Asynchronicity And Networking

Callbacks and Events

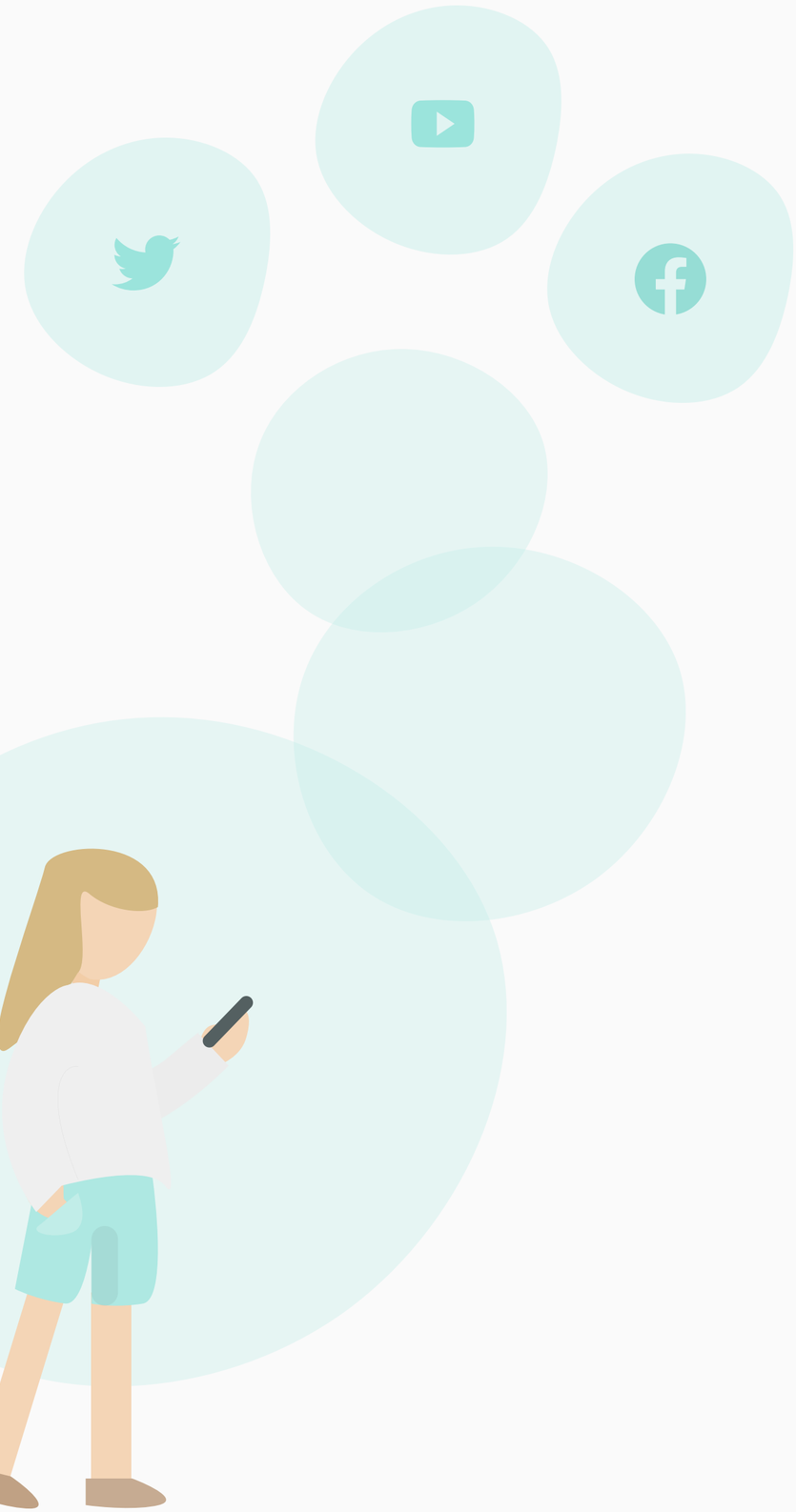
Created by Zain Afzal @zainafzal08 @blockzain

Overview

Client server Interactions

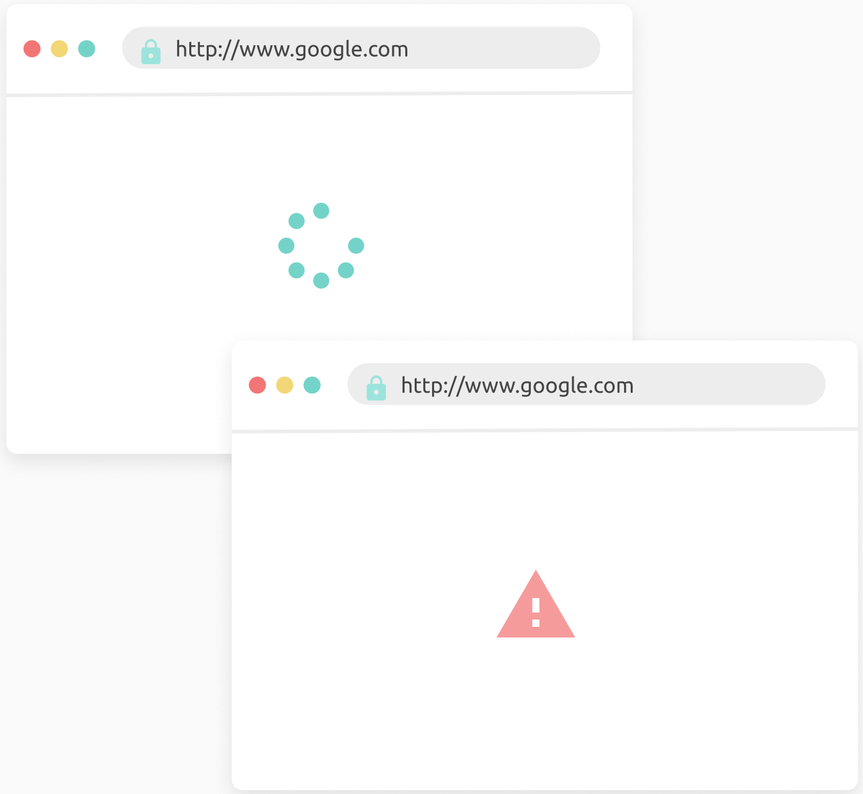
AJAX

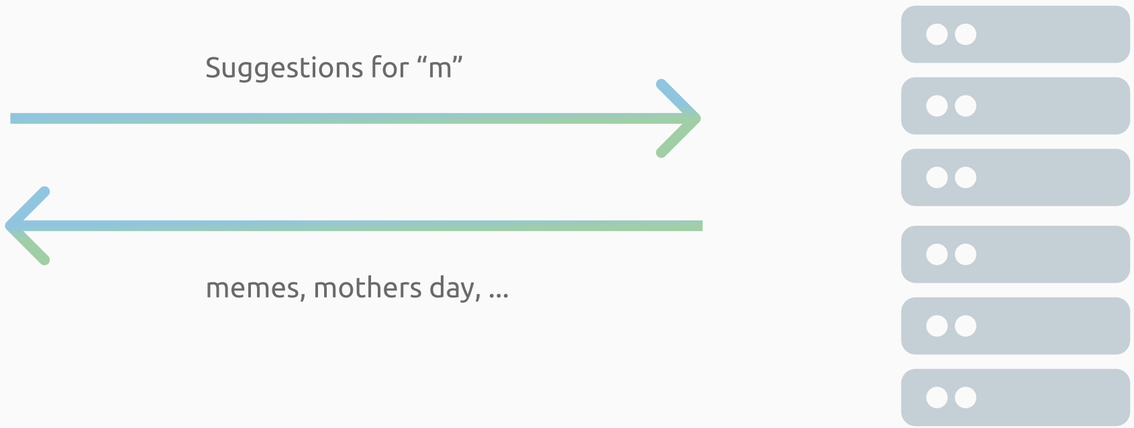
- Event Loop
- Callbacks and XMLHTTP
- Promises and Fetch
- Async Await



Event Loop

Slide 1 of 6





Recap

Ajax is a system via which we can load a page once and use background requests to provide interactivity and data exchanges. By decoupling our pages from the classical http request and response loop we gain a lot of control over how our sites behaves. This lecture we explore how we can achieve this in javascript.

What does asyncronous mean?

When programming, doing a set of instructions synchronously means that every instruction must complete before the next action can start. Asynchronous execution however means our code can run out of order and we don’t have to wait for one action to finish before we start the next.

Concurrency Models

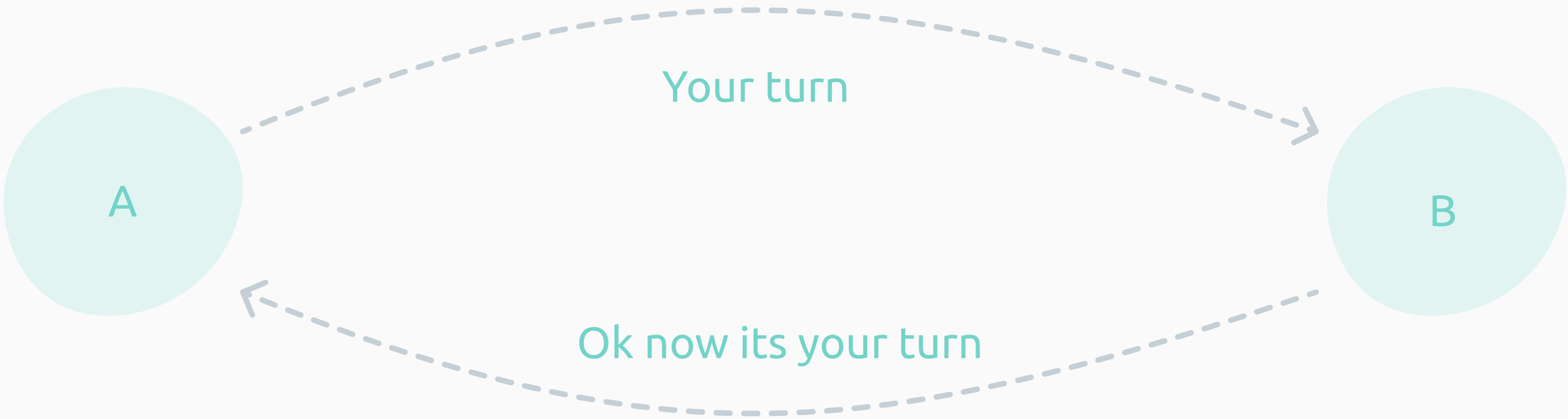
Doing multiple things at the same time with 1 thread (as is the case with js) requires a concurrency model, that is how do we distribute our work and reason about our code. There are many models we can use for example:

Threads / Preemptive Multitasking

Here you write a bunch of code and spin up threads, each threads runs the code assuming that it's the only thing running. At any random time the thread can be stopped, it's state saved and the cpu tasked with some other bit of work.
We say that a thread can be preempted, here we need mutex's and semaphores to guard critical sections & variables to make sure we don't get interrupted if we are doing something fiddily. This is the typical multi-tasking model found in many languages like Java and Python.

Do OS if you want to learn more, but this is not a model useful to us as all transformations to the dom must happen **in a predictable order** to avoid broken pages.

Event Loop



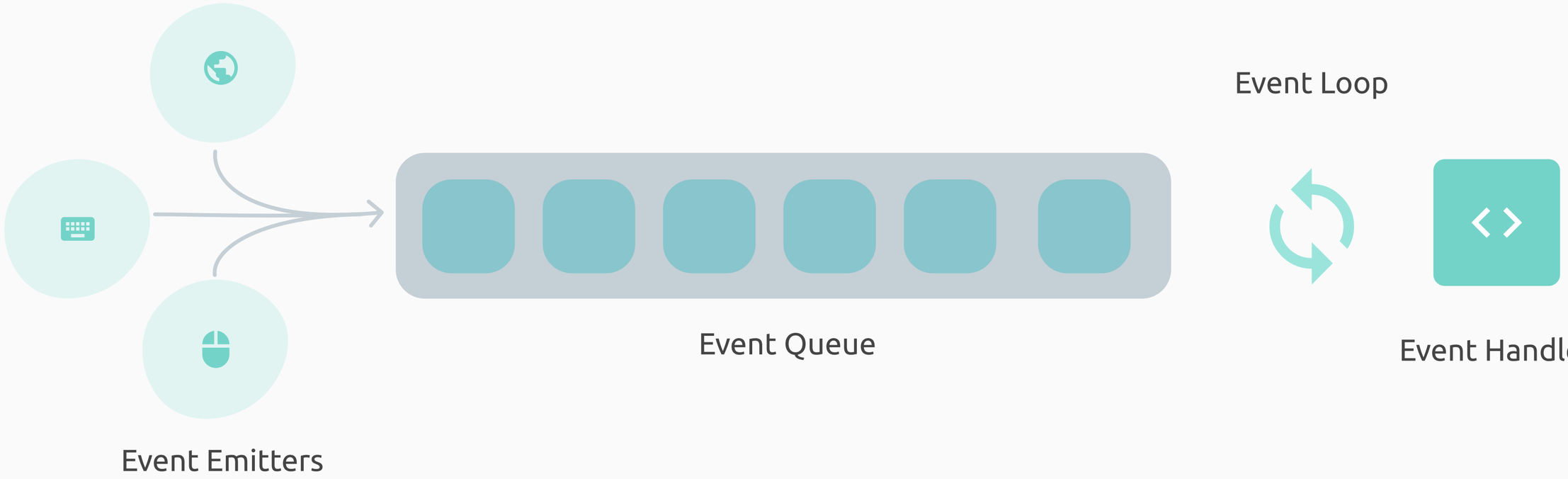
Coroutines / Cooperative Multitasking

Here we have co-routines, these are separated flows of execution which run in tandem. co-routines can cooperatively yield to other coroutines.

yield saves the state of co-routine A, and resumes B's state from it's previous yield point. No preemption between yields, so no need for mutex's or semaphores to guard critical sections. This is better since we know when we are about to be interrupted but is often designed with the idea of having predefined processes that share the CPU and isn’t as suited to the dynamic way events are generated on the web. This is however useful for general async programming in JS and is available via **async/await** but we talk about that later on in the course.

Event Loop

Slide 5 of 6



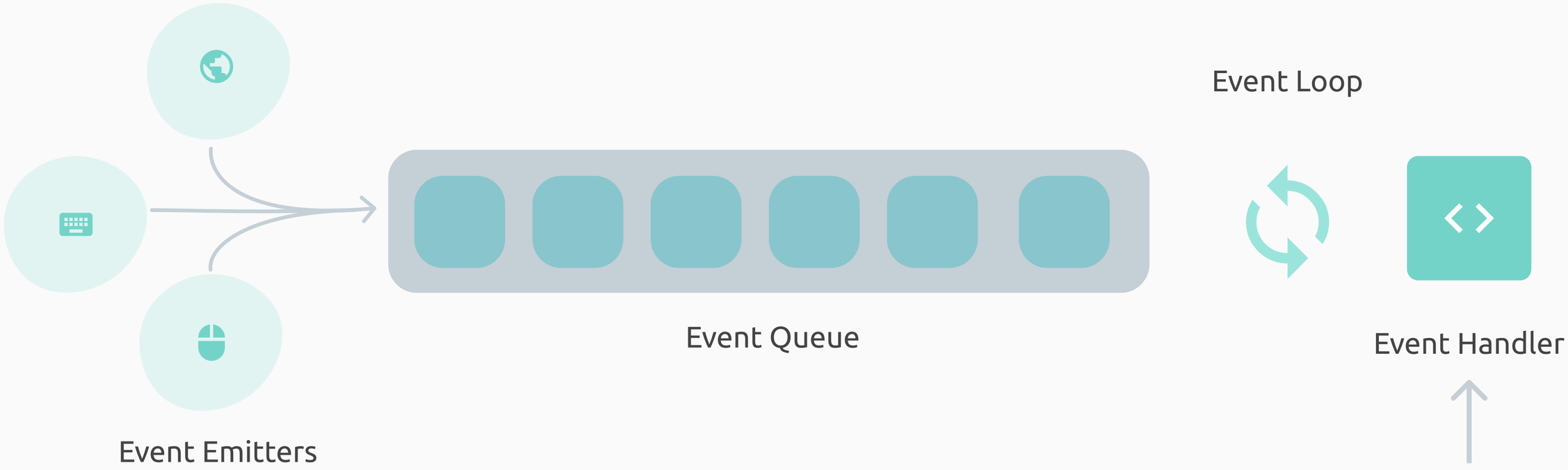
Event Loop (What js uses)

This is really just cooperative multitasking with additional logic to make the process of “yielding” the current execution enviornment transparent and seemless. Here “events” are generated by various sources and added to an event queue. Then when the thread is free the event at the front of the queue is popped off and passed to its respective handler. The event loop then **waits** for the handlers to finish before handling the next event. In this way the javascript we write in a handler runs start to finish in a predictable way (no need for locks) but we can still have many events at the same time.

```
window.addEventListener('click', () => {
    shoot();
});

window.addEventListener('keydown', () => {
    move();
});

// While we wait for those to happen we can
still do other stufff!
```

Is javascript asyncronous?

Yes and no. Javascript code runs syncronously, one line after the other, but the language gives you the ability to schedule code to run later, allowing you to do things while you wait, i.e allows you to run code asyncronously. The first and most common way to do that is something you've already seen, the humble **Callback**



What is a callback?

A function that is called when the pending work has completed. Can use either a named function or an anonymous lamdba. In essense a callback is just an event handler which responds to some event.

```
1 fs.readFile('foo.txt', (result, err) => {
2  // Do something with result here
3 })
4
5 function onComplete(result, err) {
6  // Do something with result here
7 }
8
9 fs.readFile('foo.txt', onComplete)
```

```
1 // What will the following code print out?
2 window.setTimeout(() => console.log('super'), 0);
3 console.log('hello');
4 window.setTimeout(() => console.log('world'), 10);
5 window.setTimeout(() => console.log('cool'), 0);
6
```

🔧 Event Loop Demo

Check out main-content/lectures/async-timer-demo
and then think about what will the above snippet print out?

Callbacks

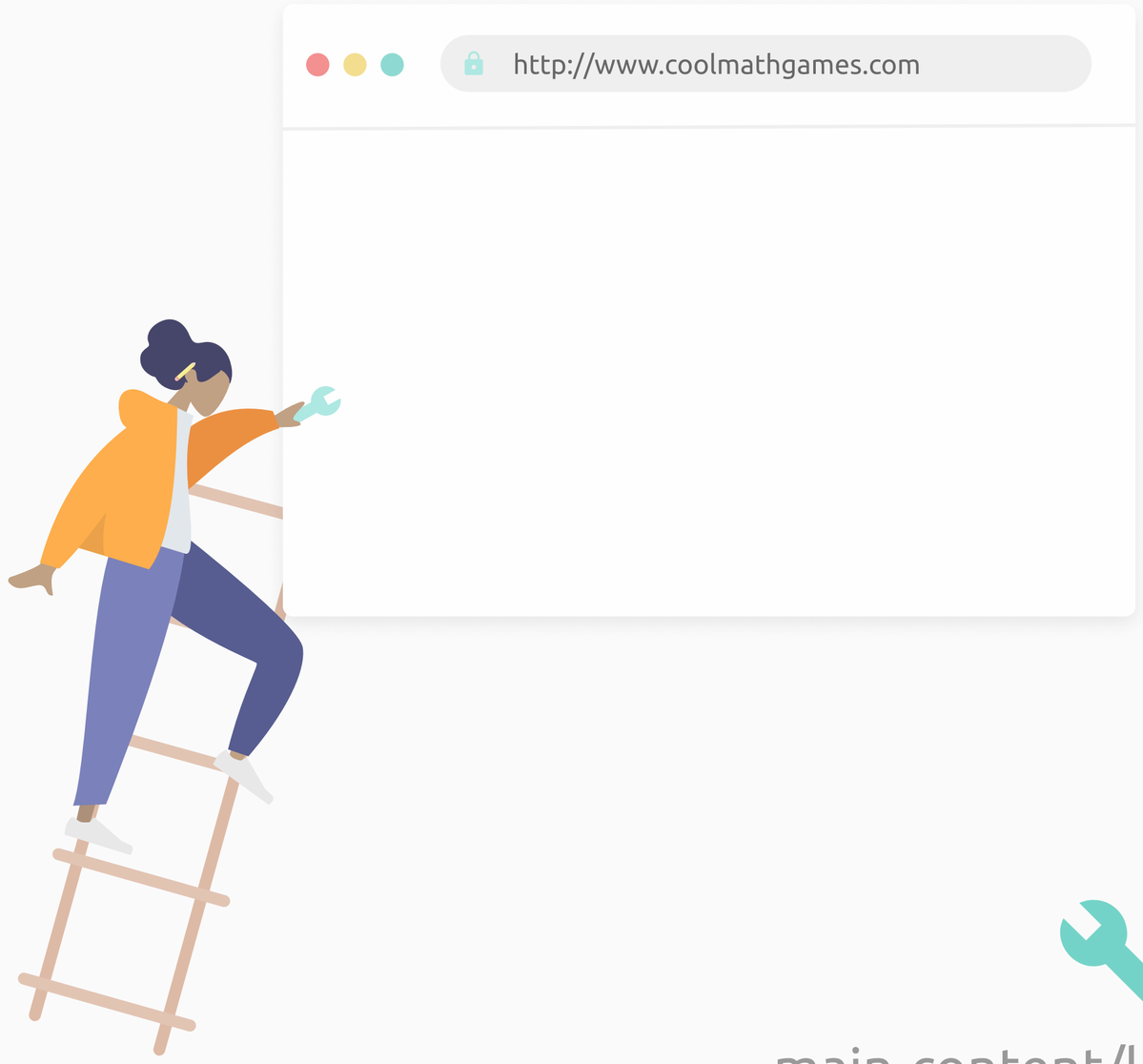
```
// What will the following code print out?
window.setTimeout(() => console.log('super'), 0);
console.log('hello');
window.setTimeout(() => console.log('world'), 10);
window.setTimeout(() => console.log('cool'), 0);

```

```
> hello
> super
> cool
> world
```

🔧 Event Loop Demo

Callbacks

Slide 4 of 13



UI Demo

main-content/lectures/async-ui-demo

Js has one thread

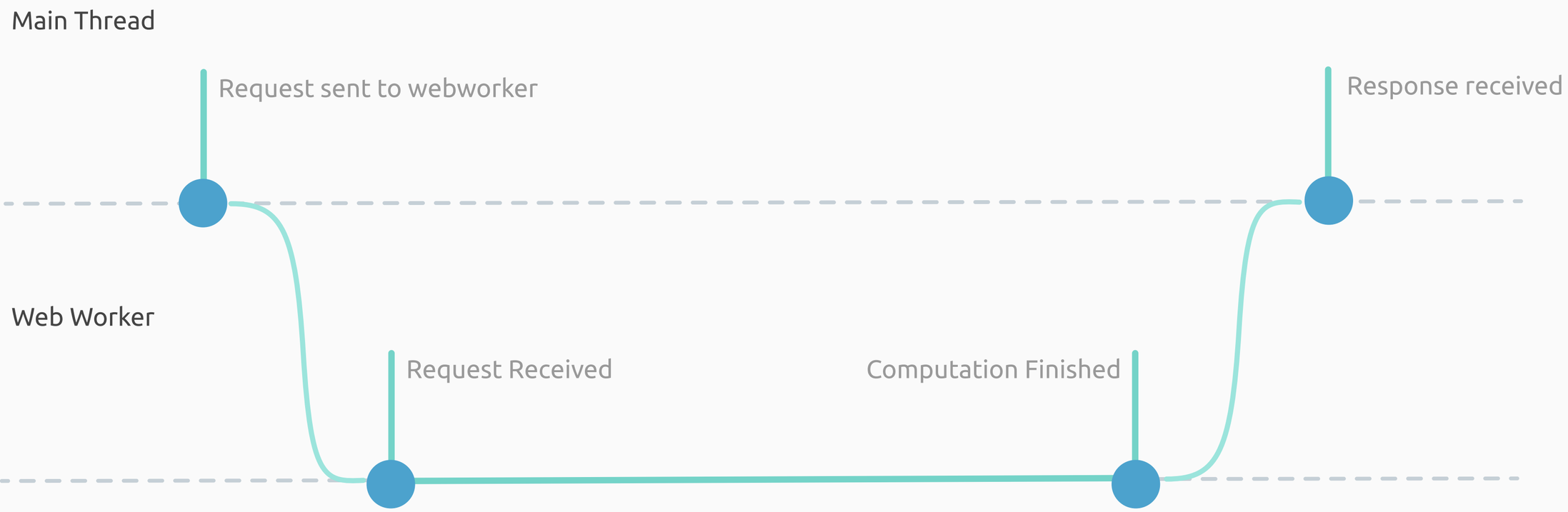
All our javascript and much of the code that helps parse and render the webpage runs in a single “main” thread and since we use a cooperative multitasking model where nobody else can use the thread until our handler finishes, we have the ability to block the thread and essentially freeze the tab.

IO Bound vs CPU Bound operations

Did you notice in the last demo fetching a large file didn't seem to block the UI at all? This is because even though fetching a large file and computing a large factorial both take a lot of time, the reason they take a lot of time is different. With IO bound operations the thing we are waiting for is for something to respond, a server, a hard drive etc. With CPU bound operations we are waiting for the cpu to finish processing some calculation. The key difference is with IO bound operations, the thread is **idle** while waiting for a response.

How to avoid network blocking

Simply shoot of a request and then register a callback to handle the response when it comes back at some point in the future. The key being not to wait for the response to come back. It’s actually quite hard to set up a blocking network request these days, by default almost all network interaction apis default to non blocking behavior, what’s crucial here is understanding why we can’t simply wait for the response, even if we have nothing to do until it comes.



How to avoid cpu blocking

This is a bit more complicated, a cpu operation needs active control of the main thread so it can do it’s work and while it’s using the cpu nothing else can. Here the best approach is to move to a different OS level thread which has the ability to use premptive multitasking or multiple cores on the CPU to manage the task without freezing (*or make the server do it*). The way we use OS threads in web development is via a **Web Worker.** We will try to cover this in more detail later in the course but the main gotcha is there is non trivial work involved with getting data to/from the web worker

```
const xhr = new XMLHttpRequest();
// Set up our request.
xhr.open("GET", "https://api.ipify.org?format=json");

xhr.onload = function (e) {
   // Make sure we got a 200 "all ok"
   if (xhr.status === 200) {
     console.log(xhr.responseText);
   }
};
// Actually fire the request off.
xhr.send();
// While we wait for onload to get called we can show a loading spinner!

```

XMLHTTP

This is a object that lets us interact with servers from javascript by creating a request, registering handlers for what to do when we get a response, and shooting off the request. This is designed to give you more control over the process so gives you lots of toggles so you can stream files, get updates on the progress of a large download etc.

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ipify.org?format=json");

xhr.onload = function (e) {
   if (xhr.status !== 200) {
     // This is a implict error, something went wrong with the server or
     // our requst was malformed.
     alert('Something went wrong!');
   }
};

// Something went wrong after our send logic.
xhr.onerror = handleError(e);

try {
  xhr.send();
} catch(e) {
  // Something went wrong with us sending out our request.
  handleError(e);
}

```

XMLHTTP Error Handling

The land of network interactions is full of errors, your users WILL run into connectivity issues or authentication issues or whatever, you need to handle these cases. Be careful to handle both explit errors such as the network failing and implict errors such as the status code being non-200. Some errors will throw before the request is sent and result in a error being thrown from xhr.send() or similar, others will only be registered once a response is received, these will trigger the xhr.onerror function if present.

Challange

Using https://jsonplaceholder.typicode.com create a website which given a username will show the title of all the posts made by that user (just title) and the number of comments on each post.

🔧 XMLHttp Demo

main-content/lectures/async-XMLHttp-demo

Callback hell sucks because

- Makes it difficult to reason about code
- Makes it harder to debug our code
- Makes it harder to extend / update our code
- Makes our code look ugly
- Makes our code fragile, a single missed callback in that mess causes a failure
- Makes it harder to catch and resolve errors

```
getData(function(x){
    getMoreData(x, function(y){
        getMoreData(y, function(z){
            ...
        });
    });
});

```

Callback Hell

You can see when you have to make multiple requests where each request relies on the previous that you get into a confusing mess of nested callbacks.

Callbacks

Slide 13 of 13

Avoid Callback hell by

- ✓ Defining functions and referencing them
- ✓ Keeping code shallow
- ✓ Avoid overuse of callbacks for things that can be syncronous
- ✓✓ Use Promises (next lecture)

```
get('api/allusers', (allUsers) => {
  allUsers.map(user => {
    get(`api/user/${user.id}/posts`, (posts) => {
      posts.map(post => {
        get(`api/post/${post.id}/comments`, (cmnts) => {
          // ...
        });
      });
    });
  });
});
```

```
function processUsers(allUsers) {
  for (user of allUsers) {
    get(`api/user/${user.id}/posts`, processPosts);
  }
}
function processPosts(posts) {
  for (post of posts) {
    get(`api/post/${post.id}/comments`, processCmnts);
  }
}
function processCmnts(comments) {
  for (comment of comments) {
   // ...
  }
}

get('api/allusers', processUsers);
```