COMPSCI 326 - Web Programming
Asynchrony & the JavaScript Event Model

join on the Slack #q-and-a channel as well as Zoom
Remember, you can ask questions of your teammates on your group Slack!
please **turn on your webcam** if you can
mute at all times *when you aren't asking a question*
(https://docs.google.com/document/d/1_DawWP1QCnsotgMq7wH13rzi8UnVUPaVHkByouW_p8A/edit?usp=sharing)

Today: Asynchrony & the JavaScript Event Model

resources:

- Concurrency model and the event loop - JavaScript | MDN
- Window setTimeout() Method
- Using Promises - JavaScript | MDN
- async function expression - JavaScript | MDN
- Using Fetch - Web APIs | MDN
- Video: What the heck is the event loop anyway? | Philip Roberts | JSConf EU

Other stuff:

- Project: unique, "socially relevant" if possible

The team project encompasses the last third of the semester - approximately 4 weeks. You will be required to work in a team of 2-4 students to design and implement a web application using the three important components of web applications including HTML, CSS, and JavaScript. The design and implementation of the application is entirely up to you, but it must solve a real-world problem, be connected to work you have done in other areas (general education), connect to your discipline (e.g., CS, Informatics) and be relatively unique as compared to other existing applications (within reason). You are expected to *not* use any frameworks beyond Bootstrap; other libraries are allowed only with prior consent of the professor.

Other languages: Threads

- in some languages - not JavaScript! - you can have multiple *threads* running simultaneously
- program is running different parts of the code at the same time
    - shared global variables and "heap" (things created by **new**)
    - separate local variables
- operating system periodically *pre-empts* code running on one thread and runs code from another thread
    - each thread can also run on its own *processor core*

JavaScript: No threads!

- JavaScript is *single-threaded* (even though the browser itself *is* multi-threaded!)
    - browser has threads to load images, load and parse JavaScript, HTML, CSS...

- BUT the browser itself has *one thread for both JavaScript and the UI*
    - all JavaScript "runs to completion"
        - infinite loop in JavaScript = frozen web page!
        - you can explicitly "yield" using `setTimeout` (below) but there's a better way
    - in the browser, all events (clicks, etc.) are inserted into a *message queue*
        - you can also add stuff to the queue directly by calling `setTimeout`
            - `setTimeout(`*`someFckingFunction, someFckingDelayInMs`*`)`
            - Tryit Editor v3.6
    - **while JavaScript is running, no events are processed**

Concurrency model and the event loop - JavaScript | MDN

Implications of browser model

- can't spend too much time in any JavaScript call
    - blocks the browser
    - need to break up code into chunks (how?)
- can't "synchronously" run inherently long-running things like I/O (think: downloading a video)
    - **in the browser, all potentially long-running operations are "non-blocking" = *asynchronous***

Can't directly do this (synchronous = one thing after another): **THIS CODE IS A LIE**

```
let var1 = loadGiantFile('file1');  // pretend JavaScript call
let var2 = loadGiantFile('file2');
```

This would *potentially* just block the browser!

Instead, need to do something different!

Promises + Async/Await

This used to be a huge pain (manually breaking up and passing things around: **continuations, Promises**).

- Promise: function is evaluating "in the background" and eventually has a value - you can now **await** the result to get it -- see code example
    - More about promises here: Promise
    - example syntax showing promise chaining: foo().then(() => { bar().then(); });
    - This kind of chaining can get out of hand and is a bit clunky.
- Alternative: Now you can do this "easily" with **async** and **await**
    - looks *almost* like sequential code!

For example, let's take a long-running function like this and turn it into a function that relinquishes control back to the event loop in the middle (making the web app more responsive):

```
function someFckingSlowFunction(inp) {
  // does something for a really long time
  return v;
}
```

- First, mark it as **async**

```
async function someFckingSlowFunction(inp) {
  // does something for a really long time
  return v;
}
```

- break it into parts

```
async function someFckingSlowFunction(inp) {
  // do part 1 for half as much time
  const v1 = someFckingNotSoSlowFunction1(inp);
  // do part 2 for half as much time
  const v2 = someFckingNotSoSlowFunction2(v1);
  return v2;
}
```

- now add **await** (IMPORTANT: anything you await must *also* be **async**! You shouldn't directly call an **async** function - you always have to **await** it...)

```
async function someFckingSlowFunction(inp) {
```

```
  // do part 1 for half as much time
  const v1 = someFckingNotSoSlowFunction1(inp);
  // do part 2 for half as much time
  // but this time, we "go to sleep" until it returns, yielding
control back to the browser
  const v2 = await someFckingNotSoSlowFunction2(v1);
  return v2;
}
```

The resulting function only blocks the UI thread for half as long each time, doubling responsiveness!

Synchronously using Async

What if you need to call an **async** function and *synchronously* wait for its result? (I just said you can't directly call an async function…)

- You have to do this weirdness, called an IIFE (immediately invoked function expression: IIFE - MDN Web Docs Glossary: Definitions of Web-related terms, Async/Await in Node.js and the Async IIFE)

an IIFE looks like this:

`(  (v) => { someFunction(v); }  ) (100);` // define the function on one argument then call it
`===` `someFunction(100);`

So what we do is we make an **async** function and then we can **await** inside, and immediately invoke (call) the whole thing:

```
async function someFckingNotSoSlowFunction2(v) {
    return v + 1; // not really that slow, just an example
}

(async () => { const r = await someFckingNotSoSlowFunction2(12);
killThisManyMonkeys(r); })();
```

result? 13

fetch

let v = fetch("https://youtube.com/AwesomeVideo");
let v2 = …;
let …

fetch("https://youtube.com/AwesomeVideo").then(...).then(...);

let v = await fetch("https://youtube.com/AwesomeVideo");
let v2 = await fetch("https://youtube.com/SecondAwesomeVideo");
let v3 = await fetch("https://youtube.com/ThirdAwesomeVideo");

- Advanced feature: Promise.all() - allows you to run multiple Promises concurrently and wait for all of them instead of one at a time.
    - Promise.all() - JavaScript | MDN
    - This can make a big difference since you aren't serializing each fetch

async/await are actually *syntactic sugar* for the underlying code, which really runs Promises.

- You can only **await** a Promise -- and all **async** functions return **Promise**s.
- Remember: JavaScript runs "to completion", but secretly, **await** ends a function and then waits for the event to happen to then continue execution from the next line.