UNDERSTANDING ASYNCHRONOUS JAVASCRIPT

TallyJS - June 2014

John Newman

@jnewmanux • github.com/jgnewman

rescuecreative@gmail.com

What is asynchronous programming?

> *There are two hard things in computer science: cache invalidation, **naming things**, and off-by-one errors.*
>
> *- Unknown*

Somebody go to Google and search **"define synchronous"**

Wikipedia

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved **concurrently** ("in parallel").

codewala.net

Asynchronous operation means that a process operates independently of other processes. If there are multiple operations, **[they] can be handled [using] different processes/threads without waiting** to complete the other ones.

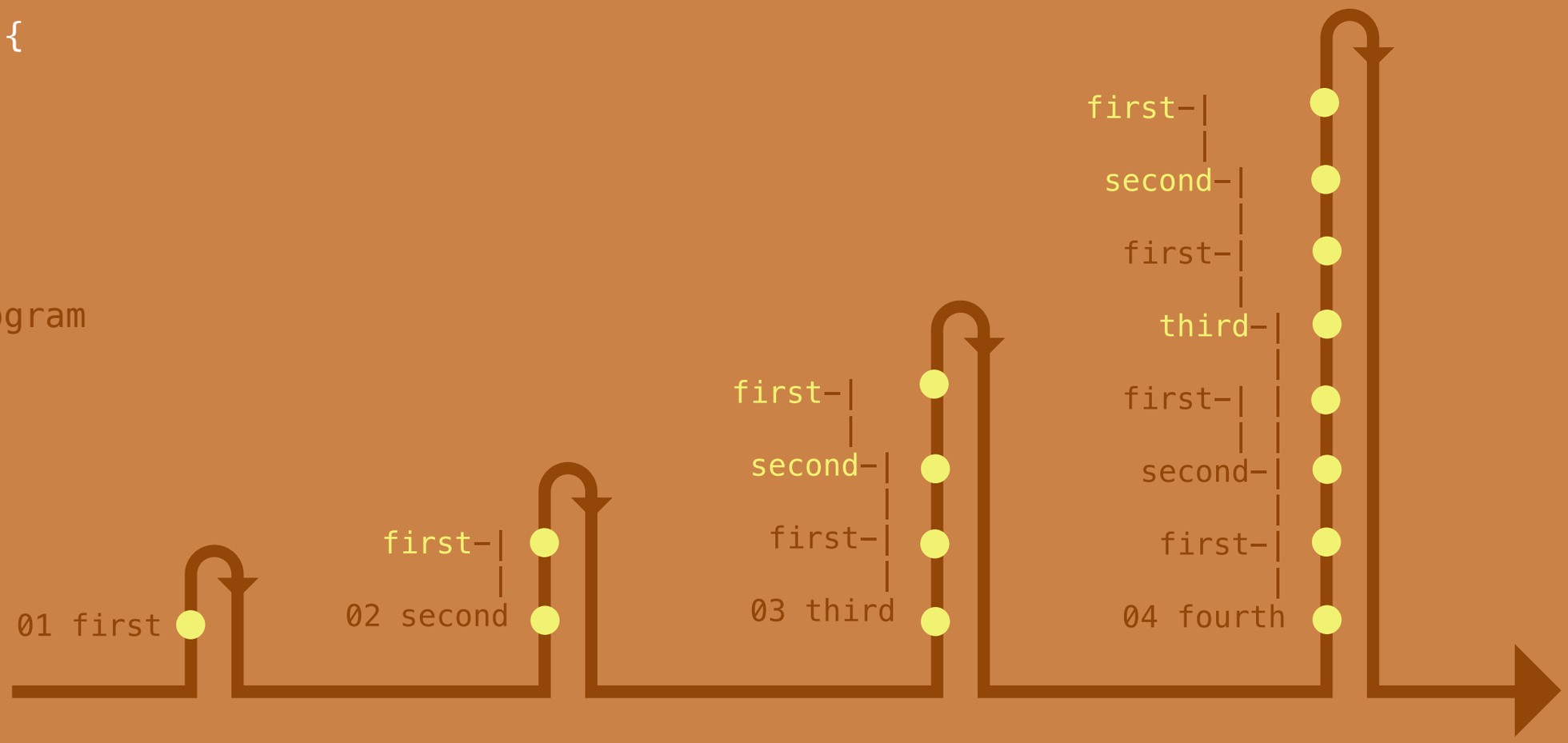
Our Simple Definition

When the flow of your program is such that code might be executed out of order or you might not be able to count on one thing happening before another thing.

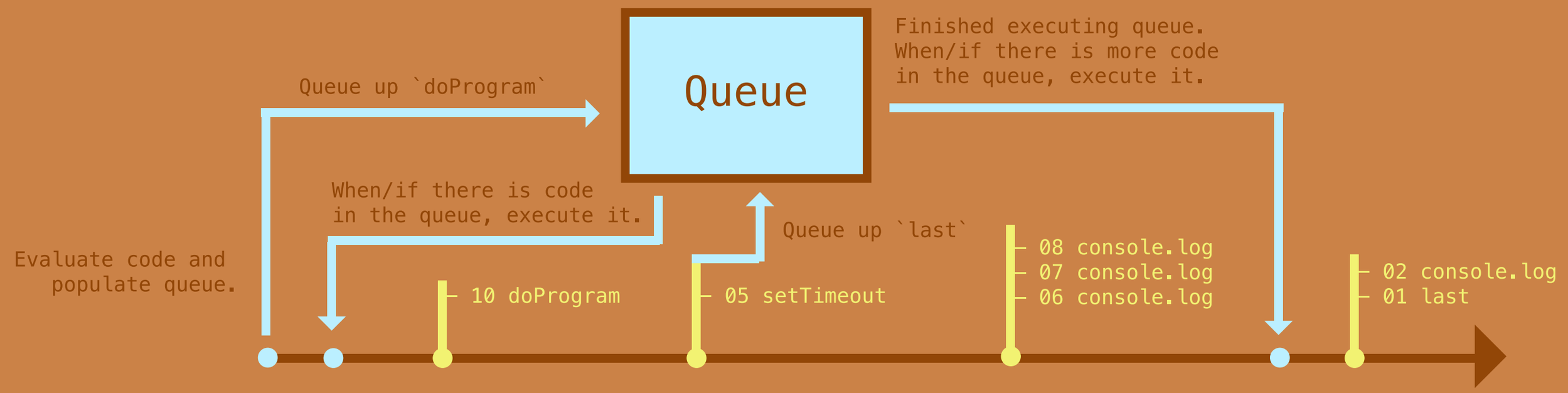
But why does (JavaScript) code run in order in the first place?

```
// Define some functions

function first() {
  return 'hello';
}

function second() {
  return first();
}

function third() {
  first();
  return second();
}

function fourth() {
  first();
  second();
  return third();
}

// Execute the program
01 first();
02 second();
03 third();
04 fourth();
```

Code is executed in a "single thread."



```
// Define some functions

01 function last() {
02   console.log("from L.A. to Tokyo");
03 }

04 function doProgram() {
05   setTimeout(last, 0);
06   console.log("i'm so fancy");
07   console.log("you already know");
08   console.log("i'm in the fast lane");
09 }

// Execute the program
10 doProgram();

//=> i'm so fancy
//=> you already know
//=> i'm in the fast lane
//=> from L.A. to Tokyo
```



> *When there is nothing to do,*
> *Check the queue.*
> *But only check the queue*
> *When there's nothing left to do.*
>
> *- JavaScript*

In other words, the current flow will **NEVER** be interrupted by code existing in the queue.

Let's prove it in the console!


```
// Add a click handler to the document.
document.addEventListener('click', function () {
  console.log('click happened!');
});

// Create a function that does nothing but waste time.
function wasteTime() {
  var i;
  for (i = 0; i < 3000000; i += 1) {
    new Array(10000000);
  }
  return 'finished wasting time';
}

// Start wasting time.
wasteTime();


//=> finished wasting time
//=> click happened!
```

Some similar examples that use the queue:


```
[ELEMENT].addEventListener('click', myFunction);

[ELEMENT].onclick = myFunction;

$([ELEMENT]).click(myFunction);
```

~~`<a onclick="myFunction“>click me</a>`~~

The queue matters because any time your program receives input, JavaScript puts the input handler into the queue.

So when we talk about "asynchronous JavaScript," we're really just talking about any code that puts things in the queue.

And if we can't know when code will enter the queue (for example, **application input**), we can't know when it will be executed.

Ways to Receive Input

1

DOM events such as clicks, hovers, keyups, etc.

Handlers go in the queue.

2

HTTP(s)/websocket communication events. (Includes $.ajax)

Handlers go in the queue.

3

Web worker message passing. (How we do threads in the browser)

Handlers go in the queue.

And if it's in the queue, you **can't** return it or error handle it.

Let's access some real data:


```
function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  return ajax;
}

getGithubInfo();
```

Let's access some real data:


```
function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  return ajax;
}

getGithubInfo();
```

NOPE!

But clearly that's because $.ajax is returning the jQuery request object instead of the actual data. Let's fix that.

Let's access some real data:


```
function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  return ajax.done(function (data) {
    return data;
  });
}

getGithubInfo();
```

Let's access some real data:


```
function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  return ajax.done(function (data) {
    return data;
  });
}

getGithubInfo();
```

STILL NOPE!

Right, but this time it's because .done is a Promise method so it's returning another Promise object. I got it this time...

Let's access some real data:


```
function getGithubInfo() {
  var result;
  $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  }).done(function (data) {
    result = data;
  });
  return result;
}

getGithubInfo();
```

Let's access some real data:


```
function getGithubInfo() {
  var result;
  $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  }).done(function (data) {
    result = data;
  });
  return result;
}

getGithubInfo();
```

FRACK! STILL NOPE!

Because no matter what we do, ajax requires an event handler and event handlers go into the queue. **Which means...**

It gets executed **AFTER** the function returns.

This time let's really do it:


```
function logResult(result) {
  console.log(result);
}

function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  ajax.done(logResult);
}

getGithubInfo();
```

This time let's really do it:


```
function logResult(result) {
  console.log(result);
}

function getGithubInfo() {
  var ajax = $.ajax({
    url: 'https://api.github.com/users/jgnewman/repos',
    dataType: 'jsonp'
  });
  ajax.done(logResult);
}

getGithubInfo();
```

SUCCESS!

What was that you said about error handling?

No error handling allowed:


```
function logResult(result) {
  console.log(result);
}

function getGithubInfo() {
  var ajax;
  try {
    ajax = $.ajax({
      url: 'fhqwhgads',
      dataType: 'jsonp'
    });
    ajax.done(logResult);
  } catch (err) {
    console.log('got an error', err);
  }
}

getGithubInfo();
```

The jQuery Solution:


```
function logResult(result) {
  console.log(result);
}

function getGithubInfo() {
  var ajax = $.ajax({
    url: 'fhqwhgads',
    dataType: 'jsonp'
  });
  ajax.done(logResult);
  ajax.fail(function (errData) {
    console.log('got an error', errData);
  });
 }

getGithubInfo();
```

Asynchrony can be hard to reason about when it gets complex.

So let's talk about some common techniques that can make it feel more synchronous.

CALLBACK FUNCTIONS

PROMISE OBJECTS

What Everyone Hates About Node.js

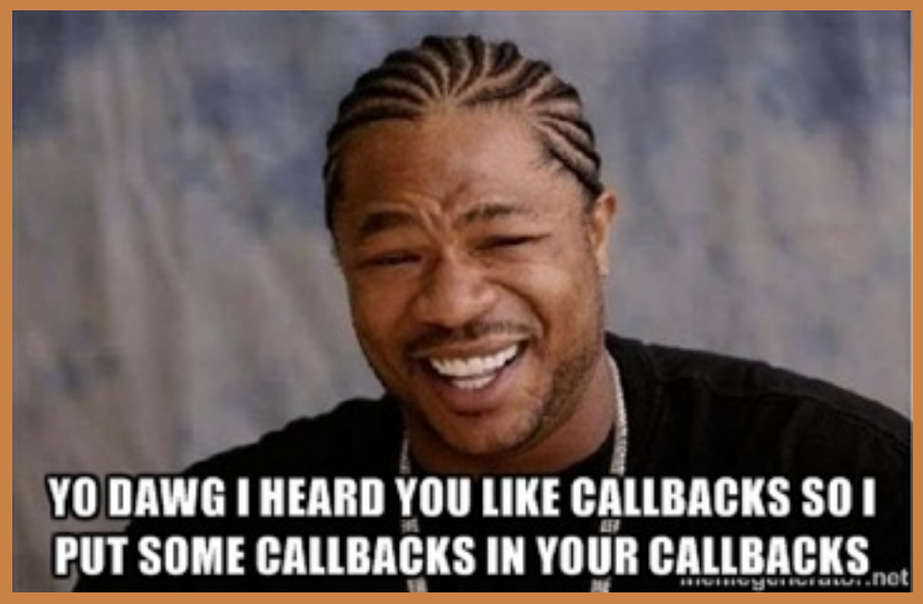

```
// Import the file system library
var fileSystem = require('fs');

// Call `readdir` (which is asynchronous) to get a list
// of files in the directory
fileSystem.readdir('/myfiles', function (err, data) {
  if (err) {
    handleTheError(err);
  } else {
    doSomethingWith(data);
  }
});
```

What's wrong with this pattern, you ask?

The fact that it gets nasty really fast:

```
/**
 * Show a list of songs for the artist with the given name and
 * execute the callback when complete.
 */
function showSongs(artistName, callback) {
    // Notice we're not even thinking about errors here
    artists.getByName(artistName, function (artist) {
        albums.getByArtist(artist, function (albums) {
            songs.getByAlbum(albums, function (songs) {
                songView.render(songs);
                return callback();
            });
        });
    });
}
```

example stolen from **clutchski** on slideshare



Can Promises fix that problem?

The can definitely ice that cake.

What is a Promise?

- It's an object with methods (usually **`done`**, **`then`**, and **`fail`**).
- Each method takes at least 1 callback.
- The callbacks are executed under different conditions (**`done`** or **`then`** on success, **`fail`** on error).
- Each Promise method returns another promise.
- Therefore, you can chain asynchronous actions.
- And you can attach as many callbacks as you want to a given event.

For example:


```
function myAsyncFn() {
  return new Promise(function (resolve, reject) {
    var async = somethingAsynchronous();
    async.addEventListener('success', resolve());
    async.addEventListener('error', reject());
  });
}
```

```
var myPromise = myAsyncFn();
myPromise.done(function () { …success stuff… });
myPromise.done(function () { …more success stuff… });
myPromise.fail(function () { …failure stuff… });
myPromise.fail(function () { …more failure stuff… });


myPromise.then(function () { return myAsyncFn() })
         .then(function () { return myAsyncFn() })
         .then(function () { return myAsyncFn() });
```

So let's go back to our ugly, nested callbacks.

Previously we had this:


```
/**
 * Show a list of songs for the artist with the given name and
 * execute the callback when complete.
 */
function showSongs(artistName, callback) {
    artists.getByName(artistName, function (artist) {
        albums.getByArtist(artist, function (albums) {
            songs.getByAlbum(albums, function (songs) {
                songView.render(songs);
                return callback();
            });
        });
    });
}
```

But if we used promises instead...


```
/**
 * Show a list of songs for the artist with the given name and
 * execute the callback when complete.
 */
function showSongs(artistName, callback) {
    artists.getByName(artistName)
        .then(function (artist) { return albums.getByArtist(artist) })
        .then(function (albums) { return songs.getByAlbum(albums) })
        .then(function (songs)  { return songView.render(songs) })
        .then(callback);
}
```

Promises are going to be native in ES6! (Currently 0 IE support)

jQuery

http://api.jquery.com/promise/

Q for Node.js

https://github.com/kriskowal/q

Tons of Others

http://promisesaplus.com/implementations

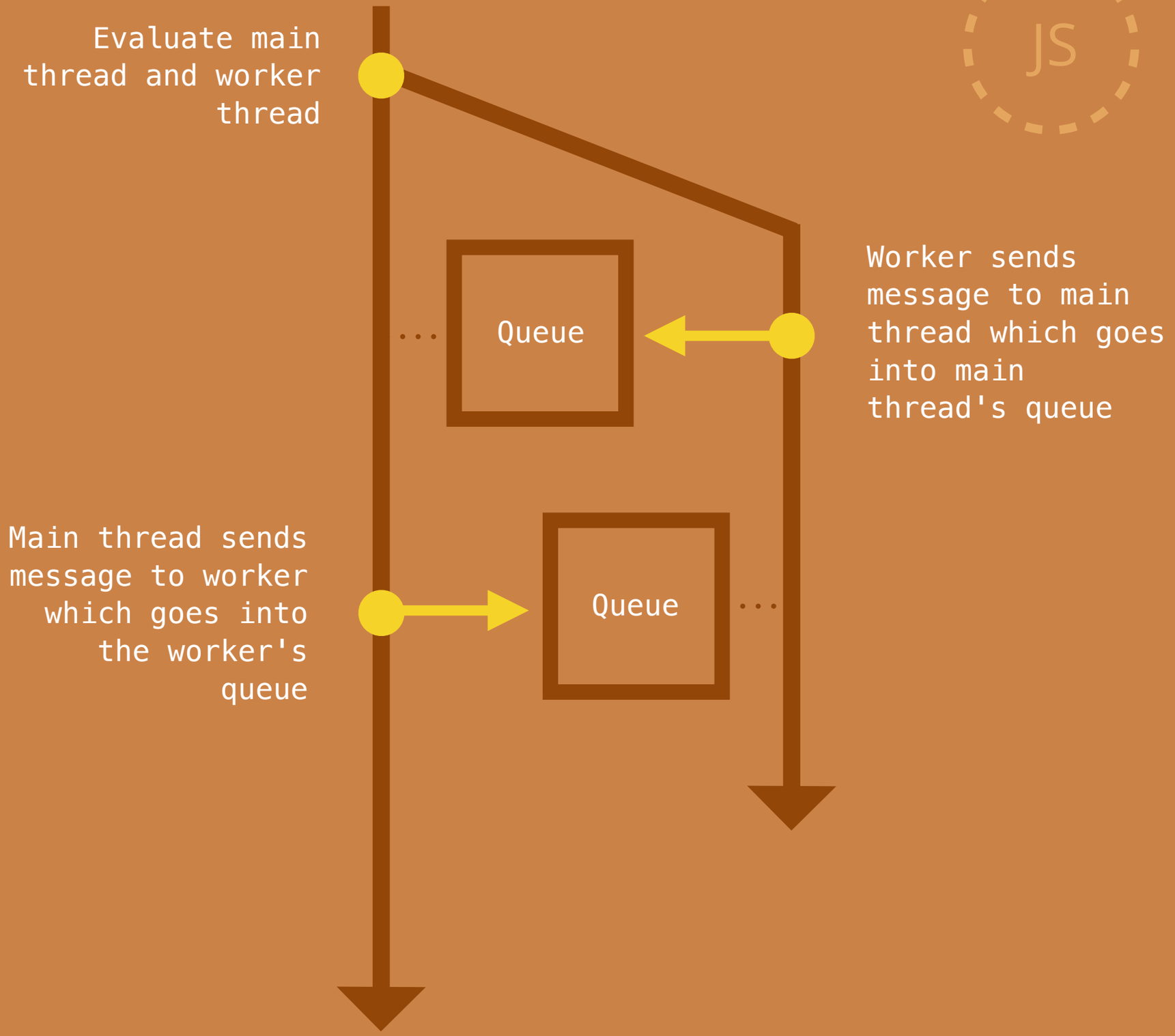
At this point, you may be wondering...

- Why asynchrony is always talked about in terms of concurrency when the word itself seems to imply the exact opposite
- What asynchronous JavaScript looks like using web workers
- When this talk is going to end

(Check the time, future John)

Concurrency becomes asynchrony when parallel threads talk to each other.

Each thread can be run on its own core thus making multiple actions **actually simultaneous.**



Every time a message is sent, that message goes into a queue thus making the code within each thread independently asynchronous.

A web worker is...

- A way to spin up multiple threads in browser-based JavaScript
- Theoretically, an extremely helpful tool that has been much needed for quite some time
- In reality, a fairly annoying and cumbersome implementation of a concept that ought to have been much simpler


```
var worker = new Worker('myworker.js');

worker.onmessage = function (evt) {
  console.log('The worker sent', evt.data);
};

worker.postMessage('');
```

main.js


```
self.onmessage = function () {
  self.postMessage('message received');
};
```

myworker.js

There's also a "secret" way to do it with one file:


```
// The intended functionality of our web worker
function workerBody() {
  self.onmessage = function () {
    self.postMessage('message received');
  };
}

// Takes a function and turns it into a web worker
function createWorker(fn) {
  var body = fn.toString().replace(/^.+\{|\}\s*$/g, ''),
      blob = new Blob([body], {type: 'text/javascript'});
  return new Worker(window.URL.createObjectURL(blob));
}

// Create and use the worker
var worker = createWorker(workerBody);
worker.onmessage = function (evt) {
  console.log('The worker sent', evt.data);
};
worker.postMessage('');
```

Let's use it to make life a bit easier:


```
// The minified worker maker
function createWorker(f){var b=f.toString().replace(/^.+\{|\}\s*$/g,''),l=new
Blob([b],{type:'text/javascript'});return new Worker(window.URL.createObjectURL(l))}

// A worker body containing a modified version of our time wasting function from before.
// Creates 3 million arrays with 10 million items and returns the amount of
// time that took.
function workerBody() {
  function wasteTime() {
    var i, begin = Date.now();
    for (i = 0; i < 3000000; i += 1) {
      new Array(10000000);
    }
    return Date.now() - begin;
  }
  self.onmessage = function () {
    var time = wasteTime();
    self.postMessage('Wasted ' + time + 'ms of time.');
  };
}

// Set up a click handler on the document
var clicks = 0;
document.addEventListener('click', function () { console.log('click', ++clicks, 'happened') });

// Launch the worker and try it out.
var worker = createWorker(workerBody);
worker.onmessage = function (evt) { console.log(evt.data) };
worker.postMessage();
```

Check out where I actually use this concept...

STREAMPUNK

A simple & surprisingly powerful application framework that offers data binding, single page apps, and tons of other tools.

streampunkjs.com

SOLID FUEL SOCKET BOOSTER

A library for the browser that allows you to take common ajax/websocket actions and dynamically run them within a web worker.

github.com/jgnewman/sfsb

You can also check out htmlgoodies.com for more examples of Web Workers in action.

http://www.htmlgoodies.com/html5/client/introduction-to-html5-web-workers-use-cases-and-identify-hot-spots-.html

THANK YOU!

John Newman
@jnewmanux
rescuecreative@gmail.com