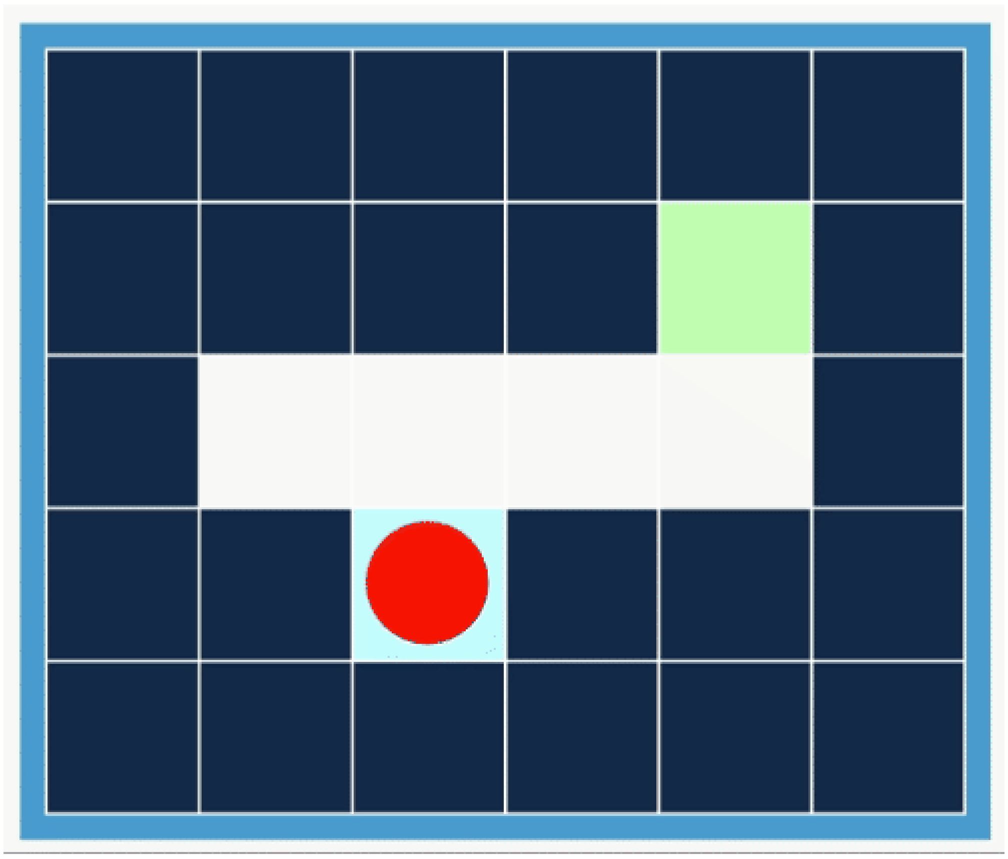
Assignment 6

Contents

In this assignment, we will be exploring different approaches to *asynchronous programming* in JavaScript. For this discussion, we are concerned with asking the computer to perform tasks that *take time to complete*. To illustrate this concept, imagine you are controlling a token as it moves through a simple on-screen maze, like the one shown below.



The goal of the game is to move the token from the starting blue shaded square to the ending green shaded square. To do this, we have to move the token up one square, to the right twice, and then up one square again.

In the code, the token is represented as an object which has a `move()` method that you can use to control where the token goes. The `move()` method takes a string parameter that specifies which direction to go. Here's some pseudocode, then, for a solution to the maze:

```
const maze  = new Maze();       // Create a new maze
const token = new Token(maze); // Create a new token in the maze

token.move('north'); // Move the token one square north
token.move('east');  // Move the token one square east
token.move('east');  // Move the token one square east
token.move('north'); // Move the token one square north
```

> If we needed to move the token south or west, we could have done so with `token.move('south')` or `token.move('west')`.

So how does this example help us introduce the idea of asynchronous programming? As shown in the GIF above, every move of the token actually represents a process that *takes time to complete*. For about two seconds after `token.move('north')` is called, the token is actually *in transit* between its initial square and the square above it. During the two-second transit time, your JavaScript must essentially "stall" on that line of code while it waits for the movement to complete before it can continue on to the next move.

The idea of having your program "stall" on each `move()` method call has the immediate benefit that the code it produces is particularly easy to read. The algorithm we're describing is a procedural, step-by-step process: go north, *and then afterwards* go right, *and then afterwards* go right, *and then afterwards* go north. Each step needs to wait for the last one to finish, and no two steps can be performed simultaneously. This one-step-at-a-time requirement is easily met if each call to the `move()` method stalls until it is finished. When a function or method "stalls" until it is finished (like the `move()` method), we say that it is **synchronous**.

> Functions or methods that "stall" until they are finished are called **synchronous**.

Part 1: Maze

1. Synchronous solution

This brings us to our first attempt at writing a solution to the maze: the **synchronous solution**.

1. In Visual Studio Code, open a new terminal in the `comp426-cli/a06` folder (`Right click a06 -> Open In Terminal`) and run `browser-sync start -sw` to start Browsersync.

   Note: If you get a `command not found` error, this means you have not installed Browsersync as a global npm package. Go back to Section 5.4 in Assignment a02 to see how to install packages globally, and follow the instructions to install Browsersync as a global npm package.

2. Browsersync should automatically open a new browser tab showing links for each section of this assignment. Click on the link labeled "Synchronous solution."

3. Your browser window should now show the maze from the GIF. The red token should appear, unmoving, at the maze's initial position.

4. Go back to Visual Studio Code and open the file `a06/s01-synchronous/controller.js`. You'll notice a `TODO` comment marking the point where you need to add the moves to solve the maze.

5. Add the following moves underneath the `TODO` comment:

```
token.moveSync('north');
token.moveSync('east');
token.moveSync('east');
token.moveSync('north');
```

> Notice that instead of naming the method `move()`, we're naming it `moveSync()`. Don't let this confuse you; it's just to make it obvious that this is a **synchronous** version of the move method.

6. Save the file and switch back to the browser to see what happened. You may want to refresh the page a few times to be sure that you see what's going on.

If you did it correctly, **you should see a blank white page for about 8 seconds, and then the maze should appear with the red token at the final position**.

So what happened? Why doesn't it show the moves?

The answer is due to the fact that **JavaScript is a single-threaded language**. By definition, the browser's JavaScript interpreter can only ever do one thing at a time---it can't even run your code and update the DOM at the same time! When your code is running, the browser has to wait for you to finish before updating the DOM. Similarly, when the DOM is updating, the browser has to wait for it to finish before running your code.

When you call a synchronous function, you are hogging the browser's only JavaScript thread until your code completely finishes running. That means the browser doesn't even get computation time to update the DOM until after your code finishes. Since each token move takes about two seconds and there are four moves in total, this means your code is monopolizing the JavaScript execution thread for about eight consecutive seconds. During this time, the browser can't update how the page looks and it therefore appears blank and locked up. As soon as your code finishes making its moves, the browser gets access to the execution thread and uses it to update the page. But by this time, the moves are already complete and the token is at the final position!

The main takeaway of the synchronous approach is that it allows you to write very straightforward, clean code, but it comes with the major limitation of locking up the active thread until the process is complete.

2. An asynchronous attempt

Obviously, the synchronous approach we tried in Section 1 is unsuitable for our use case. We need a way to allow the token to move over a period of time, while still freeing the execution thread to update the DOM.

What we are describing is an **asynchronous** version of the move method. When you make a call to an asynchronous function or method, you are signaling that you'd like that code to start running, but you don't want to wait around for it to finish. The idea of running code asynchronously is a **huge** part of programming in JavaScript, because synchronous code inherently locks up the browser window---which results in a bad user experience.

> Functions or methods that initiate a procedure or process but return control back to the caller before the procedure finishes are called **asynchronous**.

In addition to the `moveSync()` method that we used in the last section, the `Token` class also exposes a `moveAsync()` method. When `moveAsync()` is called, the token kicks off a new move action just like before. The difference is that instead of "hanging" until the move is complete (which is what the `moveSync()` method did), the `moveAsync()` method *immediately returns back to your code*, even before the move action has completed. This gives you, the programmer, the flexibility to decide what should happen in the meantime while the move is taking place.

Let's try using the `moveAsync()` function.

1. In Visual Studio Code, open the file `a06/s02-asynchronous/controller.js`. You'll notice a `TODO` comment marking the point where you need to add the asynchronous moves to solve the maze.

2. Add the following moves underneath the `TODO` comment, and then save the file:

```
token.moveAsync('north');
token.moveAsync('east');
token.moveAsync('east');
token.moveAsync('north');
```

3. In Google Chrome, go back to the assignment index page and click on the link labeled "An asynchronous attempt." Pay close attention to what happens, and refresh the page if you need to see it again!

If you did it correctly, **you should see the red token perform only the first move, and then get stuck**.

What is happening this time? You may be able to guess. You told the token to *asynchronously* make four moves. Since the methods are asynchronous, they don't "hang" while waiting to complete. That means **the second move is being launched before the first one even completes**. The `Token` class isn't set up to allow for multiple moves to take place at the same time. Instead, **the second, third, and fourth move requests error out**, because they detect that the token is still in motion. As a result, only the first move is performed.

Google Chrome provides a way to see all JavaScript errors that occur while your web page is active. They appear in a special window called the JavaScript console. Here's how to use it to see the move errors:

1. While looking at the asynchronous maze web page in Google Chrome, press `Ctrl + Shift + J` (Windows) or `Cmd + Option + J` (Mac). The JavaScript console window should open.
2. Refresh the page.
3. You should see three red errors appear in the console, corresponding to the three failed moves.

   Browsers like Google Chrome have a special built-in window, called "DevTools", which can be opened by pressing the `F12` key on your keyboard. This tool is incredibly valuable for web developers. It provides real-time access to the currently loaded DOM, CSS, HTTP requests, a JavaScript console, and more. Definitely expect to rely on Chrome DevTools heavily in your future web dev career!

The problem we're running into is that simply making our code asynchronous is not enough. Inherently, we are *still performing a procedural task* that must take place one step at a time. We need a way to wait for the last asynchronous move to complete before starting the next one---all without locking up the JavaScript thread.

3. Event handlers

At this point, we've demonstrated the major limitation of synchronous programming in JavaScript as well as the pitfall of asynchronous methods. JavaScript has been around for decades though, and this synchronous/asynchronous thing has been present since the beginning. As a result, clever programmers have devised many techniques for solving the problem. We're going to go through the most common approaches.

The goal is to use an asynchronous method (like `moveAsync()`), but introduce a way to wait until one move finishes before starting the next one. One way to do this is with an **event handler**.

Imagine that you define a special function, called an *event handler*, that is stored inside the `Token` class and is executed automatically by the token every time it finishes moving. This is an example of *event-based programming*. The "event" in question is the action of the token finishing its move. Every time this event occurs, the associated event handler function is called. As the programmer, you get to fill in this event handler function, so you get to specify what exactly happens in response to the event occurring. In our case, when the "finish move" event occurs, we want to start the next move.

Here's what this might look like in JavaScript:

```
const moves = ['north', 'east', 'east', 'north'];
token.moveAsync(moves.shift());
token.onmoveend = function() {
    if (moves.length > 0) {
        token.moveAsync(moves.shift());
    }
};
```

Notice that we're defining a special property on the token, called `onmoveend`, and assigning it to be function. That function is called the *event handler*. What does the onmoveend event handler do? Well, if there are still moves left to make, then it starts the next move. This time we're storing the moves in an array so that we can execute the same event handler function every time, even though the moves we make may change.

Another thing to notice is that there has to be an initial call to `moveAsync()` to start off the whole process. That call occurs on line 2. By the time the initial move from line 2 finishes, the `onmoveend` event handler function will have been set, allowing the next move to be initiated.

> Notice that we're using `Array.shift()` to mutate the `moves` array, essentially treating it as a queue of upcoming moves. Read more about `Array.shift()` here.

Copy this code into `a06/s03-eventhandler/controller.js` directly below the `TODO` comment, save the file, and navigate to the "Event handlers" link in Google Chrome.

Finally! You should see the token successfully navigate through the maze.

4. Event listeners

You might remember the observer/observable design pattern in Java from your COMP 401 days. You might also have noticed that the event handler idea that we just implemented in Section 3 very closely resembles the observer/observable pattern: the token is like an observable object, and every time a move finishes, it is responsible for notifying its observer by executing the onmoveend event handler function.

This is a very astute observation; in fact, these two concepts are very closely related! The big difference is that in Section 3, there is only room for a single observer---that is, we can only define one `onmoveend` event handler function. With the classical observer/observable pattern, the observable maintains an array of observers, and notifies all of them when an event occurs. This is an all-around more flexible approach.

In fact, many existing JavaScript libraries, including jQuery, also take this approach. The standard syntax is for an observable object to expose an `on(event, callback)` function which allows the programmer to specify (1) which event they want to listen to as a string and (2) register an observer callback function for that event.

The `Token` class is also set up to accept this syntax. Here's an example of a solution to the maze problem that registers an event listener via the `on()` method:

```
const moves = ['north', 'east', 'east', 'north'];
token.moveAsync(moves.shift());
token.on('moveEnd', function() {
    if (moves.length > 0) {
        token.moveAsync(moves.shift());
    }
});
```

This code defines the same exact callback function as in Section 3, which is meant to be executed every time the token finishes a move. The only difference is that this time the callback is passed to the token object via the `on()` method. The `on()` method adds the callback function to an internal array of listeners stored inside the observable token object.

Sections 3 and 4 both work by defining a callback function that is executed once the move completes. The only difference between them is in the syntax by which the callback is passed to the token object. In Section 3, the callback is assigned to the `onmoveend` property; in Section 4, it is passed via the `on()` method. Both of these syntactic approaches are commonly seen in JavaScript. Which one you use depends on the library.

Copy the above code into `a06/s04-eventlistener/controller.js` directly below the `TODO` comment, save the file, and navigate to the "Event listeners" link in Google Chrome. You should again see the red token successfully navigate the maze. Refresh the browser window if you want to see it again.

5. Promise nesting

So far, we've achieved asynchronous programming by registering callback functions as listeners which are executed whenever events occur. However, it's clear that event-based programming still isn't as straightforward or as easy to read as the code in the synchronous example we started with.

The next few sections will introduce a new programming construct called *promises* which make asynchronous programming as easy as the synchronous example.

While the idea of promises has been around for decades, it is relatively new to JavaScript. In fact, promises were only first introduced to JavaScript a few years ago in 2015, as part of the latest version of JavaScript, ES6.

A **promise** is a special JavaScript object that is created to represent an asynchronous operation that has not yet finished executing. At any point, a promise is in one of three possible states:

- *pending* - indicates that the operation is still in progress
- *fulfilled* - indicates that the operation completed successfully
- *rejected* - indicates that the operation failed

More importantly, a promise object exposes two very important methods:

- `then(callback)` - Registers a callback function as a listener to the operation. The callback function will be executed by the promise upon successful completion of the operation.
- `catch(callback)` - Registers a callback function as a listener to the operation. The callback function will be executed by the promise if the operation fails for any reason.

Now that promises are officially part of JavaScript, you'll find that they are the preferred abstraction for dealing with asynchronous operations. You can read more about promises here.

The `moveAsync()` function we've been using returns a promise object. Here's an example of how it may be used:

```
token.moveAsync('north').then(function() {
        console.log('This callback will run after the move operation finishes');
});
```

See how the `then()` method was chained right onto the end of the `moveAsync()` method? This is because `moveAsync()` returned a promise object, on which the `then()` method was called. The promise paradigm is nice because it reads like English: "*move the token north, then log to the console*."

Here's how we might use promises to write an asynchronous solution to the maze:

```
token.moveAsync('north').then(function() {
    token.moveAsync('east').then(function() {
        token.moveAsync('east').then(function() {
            token.moveAsync('north');
        });
    });
});
```

Copy this code into `a06/s05-promisenesting/controller.js` directly below the `TODO` comment, save the file, and navigate to the "Promise nesting" link in Google Chrome. You should again see the red token successfully navigate the maze.

6. Promise chaining

One ugly characteristic of the promise code from the previous section is how the callbacks are nested inside each other. This creates annoying indentation issues and makes it easy to miss a closing brace.

To solve this problem, the `Promise.then()` method itself returns a promise object. If the callback passed to `then()` returns a promise, then that promise will be resolved before the promise callback chain is allowed to continue. This is a bit confusing, so here's an example to help:

```
token.moveAsync('north').then(function() {
    return token.moveAsync('east');
}).then(function() {
    return token.moveAsync('east');
}).then(function() {
    return token.moveAsync('north');
});
```

In this code, the move callbacks are chained together as opposed to the prior section, in which the callbacks were nested. **Chaining the callbacks in this way only works when the promise returned by each `moveAsync()` is returned from the `then()` callback**. This code is exactly equivalent to what we saw in Section 5.

Copy this code into `a06/s06-promisechaining/controller.js` directly below the `TODO` comment, save the file, and navigate to the "Promise chaining" link in Google Chrome. You should again see the red token successfully navigate the maze.

7. Async-await

Phew! By now, we've seen *lots* of different techniques for dealing with asynchronous operations in JavaScript. And while each has been easier than the last, none have been *quite* as nice as the original synchronous solution that we started with.

It's time to change that. One of the most widely acclaimed features of ES6 (the newest standard for JavaScript) is a new language construct called async/await, which is essentially syntactic sugar for promises that makes them look and behave exactly as if they were synchronous. Seriously, programmers almost universally love this new syntax.

Here's how it works. Anytime you're dealing with an asynchronous function that returns a promise, and you want to wait until the promise finishes before continuing, just add the keyword `await` in front of it. That's enough to tell JavaScript to asynchronously wait for the promise before continuing on with your code. It won't lock up the execution thread and you'll be good to go.

```
await token.moveAsync('north');
// code here won't execute until the moveAsync() promise resolves
```

There is a catch, however. It's a syntax error to use `await` out in the open. **If you want to use the `await` keyword in front of a method or function call, you are required to put the code inside a function labeled `async`.** We aren't going to get in the details of why this is necessary, but essentially it means that an async/await solution to the maze problem requires an extra function definition:

```
const doMoves = async function() {
    await token.moveAsync('north');
    await token.moveAsync('east');
    await token.moveAsync('east');
    await token.moveAsync('north');
};
doMoves();
```

See how the new `doMoves()` function is labeled with the `async` keyword? This is what makes it okay to use `await`.

> All functions or methods labeled `async` actually return a `Promise` object under the hood. This means that you can add `await` in front of any call to an `async` function or method. For example, you could add `await` in front of `doMoves()`. This would force the program to wait for the `doMoves()` function to finish before execution would continue.

Hopefully you appreciate how nice this syntax is. It abstracts something that's actually quite complicated (asynchronous programming) behind a few new keywords.

Copy this code into `a06/s07-asyncawait/controller.js` directly below the `TODO` comment, save the file, and navigate to the "Async-await" link in Google Chrome. You should again see the red token successfully navigate the maze.

Part 2: Exercises

PULL THE LATEST CHANGES TO YOUR CLI

Now we are going to write some code that demonstrates the functionality of callbacks, promises, and async/await. Head over do you workspace folder. Inside `a06/exercises` you will find the following files:

1. `1_callbacks.js`
2. `2_promises.js`
3. `3_async_await.js`
4. `4_challenge.js`(extra credit)

If you dont see exactly these files pull the changes to you CLI.

2.1 Callbacks/Promises using SetTimeout

Because we aren't interacting with a server or anything that takes any measurable amount of time to respond, we need to simulate an instance where we need asynchronous execution. `SetTimeout()` is used to execute a block of code after a certain amount of delay.

> The `setTimeout()` method calls a function or evaluates an expression after a specified number of milliseconds.
>
> - 1000 ms = 1 second.
> - The function is only executed once. If you need to repeat execution, use the `setInterval()` method.
> - Use the `clearTimeout()` method to prevent the function from running.

You will use this function to simulate a delay of 1.5 seconds (1500 ms).

example:

```
console.log('before');
setTimeout(() => {
  console.log('inside');
}, 1500);
console.log('after');
```

outputs

```
before
after
inside
```

2.2 Callbacks

In a callback you can simply use `setTimeout()` and have the callback call inside. Here is a simple demo of how that might look.

```
function squareCallback(aNumber, theCallback) {
  console.log(`Before calling: ${aNumber}`);

  // waits 3 seconds and then uses the callback
  setTimeout(() => {
  console.log(`3 seconds has gone by...`);
  theCallback(aNumber ** 2);
 }, 3000);

}
```

This is how you could call `squareCallback` with a normal function:

```
squareCallback(4, function (result) {
console.log(`Got back ${result}`);
});
```

Or with an arrow function:

```
squareCallback(5, (str) => {
  console.log(`Arrow function: ${str}`);
});
```

Or we could define a function to be used as a callback:

```
// define our callback
function beingPassed(theResult) {
 console.log(`Number is: ${theResult}`);
}
```

```
// we can call this however we like
beingPassed(22);

// or use it as the callback function
squareCallback(12, beingPassed);
```

2.3 Promises

Promises do the same thing as callbacks but in a much cleaner way (this is an opinion, and async/await really helped). In the following example you will see the exact same scenario as above, but this time using promises. You might not have seen how promises are actually created before now. Notice how we are using the `new` keyword.

So first we are going to define our function that calculates the square of a number. We no longer need the callback parameter now because we are using promises. It might look odd that we are returning something straight away. Just remember that when you call this function you are given back a promise. A promise is an object that will eventually become a value or an error at some point in the future. That is why we use the `.then()` and `.catch()`.

```
function squarePromise(aNumber) {
  // First thing we need to do is return a promise.
  return new Promise(((resolve, reject) => {
    // Resolve is used as a callback on a success
    // Reject is used as a callback on a failure
    setTimeout(() => {
      if (aNumber < 0) {
        reject(`Don't pass me negative numbers please.`);
      } else {
        resolve(aNumber ** 2);
      }
    }, 3000);

  }));
}
```

We could call it like either of these two ways:

```
squarePromise(4)
  .then(num => console.log(num));

squarePromise(4)
  .then(num => {
    console.log(num);
  });
```

Or if we are expecting an error (which you should almost always expect):

```
squarePromise(-3)
  .then(num => console.log(num))
  .catch(err => console.log(err));
```

2.4 Async/Await

Async/Await is just ***syntactic sugar*** for promises. So we wont change the function from above, but we would call it differently. The reason this example looks a little odd is because in order to make a call with await, you have to be inside a function. So we had to wrap it a dummy function that does nothing. This usually isn't a problem.

```
async function makeTheCall() {
  const result = await squarePromise(5);
  console.log(result);
}

makeTheCall();
```