Asynchronicity is the backbone of modern web development in JavaScript

JavaScript is single threaded (one command executing at a time) and has a synchronous execution model (each line is executed in order the code appears)

So what if we need to **wait some time before we can execute certain bits of code**? Perhaps we need to wait on fresh data from an API/server request or for a timer to complete and then execute our code

We have a conundrum - a tension between wanting to **delay some code execution** but **not wanting to block the thread** from any further code running while we wait

Solution 1

```
function display(data){
    console.log(data)
}

const dataFromAPI = fetchAndWait('https://twitter.com/will/tweets/1')

//... user can do NOTHING here 😭
//... could be 300ms, could be half a second
// they're just clicking and getting nothing

display(dataFromAPI)

console.log(“Me later!”);
```

Problems

— Fundamentally untenable - blocks our single javascript thread from running any further code while the task completes

Benefits

— It’s easy to reason about

Goals

1. Be able to do tasks that take a long time to complete e.g. getting data from the server
2. Continue running our JavaScript code line by line without one long task blocking further JavaScript executing
3. When our slow task completes, we should be able to run functionality knowing that task is done and data is ready!

Conundrum 🤔

Solution 2 - Introducing Web Browser APIs/Node background threads

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,1000);

console.log(“Me first!”);
```

We're interacting with a world outside of JavaScript now - so we need rules

```
function printHello(){
    console.log("Hello");
}

function blockFor1Sec(){
    //blocks in the JavaScript thread for 1 second
}

setTimeout(printHello,0);

blockFor1Sec()

console.log("Me first!");
```

Problems

— No problems!
— Our response data is only available in the callback function - Callback hell
— Maybe it feels a little odd to think of passing a function *into* another function only for it to run much later

Benefits

— Super explicit once you understand how it works under-the-hood

Introducing the readability enhancer - Promises

- Special objects built into JavaScript that get returned immediately when we make a call to a web browser API/feature (e.g. `fetch`) that's set up to return promises (not all are)
- Promises act as a placeholder for the data we hope to get back from the web browser feature's background work
- We also attach the functionality we want to defer running until that background work is done (using the built in .then method)
- Promise objects will automatically trigger that functionality to run
  - The value returned from the web browser feature's work (e.g. the returned data from the server using `fetch`) will be that function's input/argument

Solution 3 - Using two-pronged ‘facade’ functions that both initiate background web browser work *and* return a placeholder object (promise) immediately in JavaScript

```
function display(data){
    console.log(data)
}

const futureData = fetch('https://twitter.com/will/tweets/1')

futureData.then(display); // Attaches display functionality

console.log(“Me first!”);
```

But we need to know how our promise-deferred functionality gets back into JavaScript to be run

```
function display(data){console.log(data)}
function printHello(){console.log(“Hello”);}
function blockFor300ms(){/* blocks js thread for 300ms with long for loop */}

setTimeout(printHello, 0);

const futureData = fetch('https://twitter.com/will/tweets/1')
futureData.then(display)

blockFor300ms()

// Which will run first?

console.log(“Me first!”);
```

We need a way of queuing up all this deferred functionality

Problems

— 99% of developers have no idea how they're working under the hood
— Debugging becomes super-hard

Benefits

— Cleaner readable style with pseudo-synchronous style code
— Nice error handling process

We have rules for the execution of our asynchronously delayed code

1. Hold each promise-deferred functions in a microtask queue and each non-promise deferred function in a task queue (callback queue) when the API ‘completes’
2. Add the function to the Call stack (i.e. execute the function) ONLY when the call stack is totally empty (Have the Event Loop check this condition)
3. Prioritize tasks (callbacks) in the microtask queue over the regular task queue

Promises, Web APIs, the Callback & Microtask Queues and Event loop allow us to defer our actions until the ‘work’ (an API request, timer etc) is completed and continue running our code line by line in the meantime

---

Asynchronous JavaScript is the backbone of the modern web - letting us build fast ‘non-blocking’ applications