HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

02 - Javascript foundation

Use testing to learn async javascript

Daily menu

- Introduction to Javascript
- Javascript foundations
- Asynchronous programming
- Expand our development environment
  - Add a testing framework
  - Make it a learning environment

Introduction to JavaScript

Javascript language

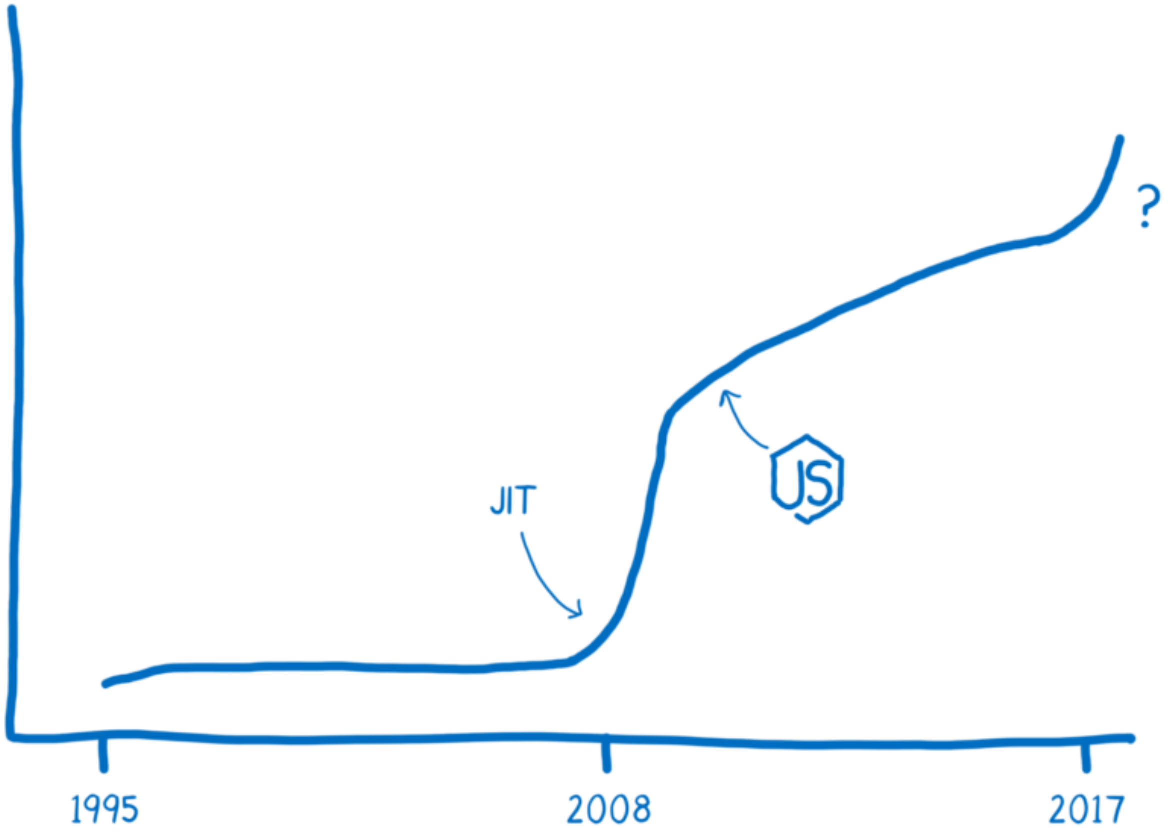
- Javascript was written in only 10 days
- Some people thinks it's a **mess** while others think that it's **great**
- A lot of data analysis platforms prove that it is one of the **most popular** technology in the engineering world
  - StackOverflow survey 2018
  - Jetbrains survey 2018

What makes javascript so popular?

- **Simplicity**: Simple to learn and implement
- **Productivity**: Fast deployment, test and debug without *compilation* process
- **Platform Independence**: Executes on any device where there exists a special program called the JavaScript engine - Browsers, servers, desktop/mobile apps.
- **Speed**: Thanks to existing javascript engines, it keeps getting faster every year !

Javascript performance

Javascript wasn't designed to be fast and for the first decade, it wasn't fast - after the introduction of **just-in-time compilers** (JIT) in **javascript engines**, it became 10x faster.



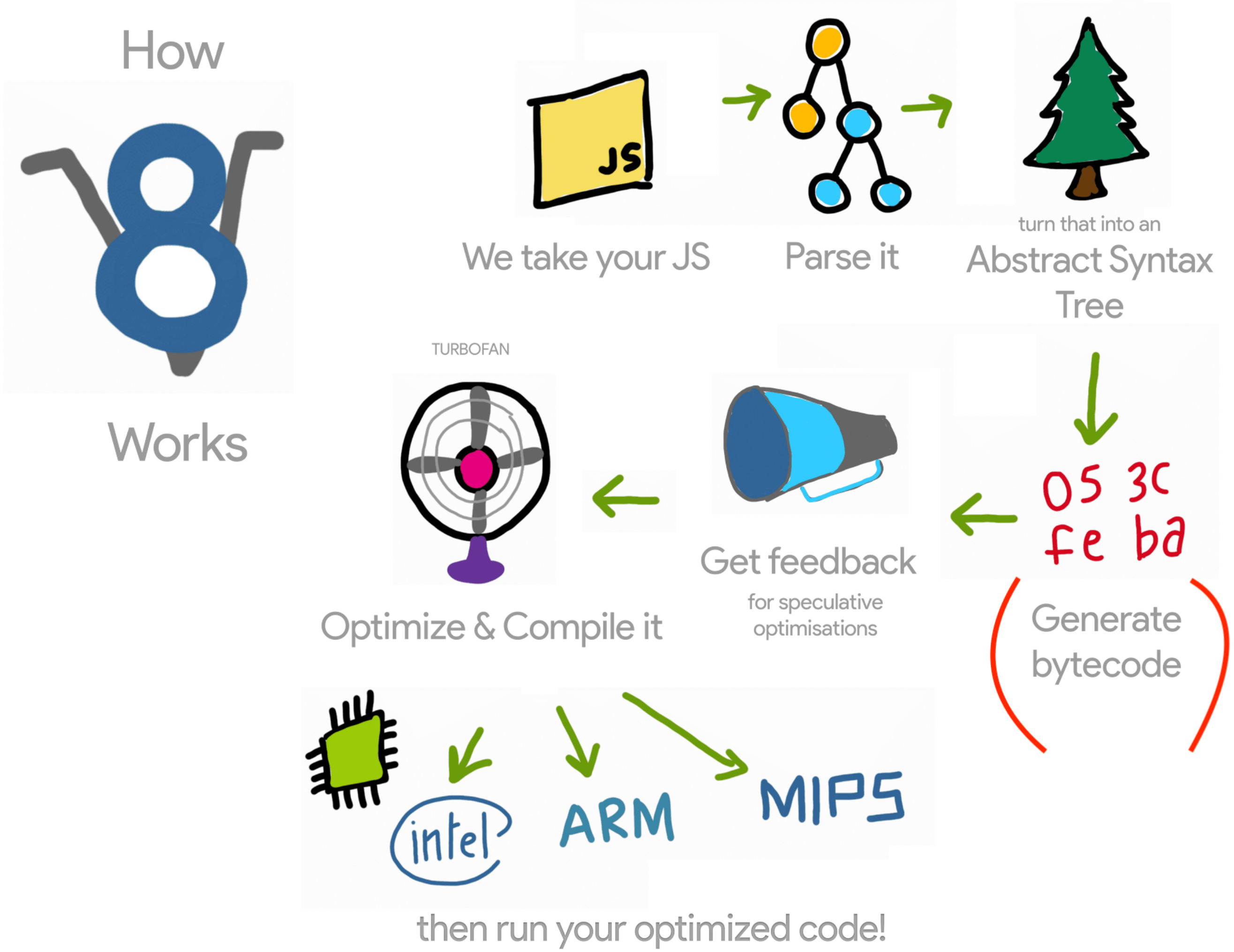
Wait, what is a javascript engine?

A javascript engine, sometimes called "JavaScript virtual machine", take the JavaScript code that a developer writes and convert it to fast, optimized code that can be interpreted by a browser or even embedded into an application.

Each JavaScript engine implements a version of ECMAScript, of which JavaScript is a dialect

- **V8 – in Google Chrome and Opera, as well in Node.js**
- SpiderMonkey - in Mozilla Firefox.
- JavaScriptCore - in Safari
- Chakra - in Microsoft Edge

How V8 works?



By @addyosmani

How JavaScript is run in the browser?

In programming, there are generally two ways of translating to machine language. You can use an interpreter or a compiler.

Interpreter
The translation happens line-by-line, on the fly. Quick to get up and running. But for a code in a loop, the same translation happens over and over.

Compiler
The translation happens ahead of time. Takes necessary time to optimize the code - the code runs faster. But takes a little bit more time to start up.

Just-in-time compilers

Mixing different types of compilers to get rid of the interpreter’s inefficiency.

- **Monitor** (profiler) - Identifies hot functions as they execute and what types are used.
- **Baseline compiler** - A fast compiler that produces unoptimized code. (almost equivalent to an interpreter)
- **Optimizing compiler** - A slower compiler that produces fast, optimized code.

V8' Just-in-time compiler

1. When first executing the JavaScript code, the `baseline compiler` is used to start executing machine code very fast.
2. During the execution, `the monitor` runs in an other thread and identifies hot functions that should be optimized and gathers statistics.
3. The `Optimizing compiler` optimizes hot functions in an other thread. Theses optimizations are based on assumptions made by the `monitor`
4. If assumptions aren't valid anymore, the execution goes back to the `baseline compiler` version. This process is called deoptimization.

References

- Lin Clark: A Cartoon Intro to WebAssembly | JSConf EU 2017
- How JavaScript works - inside the V8 engine
- JavaScript Start-up Performance

What can in-browser JavaScript do?

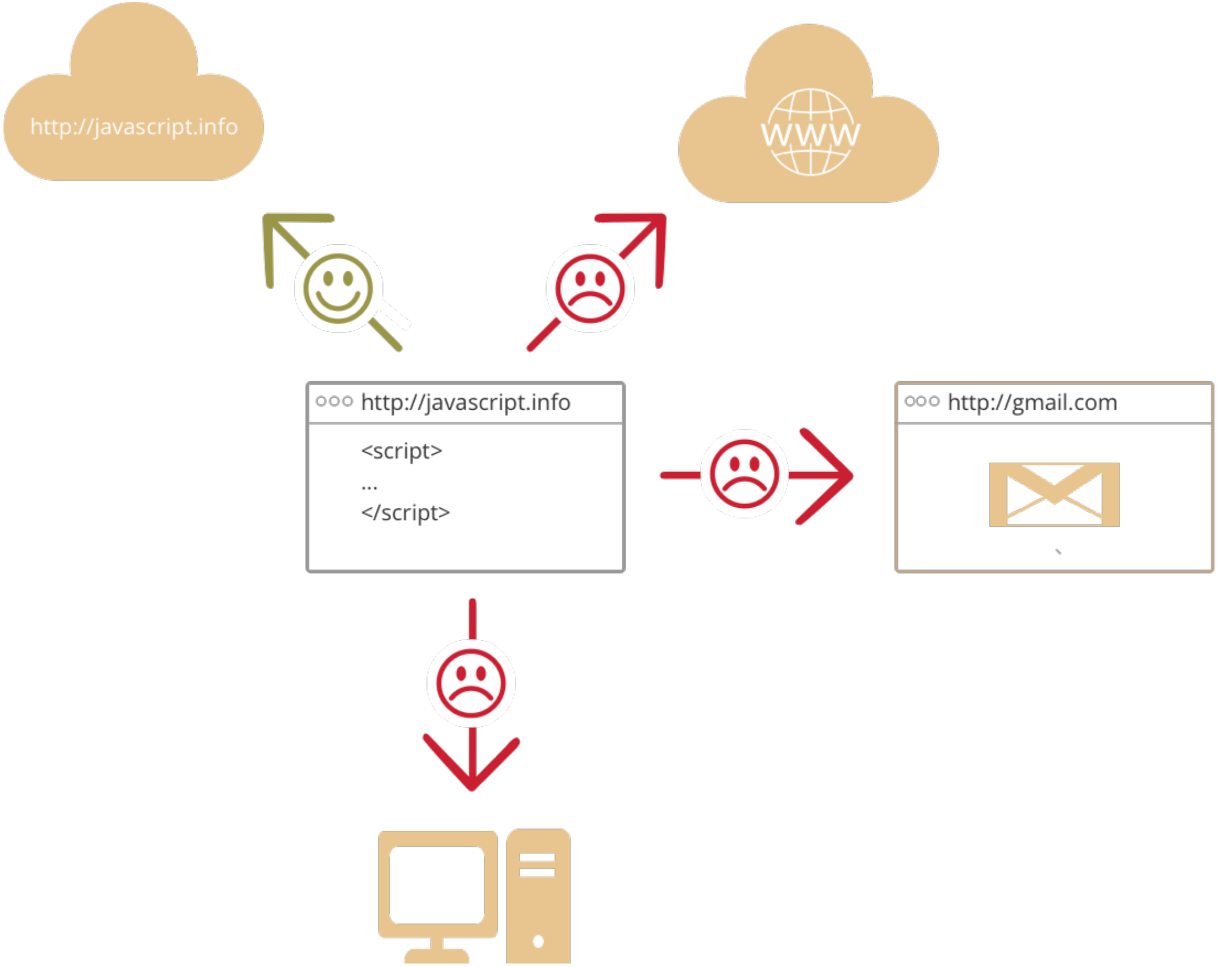
- Add new HTML to the page, change the existing content, modify styles.
- React to user actions, run on mouse clicks, pointer movements, key presses.
- Send requests over the network to remote servers, download and upload files
- Get and set cookies, ask questions to the visitor, show messages.
- Remember the data on the client-side (“local storage”).

What CAN’T in-browser JavaScript do?

JavaScript’s abilities in the browser are limited for the sake of the user’s safety.

- read/write arbitrary files on the hard disk, copy them or execute programs.
- There are ways to interact with camera/microphone and other devices, but they require a user’s explicit permission
- Different tabs/windows generally do not know about each other.
- Communicate with a server from a different domain - unless an explicit agreement (expressed in HTTP headers) is given by the server

In-browser javascript limitations



Javascript foundations

Variables

We can declare variables to store data. That can be done using `var`, `let` or `const`.

- `let` – is a modern variable declaration. The code must be in strict mode to use let in Chrome (V8).
- `var` – is an old-school variable declaration with subtle differences from `let`.
- `const` – is like `let`, but the value of the variable can’t be changed.

“var” has no block scope

`var` variables are either function-wide or global, they are visible through blocks

```
// first-script.js
if (true) {
  var myVar = 'hello'; // use "var" instead of "let"
}
console.log(myVar); // hello, the variable lives after if
```

But `var` ignores code blocks, so we’ve got a global test. Imagine an other script that runs on the same page

```
// second-script.js
console.log(window.myVar); // hello
console.log(myVar); // hello
```

`let` has block scope. If we used `let`, then `myVar` would be only visible in its current block (inside the `if`).

Types

A variable in JavaScript can contain any data. A variable can at one moment be a string and later receive a numeric value

Programming languages that allow such things are called “dynamically typed”, meaning that there are data types, but variables are not bound to any of them.

There are 7 basic types in JavaScript.

- `number` for numbers of any kind: integer or floating-point.
- `string` for strings. A string may have one or more characters, there’s no separate single-character type.
- `boolean` for true/false.
- `null` for unknown values
- `undefined` for unassigned values
- `object` for more complex data structures.
- `symbol` for unique identifiers.

Types

The typeof operator returns the type of the argument. The call to `typeof` returns a string with the type name:

```
typeof undefined     // "undefined"
typeof 0             // "number"
typeof true          // "boolean"
typeof "foo"         // "string"
typeof Symbol("id")  // "symbol"
```

```
typeof alert         // function 🤔
typeof [1, 2]        // "object" 😥
typeof null          // "object" 😱
```

Remember, Javascript was written in 10 days. `typeof null === 'object'` is a bug that stands since the beginning of JavaScript. Read More.

Stranger things

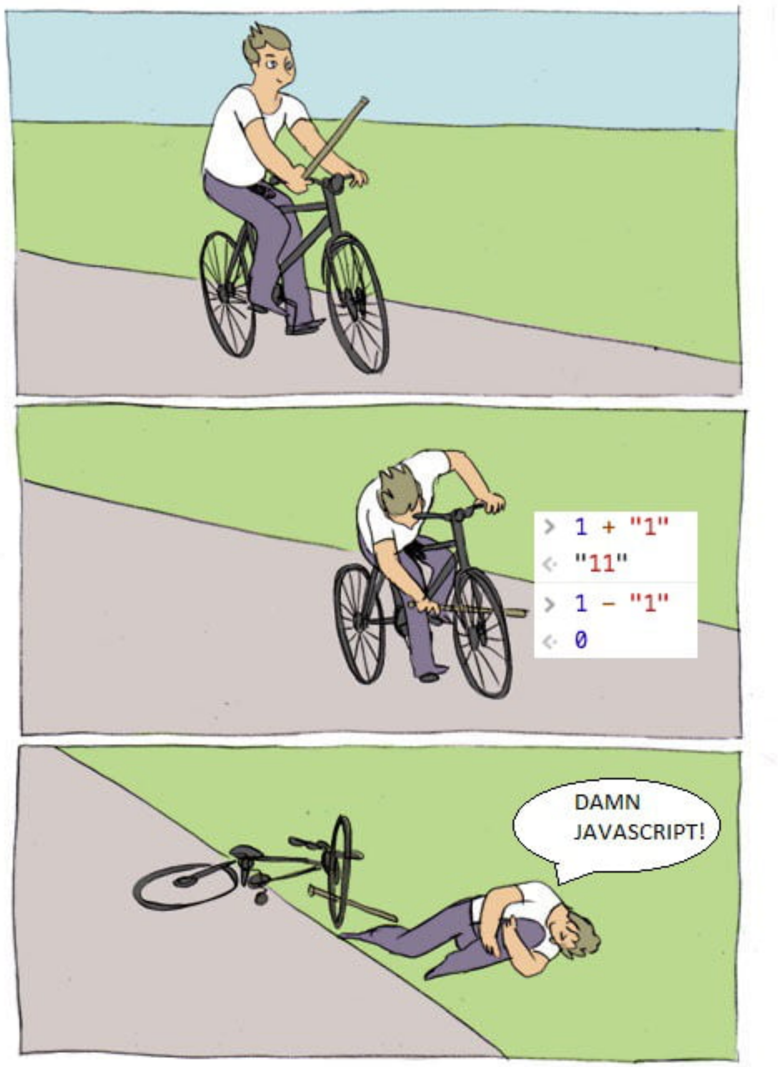
```
Boolean(0)     // false
Boolean("0")   // true
0 == "0"       // true!
```

Stranger things

```
Boolean(0)     // false
Boolean("0")   // true
0 == "0"       // true!
```

Javascript foundations

Even more strange



```
> 1 + "1"
<· "11"
> 1 - "1"
<· 0
```

Comparison

When comparing values of different types, they are **converted to numbers**.

```
'2' > 1 // true, string '2' becomes a number 2
'01' == 1 // true, string '01' becomes a number 1
```

For boolean values, `true` becomes `1` and `false` becomes `0`, that’s why:

```
true == 1      // true
false == 0     // true
```

Finally, that's why:

```
true == '1'    // true
```

In the above example, `true` becomes `1` and `'1'` becomes `1`

Comparison

A strict equality operator `===` and `!==` checks the equality without type conversion.

If `a` and `b` are of different types, then `a === b` immediately returns false without an attempt to convert them.

```
true === 1            // false
null === undefined   // false
```

Comparison

The previous examples may be confusing. To avoid confusions and making mistakes, here is what you should do:

- Follow best practices - Comparison Operators & Equality from Airbnb javascript style guide
- Write automated tests
- Understand how it works in details - Truth Equality and JavaScript

Functions

A function can be created at one moment, then copied to another variable or passed as an argument to another function and called from a totally different place later.

```
const sayHi = () => { alert('Heu... Hi!'); };

function sayHiSoon() {
  setTimeout(sayHi, 2000);
}

sayHiSoon() // Heu.. Hi!, in 2 sec
```

Javascript foundations

Functions

Function arguments are always passed by value

```javascript
function doSomething(value) {
  value = "modified";
}

let name = "original";

doSomething(name);
console.log(name); // original
```

Even if it's not recommend to do this, changing the value of the variable never changes the underlying primitive

Functions

Function arguments are always passed by value

```
function doSomething(value) {
  value = "modified";
}

let name = "original";

doSomething(name);
console.log(name); // original
```

Even if it's not recommend to do this, changing the value of the variable never changes the underlying primitive

> Never reassign parameters. eslint: `no-param-reassign` Why? Reassigning parameters can lead to unexpected behavior, especially when accessing the arguments object. It can also cause optimization issues, especially in V8. Read more...

Functions

However, when a variable refers to an object which includes array, the value is the reference/address to the object.

```
function doSomething(obj) {
  obj.age = 26;
}

let person = { age : 36 }

doSomething(person);
console.log(person.age); // 26
```

Changing the argument inside the function affect the variable passed from outside the function

> Never reassign parameters. eslint: `no-param-reassign` Why? Manipulating objects passed in as parameters can cause unwanted variable side effects in the original caller.

Local variables

A variable declared inside a function is only visible inside that function.

```javascript
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable
  console.log(message);
}

showMessage(); // Hello, I'm JavaScript!

console.log(message); // ReferenceError: message is not defined
```

Outer variables

A function can access an outer variable as well, for example:

```
let userName = 'paul';

function sayHi() {
  let message = `Hi, ${userName}`;
  console.log(message);
}

sayHi(); // Hi, paul
```

Outer variables

If a same-named variable is declared inside the function then it shadows the outer one.

```
let userName = 'paul';

function sayHi() {
  let userName = 'john'
  let message = `Hi, ${userName}`;
  console.log(message);
}

sayHi(); // Hi, john
console.log(userName) // paul, not modified by the function
```

What's the output of this function?

```javascript
let name = "Paul";

function sayHi() {
  console.log("Hi, " + name);
}

name = "Miguel";

sayHi()
```

What's the output of this function?

```
function makeWorker() {
  let name = "Miguel";

  return function() {
    console.log(name);
  };
}

let name = "Paul";

// create a function
let work = makeWorker();

// call it
work();
```

Closures

A closure is a function that remembers its outer variables and can access them

```javascript
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}

let counter1 = makeCounter();
let counter2 = makeCounter();

console.log(counter1()); // 0
console.log(counter1()); // 1
console.log(counter2()); // 0 (independent)
```

References

- Javascript.info - Open source tutorials on javascript and Node.js

Asynchronous programming

Synchronous code

Many operations in javascript are synchronous.

```
function getValue1(){
  return 1
}

function getValue2(){
  return 2
}

const value1 = getValue1();
const value2 = getValue2();
console.log(value1 + value2);   // 3
```

This code is synchronous - `getValue2()` has to wait for `getValue1()` to return before executing.

Synchronous code

The following code is still synchronous

```
function getValue1(){
  while(true);
}

function getValue2(){
  return 2
}

const value1 = getValue1();
const value2 = getValue2();
console.log(value1 + value2); // Never reached
```

`getValue2()` has to wait **forever** for `getValue1()` to return before executing.

Asynchronous code

Many actions in javascript are asynchronous. Instead of waiting for a function to return before moving on, JavaScript will keep executing.

Common asynchronous operations are for example I/O operations:

- Reading and writing files (Node.js)
- Querying data from a database (Node.js)
- Fetching data from an API (in-browser and Node.js)

Asynchronous code

Let's take a look at a asynchronous code.

```
function first(){
  setTimeout(function() {
    console.log(1)
  }, 2000)
}

function second(){
  console.log(2)
}

first();
second();
```

This code will produce the following output

```
2
1
```

Why do we need callbacks ?

A function that does something asynchronously should provide a callback argument where we put the function to run after it’s complete.

For example `setTimeout` is available in javascript (browser) and Node.js with the following syntax :

```
setTimeout(callback, milliseconds)
```

```
setTimeout(function() {
  console.log(“the callback has been invoked”);
}, 2000);
```

An event will be added to the queue in 2000 ms. In other words, the function passed as the first argument will be invoked in 2 seconds or more (the thread might be busy when the event is posted…).

Asynchronous programming

Callbacks

```
// Node.js
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

An event will be added when the file has been fully read (in a non-blocking way). When the event is taken out of the queue, the callback function has access to the file content (data).

Callbacks

```
// Event listener using JQuery
$(document).mousemove(function(event){
  $("span").text(event.pageX + ", " + event.pageY);
});
```

An event will be added to the queue whenever the mouse moves. In each case, the callback function has access to the event attributes (coordinates, key states, etc.).

```
// Ajax request using JQuery
$.get("ajax/test.html", function( data ) {
  $( ".result" ).html( data );
  alert( "Load was performed." );
});
```

An event will be added when the AJAX request has been processed, i.e. when a response has been received. The callback function has access to the payload.

Beyond simple callbacks

The principle of passing a callback function when invoking an asynchronous operation is pretty straightforward.

Things get more tricky as soon as you want to coordinate multiple tasks.

1st attempt

```javascript
let milkAvailable = false;

function milkCow() {
  console.log("Starting to milk cow...");
  setTimeout(function() {
    console.log("Milk is available.");
    milkAvailable = true;
  }, 2000);
}

milkCow();
console.log("Can I drink my milk? (" + milkAvailable + ")");
```

Solution

```
let milkAvailable = false;

function milkCow(done) {
  console.log("Starting to milk cow...");
  setTimeout(function() {
    console.log("Milk is available.");
    milkAvailable = true;
    done()
  }, 2000);
}

milkCow(function () {
  console.log("Can I drink my milk? (" + milkAvailable + ")");
});
```

Sequence

Ok... but what happens when I have more than 2 tasks that I want to execute in sequence?

```
function display(value) {
  console.log("display " + value);
}

function displaySoon(value) {
  setTimeout(function timer() {
    console.log("display soon " + value);
  }, 2000);
}

display(1);
display(2);
displaySoon(3);
displaySoon(4);
displaySoon(5);
display(6);

// outputs: 1, 2, 6, 3, 4, 5
```

Analyze it with loupe

Sequence

```
function display(value) {/* ... */}

function displaySoon(value, callback) {
  setTimeout(function timer() {
    console.log("display soon " + value);
     callback()
  }, 2000);
}

display(1);
display(2);
displaySoon(3, function() {
  displaySoon(4, function() {
    displaySoon(5, function() {
      display(6);
    });
  });
});
```

Analyze it with loupe

Parallel

Now, let’s imagine that we have 3 asynchronous tasks. We want to invoke them in parallel and wait until all of them complete.

Typical use case: you want to send several AJAX requests (to get different data models) and update your DOM once you have received all responses.

```
function fetchAll(done) {
  fetchData('/api/users/user1');
  fetchData('/api/users/user2');
  fetchData('/api/users/user3');
  done();
}
```

Parallel

Now, let’s imagine that we have 3 asynchronous tasks. We want to invoke them in parallel and wait until all of them complete.

Typical use case: you want to send several AJAX requests (to get different data models) and update your DOM once you have received all responses.

```
function fetchAll(done) {
  fetchData('/api/users/user1');
  fetchData('/api/users/user2');
  fetchData('/api/users/user3');
  done();
}
```

Double fail: not only do I invoke `done()` too early, but also I don’t have any result to send back...

Parallel

```
function fetchAll(done) {
  const results = [];
  let numberOfPendingTasks = 3;

  function reportResults(result) {
    results.push(result);
    numberOfPendingTasks -= 1;
    if (numberOfPendingTasks === 0) {
      done(results);
    }
  }

  fetchData('/api/users/user1', reportResults);
  fetchData('/api/users/user2', reportResults);
  fetchData('/api/users/user3', reportResults);
}
```

Edit on CodeSandbox

Webcasts

- Async with callbacks (1) : overview
- Async with callbacks (2): create a sync version first
- Async with callbacks (3): refactor the class and introduce a private function
- Async with callbacks (4): refactor the class: async signature and modification of the test suite
- Async with callbacks (5): fixing the problem and calling done() when everything is done
- Async with callbacks (6): write a function to fetch all pages

References

- Philip Roberts: What the heck is the event loop anyway? | JSConf EU

Automated testing

Why should I write automated tests?

- Automated testing is important for **quality** and continuous delivery
- Writing tests is also an approach to **design** and **document** software (TDD)

Solution

- Select a testing framework: mocha.js
- Select an assertion library: chai.js
- Write tests to get familiar Javascript

Install Mocha

First you need a test runner. Mocha is a popular testing framework that runs your tests serially and show results in your terminal.

Create a javascript project, with a `test` folder and a `sample-test.js` file :

```
my-project
├── test
│   └──sample-test.js
└── package.json
```

Then run the following command to install mocha locally as a development dependency :

```
$ npm install --save-dev mocha
```

Run a simple test

Write a simple test suite, then run `./node_modules/.bin/mocha`

```
// test/sample-test.js
const assert = require('assert');

describe('String', function () {
  it('should replace some characters', function () {
    const name = 'paulnta'
      .replace('au', 'o')
      .replace('n', 'en');
    assert.equal(name, 'polenta');
  });

  it('will fail', function () {
    assert.equal('1' + '1', '2');
    // AssertionError [ERR_ASSERTION]: '11' == '2'
  });
});
```

By default mocha will execute any `.js` files inside `test` folder and report results in your terminal.

Install Chai

In the previous example we're using Node.js' built-in assert module. - But Mocha allows you to use any assertion library you wish.

In practice we often add an another assertion library such as Chai to get more powerful features :

```
expect(name).to.be.a('string');
expect(polenta).to.have.a.property('color')
  .with.lengthOf(6);
```

Install chai via npm as follows :

```
$ npm install --save-dev chai
```

Install Chai

The previous example could be re-written as follows using `expect` from `chai`

```
+const { expect } = require('chai');

describe('String', function () {
  it('should transform name', function () {
    const name = 'paulnta'.replace('au', 'o').replace('n', 'en');
+   expect(name).to.equal('polenta');
  });

  it('will fail', function () {
+   expect('1' + '1').to.equal('2');
  });
});
```

Chai allows you to use different assertion styles and plugins. Use what makes the most sense for your project.

Running tests with npm

To make your tests easily runnable, add a `test` command to the `script` field of your `package.json`

```
// package.json
"scripts": {
  "test": "mocha test/**/*.js"
},
```

Now you can just type the following command to run tests with the configuration you specified.

```
$ npm test
```

Webcasts

- TDD with mocha and chai (1): overview
- TDD with mocha and chai (2): install npm modules
- TDD with mocha and chai (3): implement test module
- TDD with mocha and chai (4): implement module
- TDD with mocha and chai (5): refactor to es6 and validate
- TDD with mocha and chai (6): one more thing...