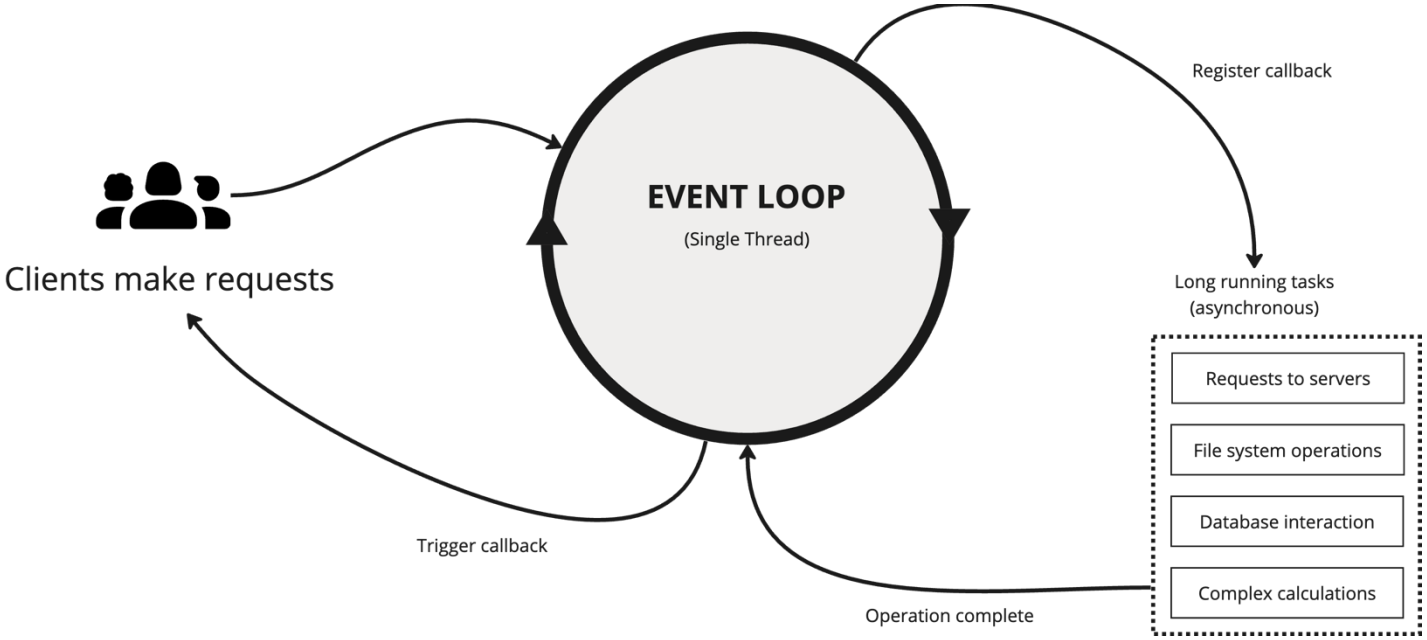
Week 1 Client side

Asynchronous Javascript

SAXION HOGESCHOOL

EVENT LOOP

Javascript has a single threaded event loop



This means that, in that loop,at any given moment only one thing can be processed.

Doing things that require time or cannot be garuanteed to complete in a given time (fetch calls, file system operation, etc.) Will have to be “off loaded” to background processes.

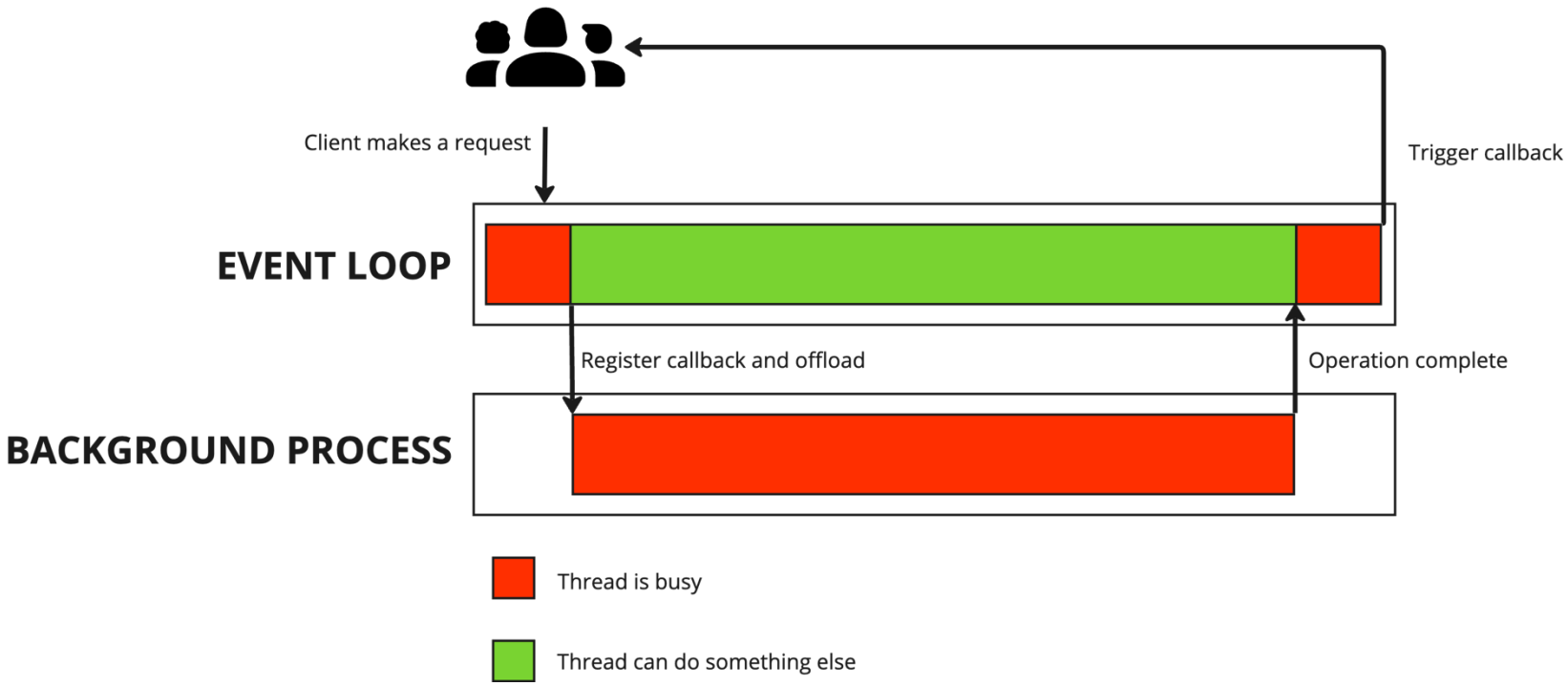
Why?

All client requests go to the event loop.

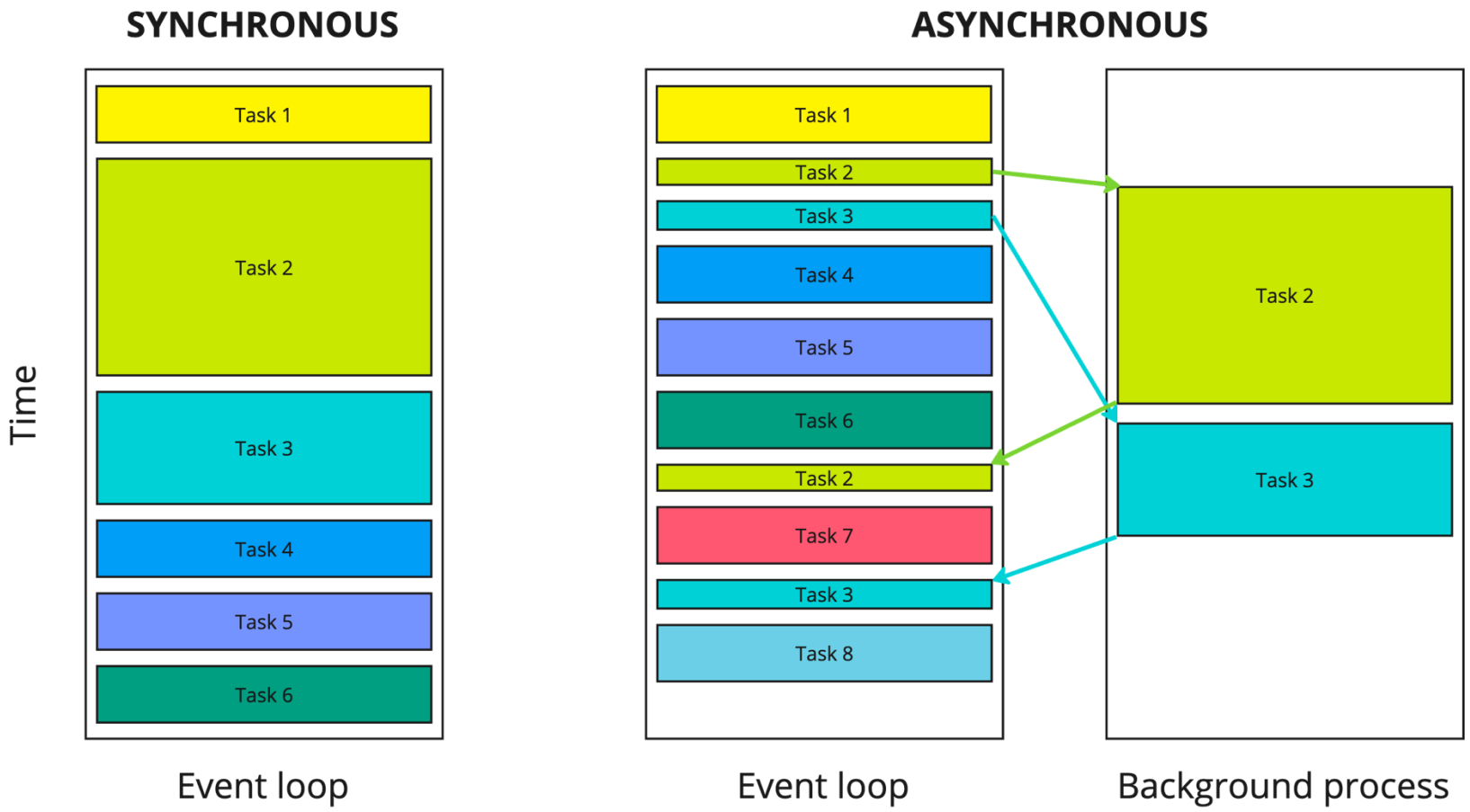
If tasks cannot be off loaded the event loop would be **blocked**.

To a client **blocked** means **unusable**.

Make sure to do as little as possible in the event loop!



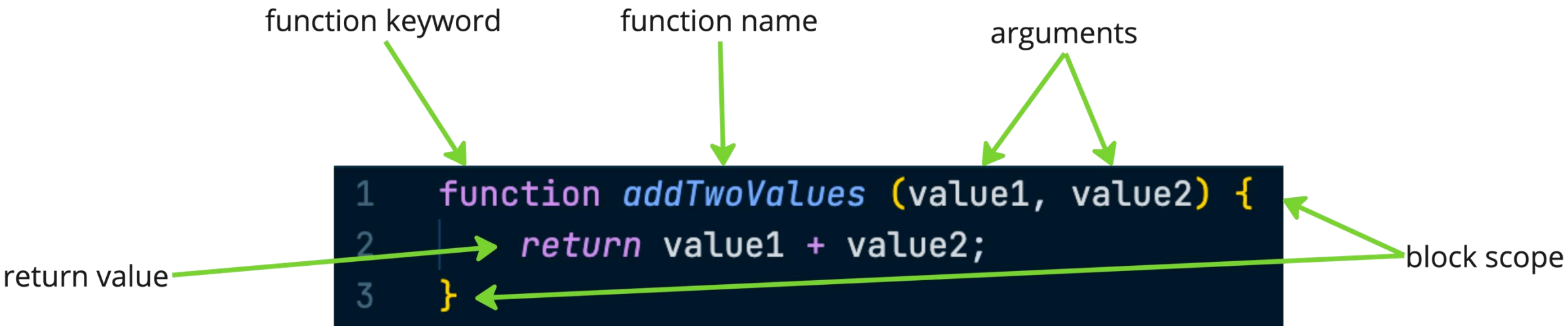
Off loading tasks is the asynchronous part.



Javascript offers a few programming techniques to accomplisch asynchronous programming

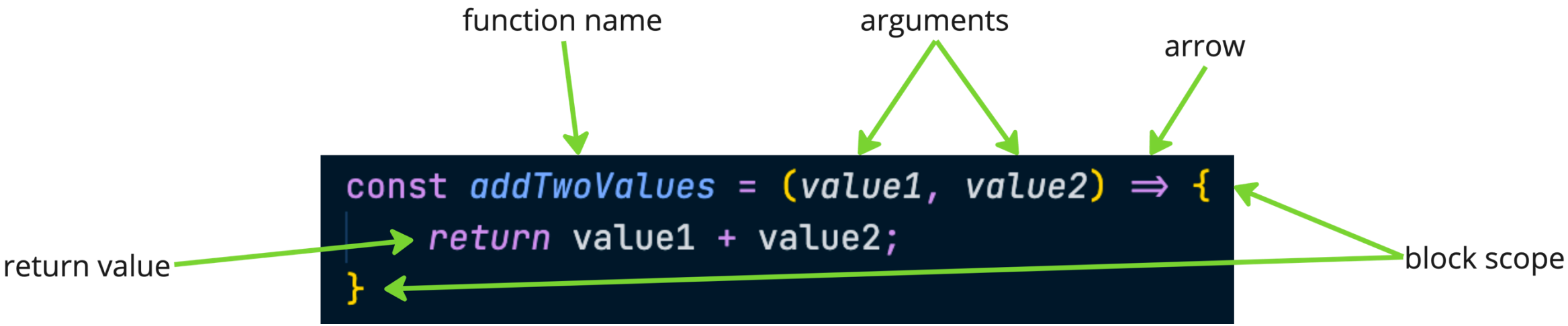
ARROW FUNCTIONS

So far we used classic functions in class:



Today we go a little deeper and take a look at arrow functions

Arrow functions are also known as Lambda functions



These are basically the same. Some differences:

- There is no `function` keyword
- The function is assigned to a variable or constant
- The function is then called using the variable name

```
addTwoValues(2,2);
```

But there is more…

Depending on what you want to do, the syntax may vary!

Consider the following code

```
const addOne = (value) ⟹ {
    return value + 1;
}
```

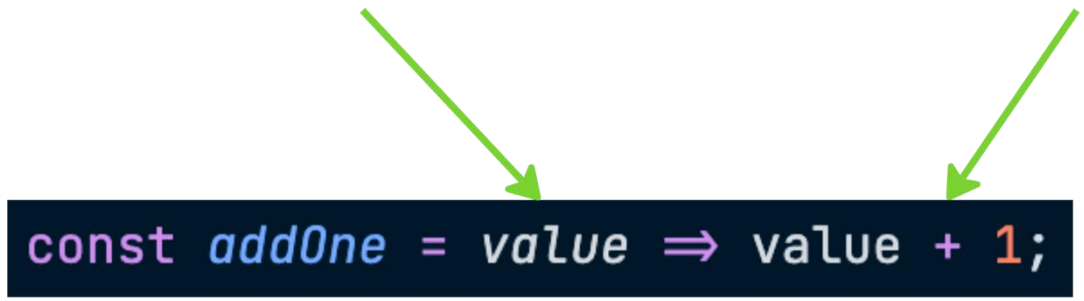
The function:

- Accepts a single parameter
- Contains a single line body

For some cases arrow functions allow shorter notation

If the function only takes a single parameter the brackets can be omitted

If the function contains a single expression the value of that expression can be returned automatically



```
const addOne = value ⟹ value + 1;
```

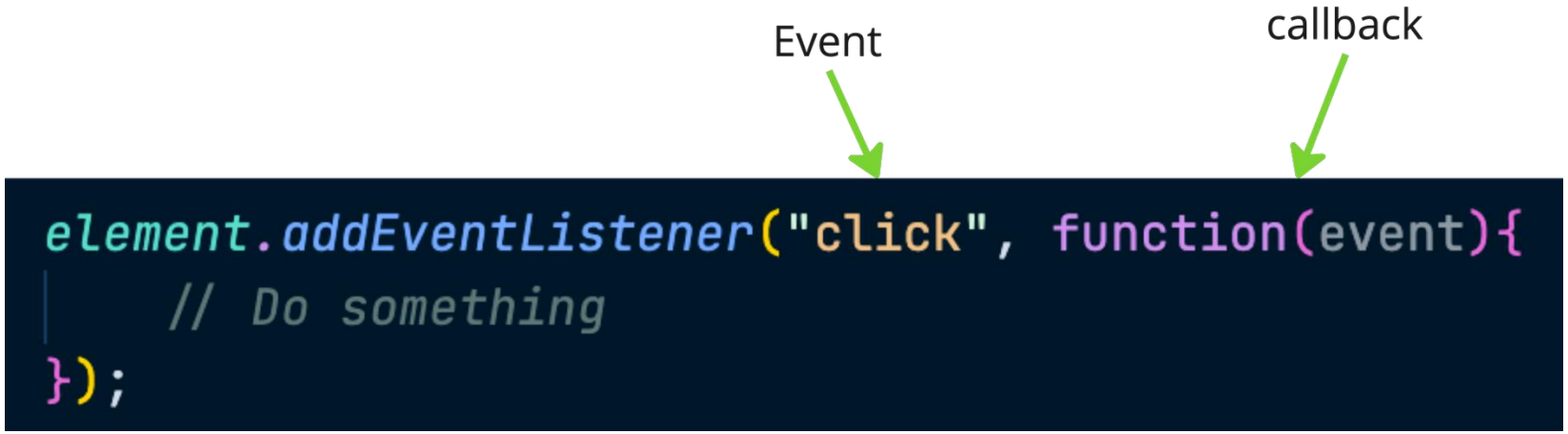
This works the same as the function on the previous slide

There is no need for brackets or the return keyword!

This allows for very small functions… Why?

Remember callbacks?

Call backs are a major pattern in Javascript. You will find them pretty much everywhere. Most notably while handling events in your Web Basics project last year.



In many cases arrow functions help keep your code clean and readable

There are more examples in the slides to come

Option 1:

CALLBACKS

SAXION HOGESCHOOL

Callbacks are functions that are passed as an argument to another function.

This function can then call the passed in function when it wants to. This is the callback.

```
function doSomethingAsync(callback) {
  // Do something asynchronous here.
  // When the asynchronous operation is complete, call the callback function.
  callback();
}

function onSomethingDone() {
  // This code will be executed when the asynchronous operation is complete.
  console.log("The asynchronous operation is complete!");
}

doSomethingAsync(onSomethingDone);
```

Callbacks with arrow functions were discussed in the server class of week 1

Callback problems: callback hell

- Callbacks are a very powerful mechanism, but by using them everywhere you might end up with several problems
- Especially if you have multiple operations that all rely on callbacks, you might end up in the “callback hell”

```
1  // Callback Hell
2
3
4  a(function (resultsFromA) {
5      b(resultsFromA, function (resultsFromB) {
6          c(resultsFromB, function (resultsFromC) {
7              d(resultsFromC, function (resultsFromD) {
8                  e(resultsFromD, function (resultsFromE) {
9                      f(resultsFromE, function (resultsFromF) {
10                         console.log(resultsFromF);
11                     })
12                 })
13             })
14         })
15     })
16 });
17
```

Callback problems: error handling

- Since results are returned by callback, there is no error mechanism available such as throwing exceptions.
- Therefore, the convention has been for a very long time “Error-First Callback functions”.
- This means that the first parameter of a callback is an error object and the rest of the parameters are the data.
  - If the first parameter is set, you know that the original method threw an error.
- Since not all libraries use this convention, it can be rather annoying having to deal with errors in callback functions.
- It is not enforced by the language…

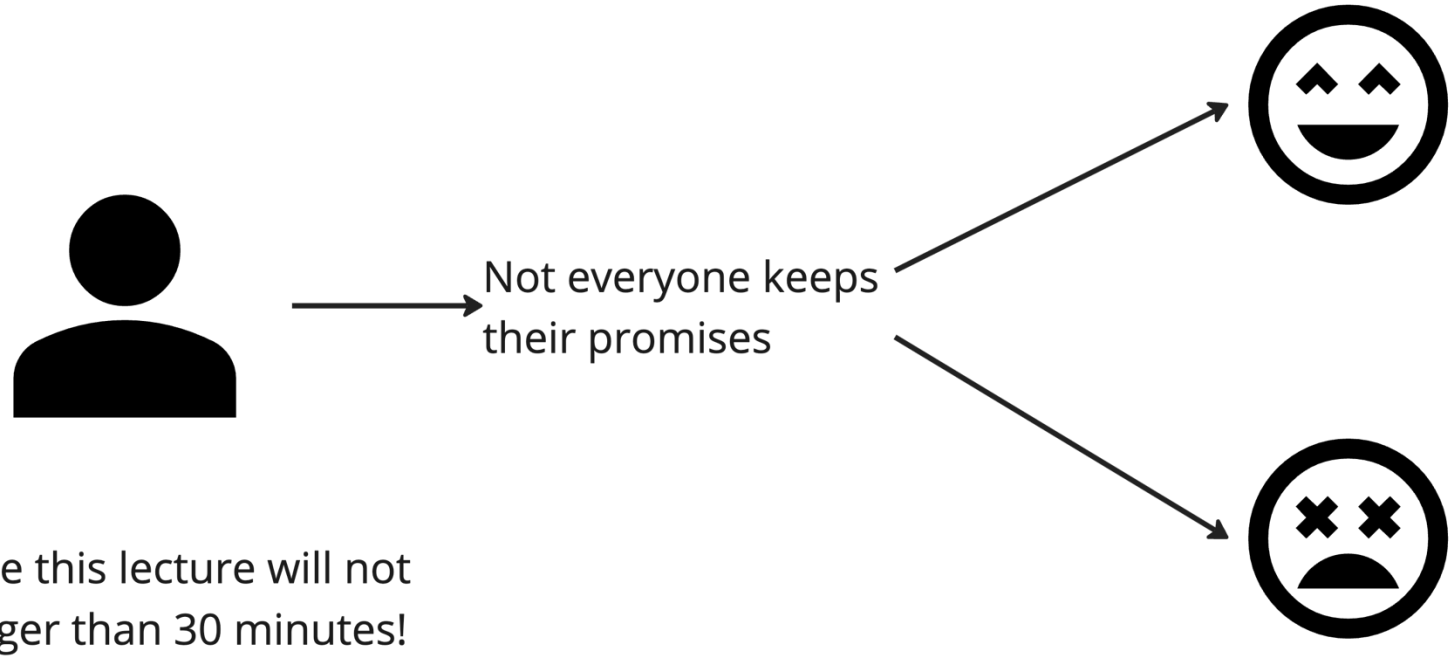
```
fs.readFile('./path/to/file.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Option 2:

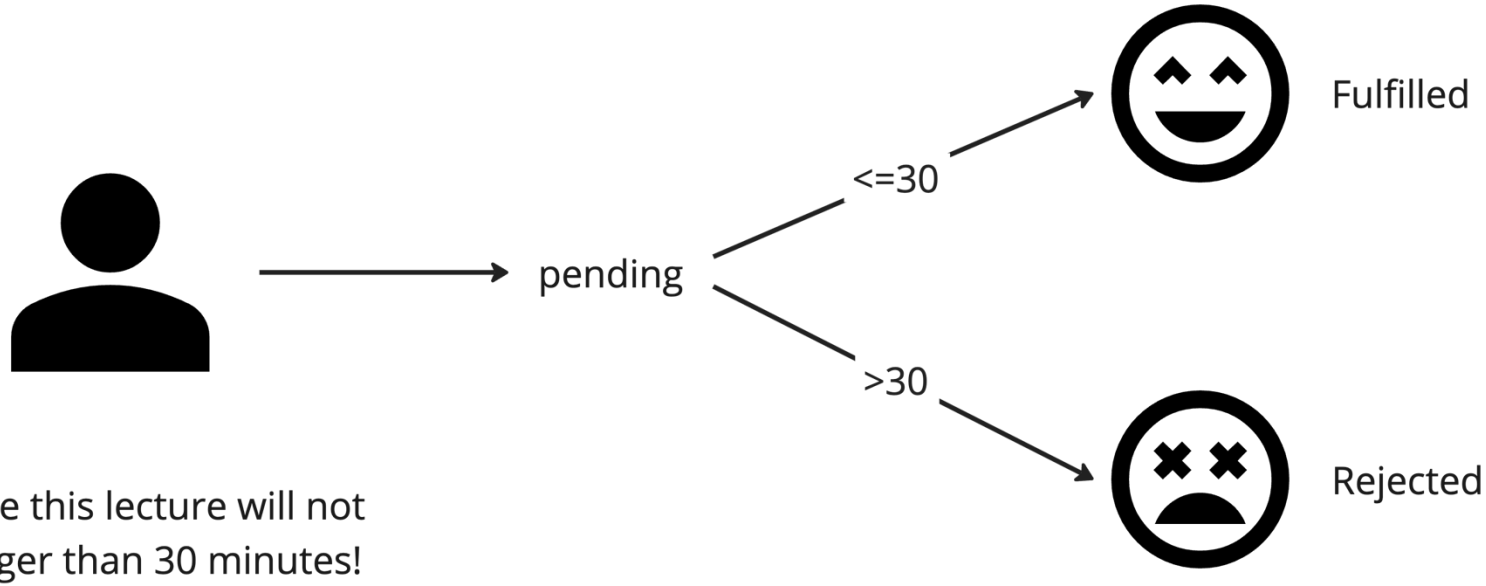
PROMISES

SAXION HOGESCHOOL

Promises in Javascript are just like promises in real life.



Javascript is the same way.



I promise this lecture will not take longer than 30 minutes!

A promise can be in one of three states:

- **Pending**: The initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation was completed successfully.
- **Rejected**: The operation failed.

So how does this work in code

Once you have created a promise, you can use it to handle the eventual success or failure of the asynchronous operation.

You can do this by chaining `then()` methods to the promise.

Handling the promise

Success (fulfilled)

Failure (rejected)

```
promise
    .then((value) => {
        // Do something when the promise is fulfilled.
        console.log("The operation was successful.");
    })
    .catch((error) => {
        // Do something when the promise is rejected.
        console.log("The operation failed.");
    });
```

Even though we are using promises callbacks are used.

Do promises replace callbacks?

- Promises were introduced to make asynchronous operations in JavaScript easier.
- **Please note:** promises are not a replacement for callbacks (`resolve` and `reject` are still callback functions).

- The advantage of using promises are:
  - We can avoid the callback hell by chaining `then()` operations
  - We have a good way of dealing with errors (`resolve` -> `then()` and `reject` -> `catch()`)
  - We can schedule multiple promises and wait for them to all be finished (`promise.all()`)

Option 3:

ASYNC/AWAIT

Async/await

Another (better readable) way to write asynchronous code

Both promises and async/await are ways of handling asynchronous code.

Promises with then() and catch() are the more traditional (older) way of handling asynchronous code.

Async/await is a newer syntax
Meant to simplify writing asynchronous code.

Async/await

Declare a function to be async

```
async function doSomethingAsync() {
  // Do something asynchronous here.
  const result = await makeANetworkRequest();
  return result;
}

try {
  const result = await doSomethingAsync();
  console.log(result);
} catch (error) {
  console.error(error);
}
```

You can use standard try/catch

Await the result of a function call

Does async/await replace Promises?

- NO! Actually, every async function returns a Promise!
- Async/await was introduced to further simplify the way developers work with asynchronous code.
- It has been built on top of Promises

- The idea is that you can write asynchronous code as if it was synchronous code, by just using normal constructions like try...catch.

- Every function that returns a Promise can be called by using await.

```
moveTo(50, 50)
  .then(() => moveTo(20, 100))
  .then(() => moveTo(100, 200))
  .then(() => moveTo(2, 10))

// becomes

async function animate() {
  await moveTo(50, 50);
  await moveTo(20, 100);
  await moveTo(100, 200);
  await moveTo(2, 10);
}
```

Questions?