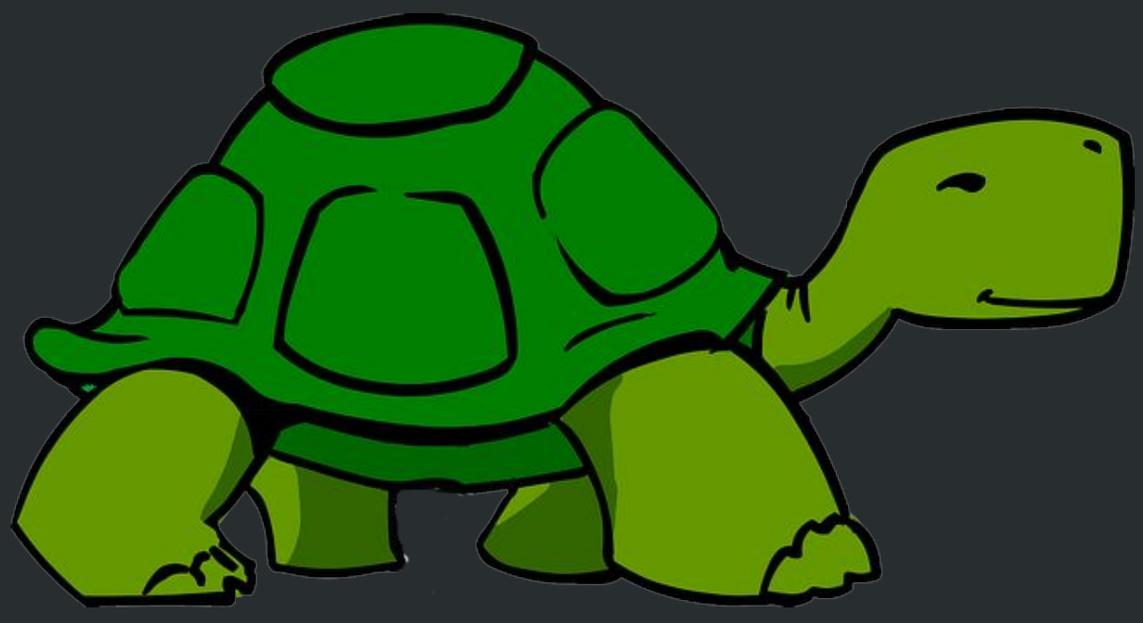
Asynchrony & Callbacks

Slow things

There are many things that are slow in the programming world.

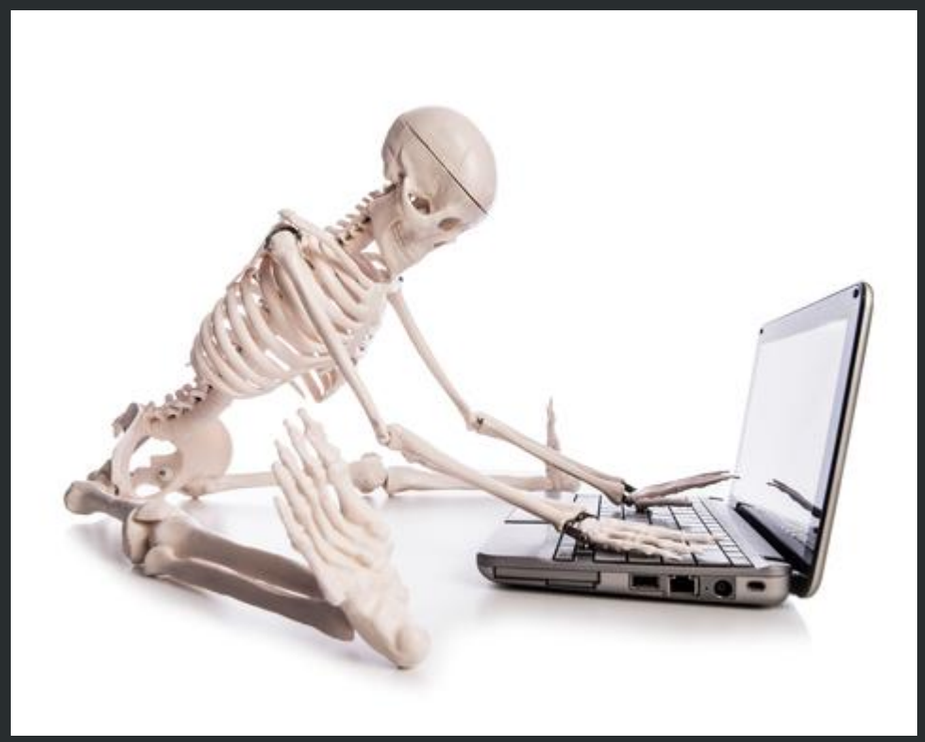


Slow things

We've already made reference to some of them.

Slow things

Database calls, file system reads, network requests... do you remember IMDB?



Slow things

Say we had a really slow computation.

Slow things

Like this one...

```
// Takes 25 seconds
var result = superSlowThing(42);
```

Slow things

We wouldn't want our program to have to wait around.

```
// Takes 25 seconds
var result = superSlowThing(42);
```

Slow things

Because that means that it can't do anything else.

```
// Takes 25 seconds
var result = superSlowThing(42);

// Takes 25 seconds to get here
var otherResult = fastThing();
```

Slow things

We want our programs to do slow things in the background.

Slow things

And continue working on other tasks in the meantime.

Asynchrony

In other words we want slow computations to be performed *asynchronously*.

Asynchrony

Asynchrony is a state of not being synchronized.

Asynchrony

Basically, it's when things happen outside the regular flow of time.

Asynchrony

Slow computations can be done asynchronously and while that happens, our program can do other work.

Asynchrony

To illustrate the concept let me give you a real-world example.

Asynchronous Pizza

Let’s say you are watching The Godfather and want to have pizza for dinner.

Asynchronous Pizza

You've got two options.

Asynchronous Pizza

First option: make it yourself.

Asynchronous Pizza

You stop the film, go to the kitchen and make the dough.

Asynchronous Pizza

You cover the dough with sauce, cheese and your favorite ingredients.

Asynchronous Pizza

Then you preheat the oven, stick it in there.

Asynchronous Pizza

When it’s ready you can eat pizza while you watch the film.

Asynchronous Pizza

That just took 30 minutes of your time and attention.

Asynchronous Pizza

And you couldn't do anything else until the pizza was done.

Asynchronous Pizza

Second option: order a pizza.

Asynchronous Pizza

You stop the film and order a pizza from your favorite pizzeria.

Asynchronous Pizza

You continue watching the film for 30 minutes.

Asynchronous Pizza

Then you get a call:
your pizza is here.

Asynchronous Pizza

You stop the film, receive your food and pay the delivery person.

Asynchronous Pizza

You can now eat pizza while you watch the film.

Asynchronous Pizza

That took the same amount of time, but you were able to use your time more effectively.

Asynchronous Pizza

Because you only had to use your time and attention to order the pizza and then receive it.

Asynchronous Pizza

Pizza example based on a [Stack Overflow answer] by Eric Pascarello.

Asynchrony

The point is, doing things asynchronously is less about calling functions to return values.

Asynchrony

And more about calling functions to *order* values and having them *delivered* to us.

Quiz

Which of these functions should be asynchronous?

```
saveToDatabase('Nizar');
readFile();
sortArray([ 40, 50, 20 ]);
add(20, 22);
wait(60);
```

Callbacks

In JavaScript, it's very common to use *callbacks* for asynchronous tasks.

Callbacks

A callback is a function you pass in as a value.

Callbacks

It’s just like blocks in Ruby.

Callbacks

So far, we've seen functions receive all kinds of values as arguments.

Callbacks

In JavaScript, since functions are also values, they can be used as arguments as well.

Callbacks

When you pass in a function as an argument, that function is a callback.

Callbacks

There are two styles of doing this.

Callbacks

One is to pass in a callback function value directly.

```
someAsyncFunction(function () {
  // Do stuff here when done
});
```

Callbacks

We call this style
the *function expression* style.

```
someAsyncFunction(function () {
  // Do stuff here when done
});
```

Callbacks

The other is to define a callback function first.

```
function done () {
  // Do stuff here when done
}

someAsyncFunction(done);
```

Callbacks

We call this style
the *named function* style.

```
function done () {
  // Do stuff here when done
}

someAsyncFunction(done);
```

Callbacks

Let's imagine what our `superSlowThing` would be like if it were an asynchronous function that accepted callbacks.

Callbacks

So we start with this.

```
// Takes 25 seconds
var result = superSlowThing(42);
```

Callbacks

Let’s get rid of the assignment.

```
// Takes 25 seconds
var result = superSlowThing(42);
```

Callbacks

Asynchronous functions don’t usually return anything.

```
// Takes 25 seconds
superSlowThing(42);
```

Callbacks

Now we give it a function argument.

```
// Takes 25 seconds
superSlowThing(42, callback);
```

Callbacks

In this case, we are using the named function style.

```
// Takes 25 seconds
superSlowThing(42, callback);
```

Callbacks

Of course, we need to define the function for this to work.

```
function callback () {
}

// Takes 25 seconds
superSlowThing(42, callback);
```

Callbacks

The callback will be provided the result through a parameter.

```
function callback (result) {
  // Use result
}

// Takes 25 seconds
superSlowThing(42, callback);
```

Callbacks

Final product: the proper way to do slow things in JavaScript.

```
function callback (result) {
  // Use result
}

// Takes 25 seconds
superSlowThing(42, callback);
```

Practical examples

Okay that's the theory.

Practical examples

What are actual cases in which you use these things?

Practical examples

Let’s look at a couple of basic asynchronous tasks in JavaScript.

`setTimeout`

JavaScript doesn't have a `sleep` function.

`setTimeout`

To execute code after a certain amount of time, you use the asynchronous `setTimeout`.

`setTimeout`

It works like this.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

setTimeout

The first argument is the callback.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

setTimeout

It will be invoked after the amount of time has elapsed.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

setTimeout

The second argument specifies the amount of time.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

setTimeout

Careful! You specify the amount of time in *milliseconds*.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

`setTimeout`

So this code will call `shout` after *one second*.

```
function shout () {
  console.log('Ahhhhhhhh!');
}

setTimeout(shout, 1000);
```

Exercise: `setTimeout`

Let's implement Ruby's `sleep`
but for JavaScript.

Exercise: `setTimeout`

Of course, it should be asynchronous and receive a callback:

```
sleep(10, function () {
  console.log('It's been 10 seconds.');
});
```

Big example:

`socket server`

What is a socket?

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

The server just waits, listening to the socket for a client to make a connection request.

What is a socket?

With amazing JavaScript asynchrony, we are going to create a socket server for ourselves...

socket server

```
var net = require('net');
var port = 1702;
var server = net.createServer(function(connection) {
    console.log('Connection to %s open', port);
    connection.write('Hello?\r\n');
    connection.on('data', function(data) {
        if (String(data).trim() != 'hello') {
            connection.write('ERROR\r\n');
        } else {
            connection.end('world\r\n');
            console.log('Connection to %s closed', port);
        }
    });
});
server.listen(port);
```

Continuations in yellow
Parameters in **bold**
Clausures in **cyan**

Killer Exercise:

`The simplecached`

First, we need to install a package called simplecached. Open a terminal. In your project folder execute

```
$ npm install simplecached
```

The Client:

To create a client in simplecached,

```
var simplecached = require('simplecached');

var options = {
    port: 11312,
    host: '127.0.0.1'
};
var client = new simplecached.Client(options, function(error)
{
    console.log('Connected');
});
```

The Client: these instructions can be called on the server

client.get(key, callback);

Get a key from the remote simplecached. The callback is a `function(error, result)` that will be called either with an error or the result, or null if the value was not found.

“GET KEY” <= The server just receives a string like this

client.set(key, value, callback);

Set a value into the remote simplecached. The callback is a `function(error, result)` that will be called with an error or a result. The `result` can be true if the value was stored, false otherwise.

“SET KEY VALUE“ <= The server just receives a string like this

client.delete(key, callback);

Delete a value from the remote simplecached. The callback is a `function(error, result)` that will be called with an error or a result. The `result` can be true if the value was deleted, false if not found.

“DELETE KEY” <= The server just receives a string like this

client.close(callback);

Close the connection. The optional callback will be called after the connection is actually closed.

“CLOSE” <= The server just receives a string like this

Killer Exercise:

`The simplecached`

The Challenge:

Your mission, should you choose to accept it, is to create a server that waits for a connection. A client opens a connection. Then, it sets a key, retrieves it and checks if the value is correct. Then, it should be able to close the connection.