# Advanced Javascript

Thierry Sans

# Outline

- The Javascript Event Loop
- Dealing with asynchronism (promises and async/await)
- Web workers

# Javascript Execution Model

## a.k.a The Event Loop

# Trivia

```
3  // begin timeout
4  setTimeout(function(){
5      console.info('1. Timeout');
6  }, 5000);
7
8  // generating the array
9  let a = Array.from({length: 10000000}, () => Math.random());
10 console.info('2. Array created');
11
12 // sorting the array
13 a.sort();
14 console.info('3. Array sorted');
```

In what order 1, 2 and 3 are going to be printed?

# Synchronous and Asynchronous Function Calls

There are two types of function calls

- **Asynchronous** calls pushed to the event loop
- **Synchronous** calls pushed to the call stack

# Asynchronous function calls

- DOM events (browser)
- Ajax requests (browser)
- Timer (browser and NodeJs)
- any non-blocking I/O (NodeJs)
- but promises are not necessarily async by default

# The Javascript Event Loop

Heap

Stack

Synchronous API

```
Array.sort(f)
```

Asynchronous API

```
setTimeout(f,t)
```

# The Javascript Event Loop

Heap

Stack

Synchronous API

```
Array.sort(f)
```

```
setTimeout(f,t)
```

Asynchronous API

```
setTimeout(f,t)
```

# The Javascript Event Loop

Heap

Stack

Synchronous API

```
Array.sort(f)
```

Asynchronous API

```
setTimeout(f,t)
```

# The Javascript Event Loop

Heap

Stack

Synchronous API

```
Array.sort(f)
```

```
a=new Array()
```

Asynchronous API

```
setTimeout(f,t)
```

# The Javascript Event Loop

Heap

Stack

Synchronous API

Array.sort(f)

Array a

a=new Array()

Asynchronous API

setTimeout(f,t)

# The Javascript Event Loop

Heap

Array a

Stack

Synchronous API

```
Array.sort(f)
```

Asynchronous API

```
setTimeout(f,t)
```

# The Javascript Event Loop

Heap

Stack

Synchronous API

Array a

a.sort()

Array.sort(f)

Asynchronous API

setTimeout(f,t)

# The Javascript Event Loop

Heap

Array a

Stack

a.sort()

Synchronous API

Array.sort(f)

Asynchronous API

setTimeout(f,t)

f

# The Javascript Event Loop

Heap

Array a

Stack

Synchronous API

Array.sort(f)

Asynchronous API

setTimeout(f,t)

f

# The Javascript Event Loop

Heap

Array a

Stack

Synchronous API

Array.sort(f)

Asynchronous API

setTimeout(f,t)

f

# The Javascript Event Loop

Heap

Array a

Stack

f

Synchronous API

Array.sort(f)

Asynchronous API

setTimeout(f,t)

f

# The Javascript Event Loop

Heap

Array a

Stack

Synchronous API

Array.sort(f)

Asynchronous API

setTimeout(f,t)

f

# Multi-threaded vs Single-threaded

Multi-threading does necessarily means things are executed in parallel

➡ We only have one CPU !

So, why do we need multithreading ?

➡ Because programming languages have blocking I/O, and by default, programs wait for the I/O to be completed

But multithreading is expensive

- in terms of software design (synchronization)
- in terms of performances (context switch)

What is the alternative to multi-threading?

➡ Single-threaded with non-blocking I/O

# Can you run a single-threaded web server?

Good performance, as long as the requests handlers :

- do some asynchronous I/O
  (filesys, database, cache, network and so on)

- do NOT do any heavy but yet synchronous computations
  (complex math, intensive data processing and so on)

# Asynchronism

# Callback - the building block for asynchronism

```
fs.readFile(filepath, 'utf8', function (err, data) {
        if (err) console.log(err);
        else console.log(data);
});
```

# Defining a promise

```
const readFile = function(filepath){

    return new Promise(function(resolve, reject){

        fs.readFile(filepath, 'utf8', function (err, data) {
            if (err) return reject(err);
            return resolve(data);
        });

    });

}
```

# Calling a promise

```
readFile(filepath)
    .then(function(data){
        console.log(data);
    })
    .catch(function(err){
        console.log(data);
    });
```

# Calling a promise with async/await

```
async function run() {
    let data = await readFile(filepath);
    console.log(data);
};

run().catch(err => console.error(err));
```

# Web Workers

http://afshinm.github.io/50k/

# How about multi-threaded Javascript?

But, if needed, Javascript can be multi-threaded

- Node cluster (NodeJS only)
- Web Workers (Browser and NodeJS)

✓ Good for heavy but yet synchronous computations

✓ Takes advantages of multicore machine

# Web Workers for parallelism

- Create threads in Javascript (now frontend and backend)
- These threads can run in parallel (separate event loop)

# What a web worker can/cannot do
# **on the frontend**

- ✓ `XMLHttpRequest`
- ✓ `indexedDB`
- ✓ `location` (read only)

- ◉ `window`
- ◉ `document` (not thread safe)

# Create a web worker

doSomething.js

```
function(){
 "use strict";

 // receive message
 self.addEventListener('message', function(e){
   let data = e.data;
   // send the same data back
   self.postMessage(data);
 }, false);

});
```

# Instantiate a web worker

main.js

```
let worker = new Worker('doSomething.js');

// sending a message to the web worker
worker.postMessage({myList:[1, 2, 3, 4]});


// receive message from web worker
worker.addEventListener('message', function(e) {
     console.log(e.data);
}, false);
```