Asynchronous JavaScript

September 22

Today's Goal: learn about error handling, and the various ways to perform asynchronous code in JavaScript.

Errors/Exceptions in JavaScript

Up until now we've just used exceptions as something to help us debug our code. When something goes wrong, we check the console log, and sometimes there's some text highlighted in red for us to read.

Often times exceptions will stop code execution however, and that can be undesirable. When programs get very very large, there are a lot of places for something to go wrong, but a lot of other parts of the code unaffected by the errors that you want to keep running.

The solution is to *handle* the errors, and then even further to section off the errors using asynchronous code execution.

The `throw` keyword

You can throw your own exceptions using the `throw` keyword.

Example:

```
throw ‘some error’;
```

Or even better:

```
throw new Error(‘some error’);
```

`try…catch` statements

You can use `try…catch` statements to handle the errors in your code.

`try` the code that might potentially `throw` an error, and then `catch` that error to *handle* it and let the unaffected rest of the code to continue running.

Example:

```
try {
  throw ‘some error’;
} catch (e) {
  console.log(e);
}
```

The optional finally block

There is an optional third block you can add to a try…catch statement, and that is the finally block that allows you to run some code after the catch block finishes its job. This might not seem too handy until you start working with incredibly complicated catch blocks.

This is one of the many ways that modern JavaScript naturally sneaks in asynchronous code execution where the developer doesn't have to think about it. You've already encountered this many times before with event listeners and callback functions.

Callback Functions and `setTimeout()`

Callback functions, such as when you use them for event listeners, run asynchronously.

setTimeout() is a built in browser function that allows you to run any given function as a callback after a certain amount of time. We can combine this with a timer of 0 to essentially force any function to run asynchronously.

```
function myFunction() { … }
setTimeout(myFunction, 0);
```

Promises

Promises are the modern (ES6+) way of handling asynchronous code execution in JavaScript, and they provide better error handling than callback functions.

Promises are chained together using `.then()` syntax, and errors are handled with a `.catch()` clause at the end of the chain.

Example:

```
doSomethingAsynchronously()
  .then(function(result) {
    return doSomethingElseAsynchronously(result);
  })
  .then(function(otherResult) {
    return doSomeOtherThingAsynchronously(otherResult);
  })
  .catch(function(error) {
    // handle the error somehow
  })
```

`async/await` keywords

Chaining together code with promises can feel unnatural compared to how we are used to using synchronous functions.

`async` functions are a level of “syntax sugar” over normal promises, that when combined with the `await` keyword, help to bridge the gap between our understanding of synchronous and asynchronous code.

Creating Asynchronous Code Execution Loops

You may have noticed that you can chain together multiple `setTimeout()`s into existing `setTimeout()`s to create a sort of asynchronous recursion through a delay. Even better than that, you can use a very similarly named function `setInterval()`.

But if you are using this loop to create some sort of animation, or update the DOM very quickly and regularly, using `setInterval()` can result in your animation falling out of sync with the browser's render cycle. For animations and other actions that require a lot of computational resources, the built in browser function `requestAnimationFrame()` is your best bet.