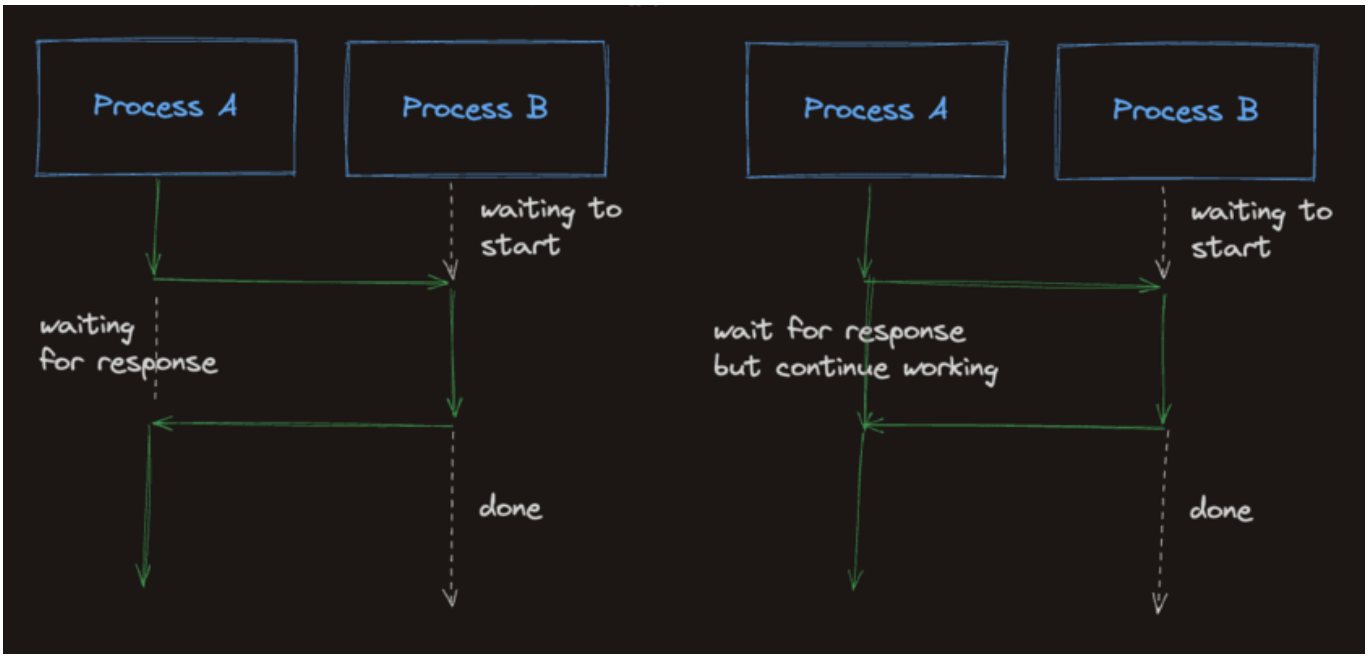
Synchronous / Asynchronous

Synchronous vs Asynchronous

Synchronous code means code that runs in sequence. Each line of code executes in order, one after the next.



Example of synchronous code:

```
console.log("I print first");
console.log("I print second");
console.log("I print third");
```

Asynchronous or async code runs concurrently. That means while the main thread continues running subsequent code, the async tasks are handled outside the main execution flow and are completed as system resources allow. A good way to visualize this is with the JavaScript function setTimeout().

`setTimeout` accepts a function and a number of milliseconds as inputs. It sets aside the function to be run after the number of milliseconds has passed, at which point it gets queued for execution when the main thread is available.

Example of asynchronous code:

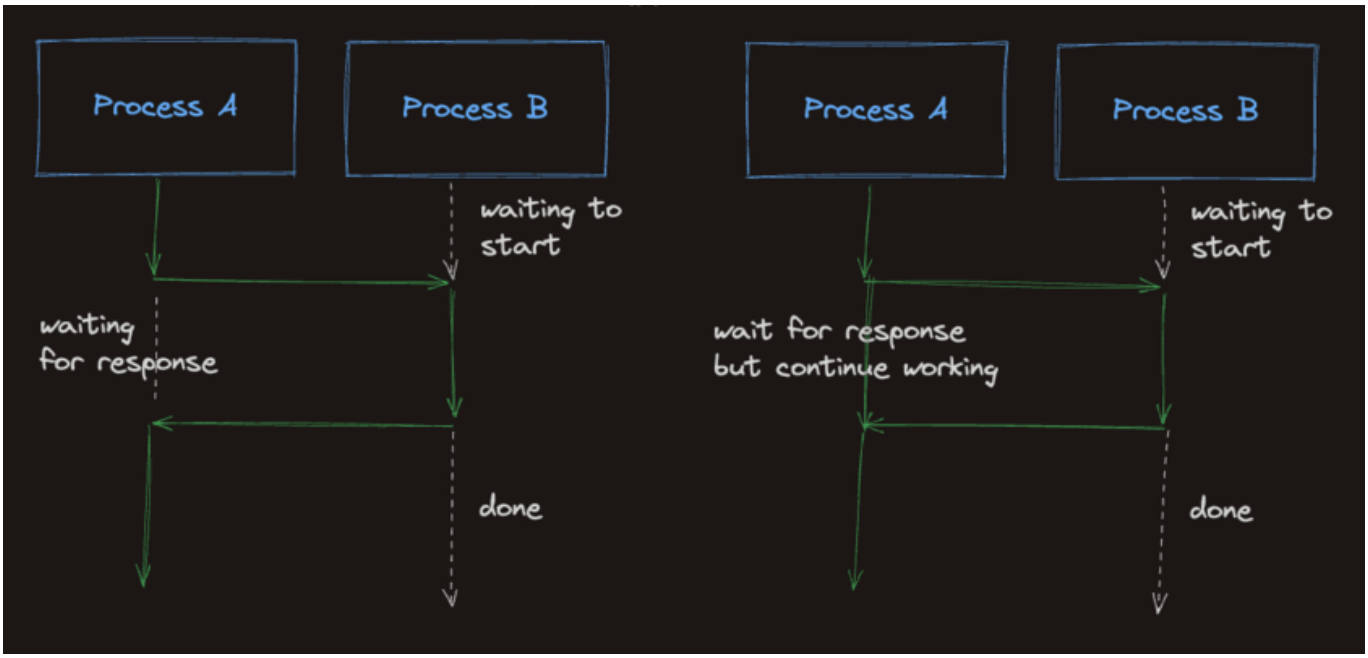
```
console.log("I print first");
setTimeout(() => console.log("I print third because I'm waiting 100
milliseconds"), 100);
console.log("I print second");
```

Why do we want async code?

We try to keep most of our code synchronous because it's easier to understand, and therefore often has fewer bugs. However, sometimes we need our code to be asynchronous. For example, whenever you update your user settings on a website, your browser will need to communicate those new settings to the server. The time it takes your HTTP request to physically travel across all the wiring of the internet is usually around 100 milliseconds. It would be a very poor experience if your webpage were to freeze while waiting for the network request to finish. You wouldn't even be able to move the mouse while waiting!

By making network requests asynchronously, we let the webpage execute other code while waiting for the HTTP response to come back. This keeps the user experience snappy and user-friendly.

As a general rule, we should only use async code when we need to for performance reasons. Synchronous code is simpler.



Promises in JavaScript

A Promise in JavaScript is very similar to making a promise in the real world. When we make a promise we are making a commitment to something. For example, I promise to explain JavaScript promises to you, my promise to you has 2 potential outcomes: it is either fulfilled, meaning I eventually explained promises to you, or it is rejected meaning I failed to keep my promise.

The Promise Object represents the eventual fulfillment or rejection of our promise and holds the resulting values. In the meantime, while we're waiting for the promise to be fulfilled, our code continues executing. Promises are the most popular modern way to write asynchronous code in JavaScript.

Declaring a Promise

Here is an example of a promise that will resolve and return the string "resolved!" or reject and return the string "rejected!" after 1 second.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (getRandomBool()) {
```

```
      resolve("resolved!")
    } else {
      reject("rejected!")
    }
  }, 1000)
})

function getRandomBool(){
  return Math.random() < .5
}
```

Using a promise

Now that we've created a promise, how do we use it?

The `Promise` object has .then and [].catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) that make it easy to work with. Think of .then as the expected follow-up to a promise, and .catch as the "something went wrong" follow-up.

If a promise resolves, its `.then` function will execute. If the promise rejects its .catch`` method will execute.

Here's an example of using `.then` and `.catch` with the promise we made above:

```
promise.then((message) => {
    console.log(`The promise finally ${message}`)
}).catch((message) => {
    console.log(`The promise finally ${message}`)
})

// prints:
// The promise finally resolved!
// or
// the promise finally rejected!
```

Why are Promises useful?

Promises are the cleanest (but not the only) way to handle the common scenario where we need to make requests to a server, which is typically done via an HTTP request. In fact, the fetch() function we were using earlier in the course returns a promise!

I/O, or "input/output"

Almost every time you use a promise in JavaScript it will be to handle some form of I/O. I/O, or input/output, refers to when our code needs to interact with systems outside of the (relatively) simple world of local variables and functions.

Common examples of I/O include:

- HTTP requests

- Reading files from the hard drive
- Interacting with a Bluetooth device
- Promises help us perform I/O without forcing our entire program to freeze up while we wait for a response.

Promises and the "await" keyword

The `await` keyword is used to wait for a `promise` to resolve. Once it has been resolved, the [await] expression returns the value of the resolved `promise`.

Example with .then callback:

```
promise.then((message) => {
  console.log(`Resolved with ${message}`)
})
```

Example of awaiting a promise:

```
const message = await promise
console.log(`Resolved with ${message}`)
```

The async keyword

While the `await` keyword can be used in place of `.then()` to resolve a promise, the async keyword can be used in place of new Promise() to create a new promise.

When a function is prefixed with the `async` keyword, it will automatically return a Promise. That promise resolves with the value that your code returns from the function. You can think of `async` as "wrapping" your function within a promise.

New Promise

```
function getPromiseForUserData(){
  return new Promise((resolve) => {
    fetchDataFromServerAsync().then(function(user){
      resolve(user)
    })
  })
}

const promise = getPromiseForUserData()
```

Async

```
```

```
async function getPromiseForUserData(){
  const user = await fetchDataFromServer()
  return user
}

const promise = getPromiseForUserData()
```

Note!!!: `await` can only be used inside an `async` function or at the top level of a module.

.then() vs await

In the early days of web browsers, promises and the `await` keyword didn't exist, so the only way to do something asynchronously was to use callbacks.

A "callback function" is a function that you hand to another function. That function then calls your callback later on. The setTimeout function we've used in the past is a good example.

```
function callbackFunction(){
  console.log("calling back now!")
}
const milliseconds = 1000
setTimeout(callbackFunction, milliseconds)
```

The `.then()` syntax is generally easier to use than callbacks without the Promise API, but `async` and `await` make handling promises even simpler. As a general rule, prefer `async` and `await` over `.then` and New Promise() for more readable and maintainable code.

To demonstrate, which of these is easier to understand?

```
fetchUser.then(function(user){
  return fetchProjectForUser(user)
}).then(function(project){
  return fetchBoardForProject(project)
}).then(function(board){
  console.log(`The board is ${board}`)
});
```

```
const user = await fetchUser()
const project = await fetchProjectForUser(user)
const board = await fetchBoardForProject(project)
console.log(`The board is ${board}`)
```

They both do the same thing, but the second example is so much easier to understand! The `async` and `await` keywords weren't released until after the `.then` API, which is why there is still a lot of legacy `.then()` code

out there.