Asynchronous JavaScript

(https://github.com/learn-co-curriculum/phase-1-asynchronous-javascript) (https://github.com/learn-co-curriculum/phase-1-asynchronous-javascript/issues/new)

Learning Goals

- Establish a metaphor for synchronous versus asynchronous work
- Describe a synchronous code block
- Describe an asynchronous code block
- Identify a synchronous code block
- Identify an asynchronous code block

Introduction

Browsers have to manage a lot. They're animating a `gif`, they're displaying text, they're listening for clicks and scrolls, they're streaming a SoundCloud demo in a background tab, and they're running JavaScript programs.

To do all that work efficiently, browsers use an *asynchronous* execution model. That's a fancy way of saying "they do little bits of lots of tasks until the tasks are done."

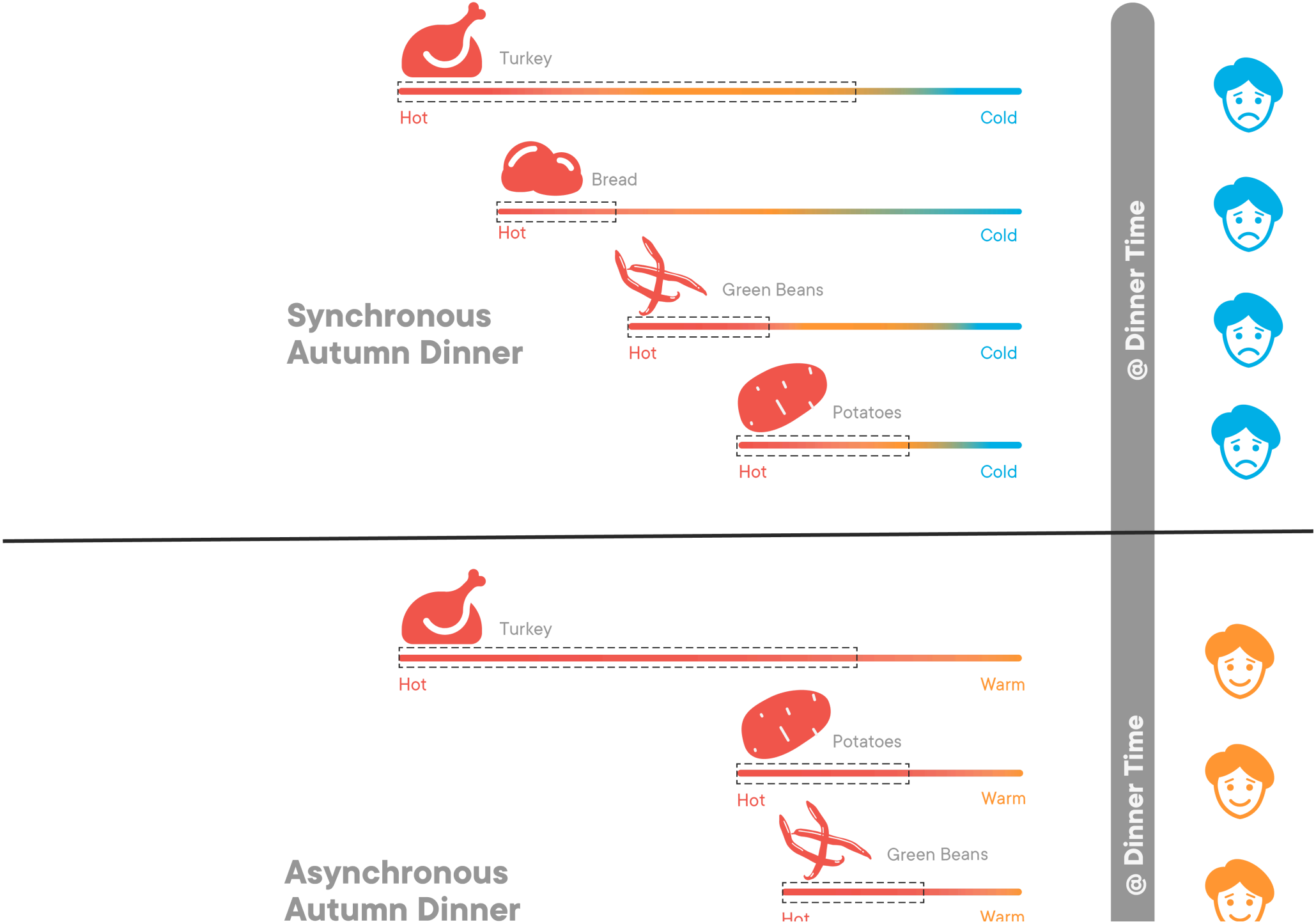
In this lesson we'll build a foundation of understanding around the asynchronous execution model of JavaScript.

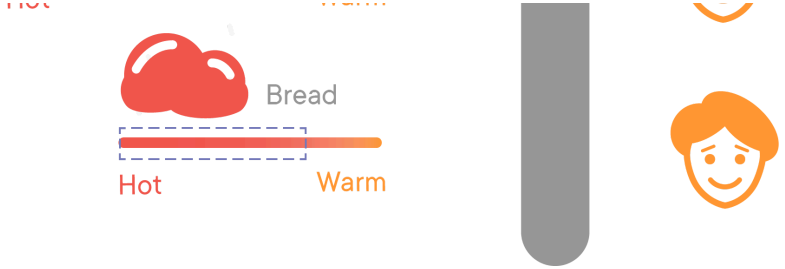
Establish a Metaphor for Synchronous Versus Asynchronous Work

Let's imagine a chef in a kitchen preparing a big meal. There's only one chef in this kitchen. The chef could prepare a turkey, then prepare some potatoes, then prepare some bread, then prepare green beans, and then serve it.

Our diners would be treated to cold turkey, cold bread, cold green beans, and cold potatoes! This is not the goal. This meal was prepared in a *synchronous* model: one-thing-after-the-other. Whatever happened "blocked" the rest of things that were waiting for work.

Instead, our chef should move between each of these tasks quickly. The chef should use the *asynchronous* execution model browsers use. They should stuff the Turkey, they should measure the ingredients for the bread, they should peel the potatoes, etc. in a loop, *as fast as possible* so that all the tasks *seem* to be advancing at the same time. If the chef were to adopt this *asynchronous* model of work, the diners would be treated to piping-hot turkey, steaming potatoes, soft warm bread, and fresh warm green beans.


Turkey
Hot
Cold
Bread
Hot
Cold
Green Beans
Hot
Cold
Potatoes
Hot
Cold
Synchronous
Autumn Dinner
@ Dinner Time
Turkey
Hot
Warm
Potatoes
Hot
Warm
Green Beans
Hot
Warm
Asynchronous
Autumn Dinner
@ Dinner Time



Describe a Synchronous Code Block

So far in JavaScript, we've mostly written *synchronous* code where the execution model didn't matter.

```
const sum = 1 + 1; // Line 1
const lis = document.querySelectorAll("li"); // Line 2
```

In this case, when we hit the definition of `sum`, this work doesn't rely on any "unknowably long" process. As soon as the work of `Line 1` is done, JavaScript will then go to work finding elements and assigning them to `lis` in Line 2.

But let's consider a "blocking" operation. Imagine we had a synchronous function called `synchronousFetch("URL STRING")` that fetches data from the network.

```
const tooMuchData = synchronousFetch("http://genome.example.com/..."); // Line 1
const lis = document.querySelectorAll("li"); // Line 2
console.log(tooMuchData);
```

That work in Line 1 could take a long time (e.g. slow network), or might fail (e.g. failed login), or might retrieve a ***huge*** amount of data (e.g. The Human Genome).

With this synchronous approach, JavaScript won't continue to the next line of code until `synchronousFetch` has finished executing, so it's possible that the `const lis` in Line 2 *will never execute*! Furthermore, while JavaScript is executing `synchronousFetch` it will not be able to animate gifs, you won't be able to open a new tab, it will stop streaming SoundCloud, it will appear "locked up." Recall our chef metaphor: while the chef prepares the potatoes, the green beans grow cold and the turkey congeals. Gross.

Describe an Asynchronous Code Block

Asynchronous code in JavaScript looks a lot like event handlers. And if we think about it, that makes sense. You tell JavaScript:

> Hey, do this thing. While you're waiting for that to finish, go do whatever maintenance you need: animate that gif, play some audio from SoundCloud, whatever. But when that first thing has an "I'm done" event, go **back** to it and *then* do some work that I defined in a function when I called it.

Let's imagine a function called `asynchronousFetch` that takes two arguments:

- A URL String
- A callback function that will have the fetched data passed into it as its first argument when the `asynchronousFetch` work is done

```
asynchronousFetch("http://genome.example.com/...", tonOfGeneticData => sequenceClone(tonOfGeneticData)); // Line 1
const lis = document.querySelectorAll("li"); // Line 2
```

In this case, JavaScript *starts* the `asynchronousFetch` in Line 1, and then sets `lis` in Line 2. Some time later (who knows how long?), the fetch of data finishes and *that* data is passed into the "callback" function as `tonOfGeneticData` — back on Line 1.

Most asynchronous functions in JavaScript have this quality of "being passed a callback function." It's a helpful tool for spotting asynchronous code "in the wild."

Let's try seeing how synchronous versus asynchronous works in real JavaScript code.

Identify a Synchronous Code Block

As we have experienced in JavaScript, our code executes top-to-bottom, left-to-right.

```
function getData(){
  console.log("2. Returning instantly available data.");
  return [{author: "Ta-Nehisi Coates"}, {author: "Cathy Park Hong"}];
}
```

```
function main(){
  console.log("1. Starting Script");
  const data = getData();
  console.log(`3. Data is currently ${JSON.stringify(data)}`);
  console.log("4. Script Ended");
}

main();
```

We can copy and paste this into a DevTools console to see the result. It matches our default model of "how code runs."

Identify an Asynchronous Code Block

The easiest asynchronous wrapper function is `window.setTimeout()` (https://www.w3schools.com/jsref/met_win_settimeout.asp). It takes as arguments:

- a `Function` (the "callback" function)
- a `Number` representing milliseconds

The `setTimeout()` will wait the specified number of milliseconds and then execute the callback.

```
setTimeout(() => console.log('Hello World!'), 2000);
```

This says "Hello World!"... in 2 seconds. Try it out in the DevTools console!

Since this code is in an *asynchronous* container, JavaScript can do other work and *come back* when the work "on the back-burner" is done. If JavaScript *didn't* have an asynchronous model, while you waited those 2 seconds, no gifs would animate and streaming audio might stall. Asynchronous execution makes browsers the exceedingly useful tools they are.

What do you think the output will be here?

```
setTimeout(() => console.log('Hello World!'), 2000);
console.log("No, me first");
```

Sure enough:

```
No, me first
Hello World!
```

JavaScript is so committed to trying to squeeze in work when it gets a chance that this has the exact same output!

```
setTimeout(() => console.log('Hello World!'), 0); // 0 Milliseconds!!
console.log("No, me first");
```

Here the browser has < 0 milliseconds (i.e. nanoseconds) to see if it can find any work to do — and it still does!

Conclusion

JavaScript in the browser has an asynchronous execution model. This fact has little impact when you're writing simple code, but when you start doing work that might block the browser you'll need to leverage asynchronous functions. Remember, these functions can be surprising and nearly every JavaScript developer sooner or later forgets to reckon with asynchrony.

While working asynchronously can be a bit of a headache for developers, it allows JavaScript to do other work whenever it has an opportunity. Important methods which require us to think asynchronously are `setTimeout()` and `fetch()`, among others.