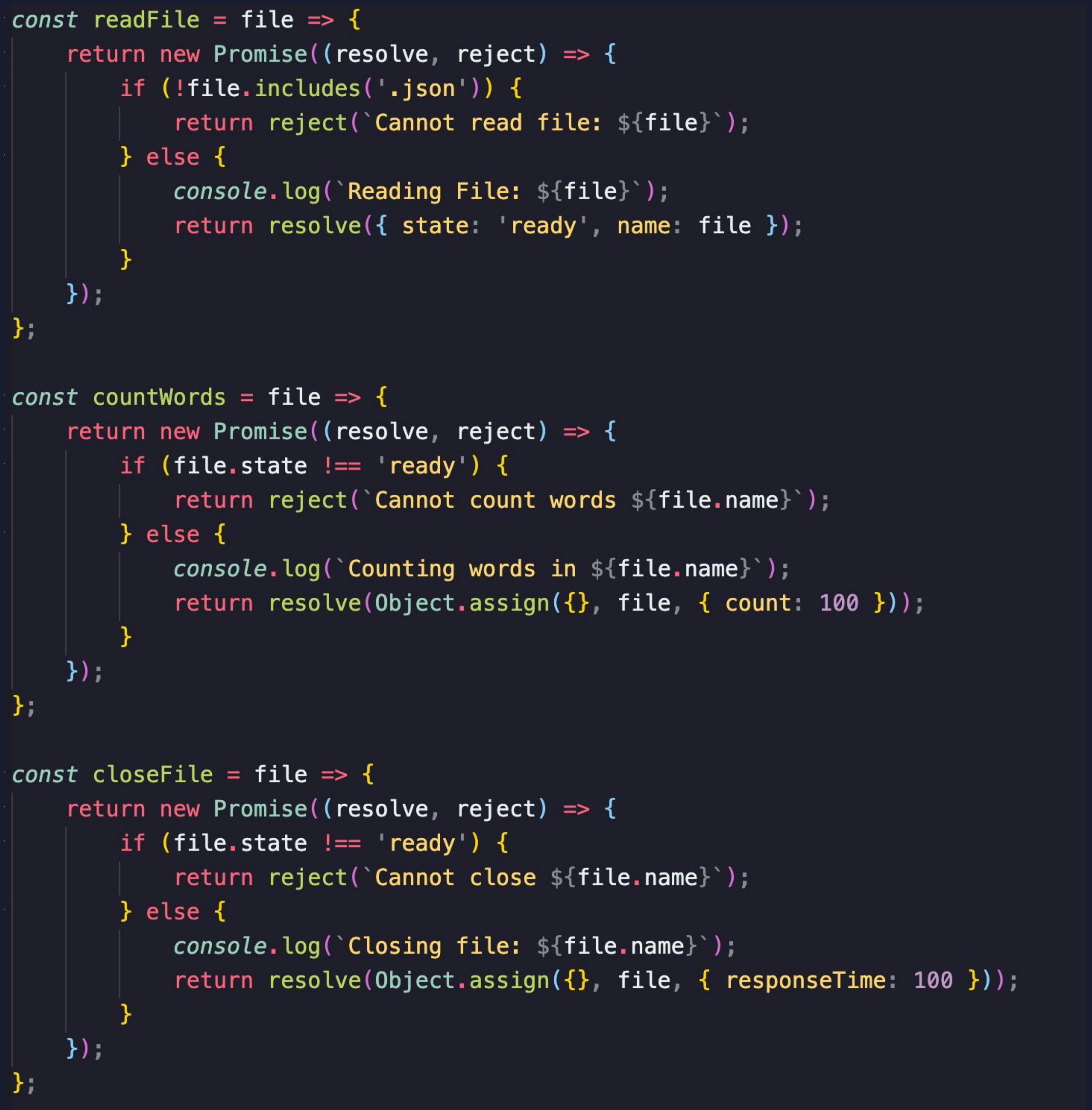
A Look at Callbacks

```js
const readFile = (file, callback) => {
    if (!file.includes('.json')) {
        callback(`Cannot read file: ${file}`);
    } else {
        console.log(`Reading File: ${file}`);
        callback(null, { state: 'ready', name: file });
    }
};

const countWords = (file, callback) => {
    if (file.state !== 'ready') {
        callback(`Cannot count words ${file.name}`);
    } else {
        console.log(`Counting words in ${file.name}`);
        callback(null, Object.assign({}, file, { count: 100 }));
    }
};

const closeFile = (file, callback) => {
    if (file.state !== 'ready') {
        callback(`Cannot close ${file.name}`);
    } else {
        console.log(`Closing file: ${file.name}`);
        callback(null, Object.assign({}, file, { responseTime: 100 }));
    }
};
```

Code available in `example.callbacks.js`

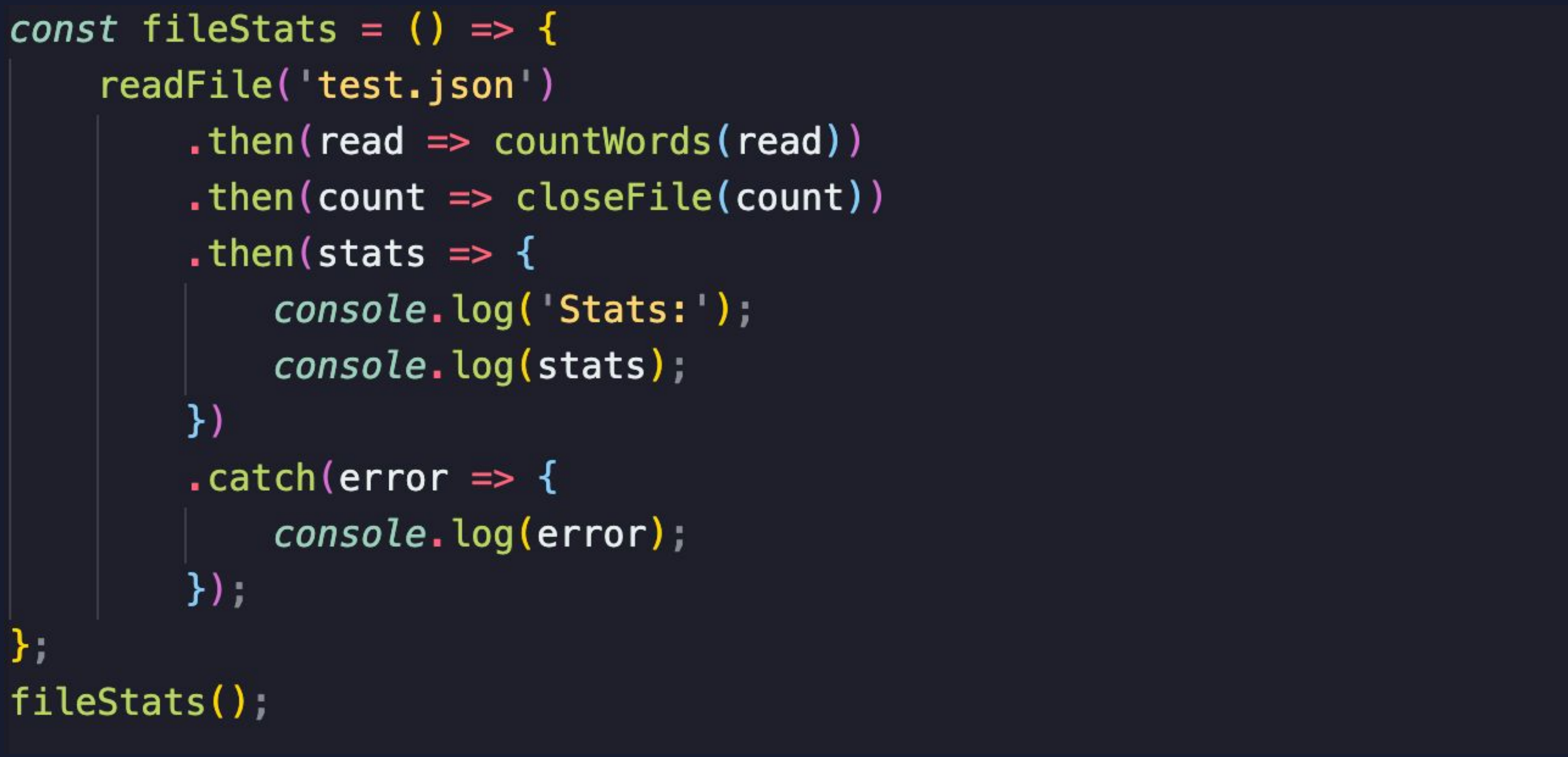
```js
const fileStats = () => {
    readFile('test.json', (error, file) => {
        if (!error) {
            countWords(file, (error, count) => {
                if (!error) {
                    closeFile(count, (error, stats) => {
                        if (!error) {
                            console.log(`Stats:`);
                            console.log(stats);
                        }
                    });
                }
            });
        } else {
            console.log(error);
        }
    });
};
fileStats();
```

A Look at Promises



```javascript
const readFile = file => {
    return new Promise((resolve, reject) => {
        if (!file.includes('.json')) {
            return reject(`Cannot read file: ${file}`);
        } else {
            console.log(`Reading File: ${file}`);
            return resolve({ state: 'ready', name: file });
        }
    });
};

const countWords = file => {
    return new Promise((resolve, reject) => {
        if (file.state !== 'ready') {
            return reject(`Cannot count words ${file.name}`);
        } else {
            console.log(`Counting words in ${file.name}`);
            return resolve(Object.assign({}, file, { count: 100 }));
        }
    });
};

const closeFile = file => {
    return new Promise((resolve, reject) => {
        if (file.state !== 'ready') {
            return reject(`Cannot close ${file.name}`);
        } else {
            console.log(`Closing file: ${file.name}`);
            return resolve(Object.assign({}, file, { responseTime: 100 }));
        }
    });
};
```

Code available in `example.promises.js`



```javascript
const fileStats = () => {
    readFile('test.json')
        .then(read => countWords(read))
        .then(count => closeFile(count))
        .then(stats => {
            console.log('Stats:');
            console.log(stats);
        })
        .catch(error => {
            console.log(error);
        });
};
fileStats();
```

Async/Await

Technically, it was ES8 that added the syntactic sugar on top of Promises in the form of **async/await.**

Using the keyword **async** before the function declaration defines it as an asynchronous function. These async functions return an implicit Promise.

An async function is **always paired** with an **await** expression. The await is used for calling a Promised based function and then waiting for it to be resolved or rejected . The await blocks the execution of the code only inside the async function. When using async/await - we also need to use **try/catch for error handling**.

A Look at Async/Await

```
const readFile = file => {
    return new Promise((resolve, reject) => {
        if (!file.includes('.json')) {
            return reject(`Cannot read file: ${file}`);
        } else {
            console.log(`Reading File: ${file}`);
            return resolve({ state: 'ready', name: file });
        }
    });
};

const countWords = file => {
    return new Promise((resolve, reject) => {
        if (file.state !== 'ready') {
            return reject(`Cannot count words ${file.name}`);
        } else {
            console.log(`Counting words in ${file.name}`);
            return resolve(Object.assign({}, file, { count: 100 }));
        }
    });
};

const closeFile = file => {
    return new Promise((resolve, reject) => {
        if (file.state !== 'ready') {
            return reject(`Cannot close ${file.name}`);
        } else {
            console.log(`Closing file: ${file.name}`);
            return resolve(Object.assign({}, file, { responseTime: 100 }));
        }
    });
};
```

Code available in `example.asyncawait.js`

```
const fileStats = async function() {
    try {
        const file = await readFile('test.json');
        const count = await countWords(file);
        const stats = await closeFile(count);

        console.log(`Stats:`);
        console.log(stats);
    } catch (error) {
        console.log(error);
    }
};
fileStats();
```

```
console.log(‘Week 05’);
console.log(‘Code Examples’);
```

Lab, Homework and Prep

Lab Time

- Homework Review
- Work on Homework 3

Preparation for Next Week

- Read Eloquent Javascript Chapters 20
- Watch YouTube Event Loop Video (Slide 6)