Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

JavaScript

... y un poco de Node.js

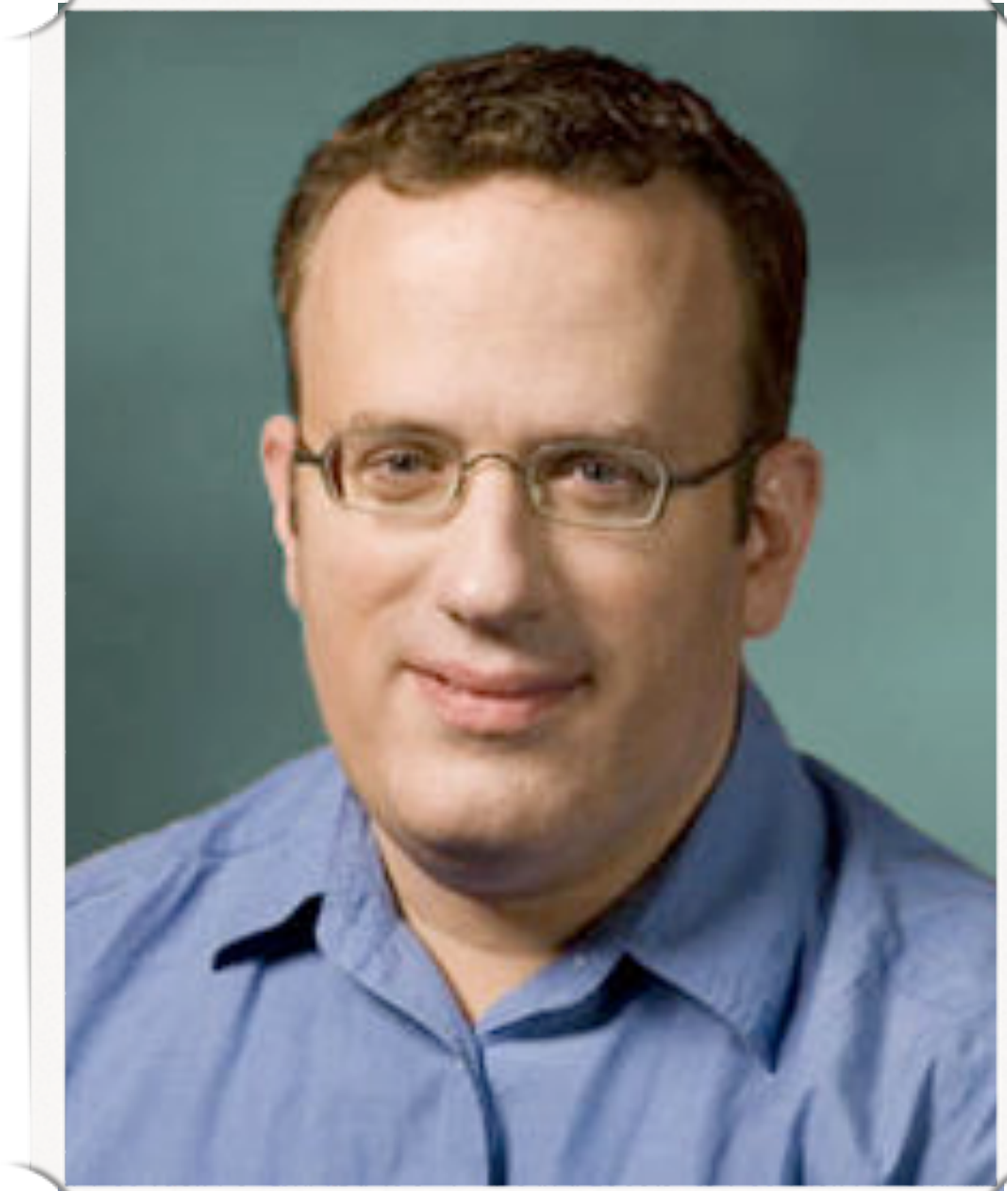
Raúl Montes T.

JavaScript

I'm not BAD...
I'm just
MISUNDERSTOOD!

JavaScript ≠ Java

Se parecen tanto como casilla a silla...



Brendan Eich

1995

Netscape

Nació en medio de las...

Mocha (codename)

LiveScript (1995) - Netscape 2.0 betas

JavaScript (1996)

JScript (Microsoft)

ECMAScript (ECMA-262, 1997)

ECMAScript (ECMA-262, 1997)

ECMAScript 2nd Edition, 1999

ECMAScript 3rd Edition, 1999

ECMAScript 5th Edition, 2009

ECMAScript 5.1, 2011

ECMAScript 6th Edition, Junio 2015

ECMAScript 6th Edition, Junio 2015

“ES6” ➜ “ES2015”

ECMAScript 7th Edition, Junio 2016

ECMAScript 8th Edition, Junio 2017

...

¿Qué versión usamos?

Depende del contexto...

Browser ➜ ES5*

latest Node.js ➜ ES2016 / ES2017

* P: ¡buuu! ¿en serio tan antigua version?
  R: a menos que uses un *transpiler*

¿Cómo se relacionan Node.js con JavaScript?

1995

JavaScript comenzó como un lenguaje de programación para el browser

2008

V8, JavaScript engine open source de Chrome

2009

Node.js combinó V8 con I/O APIs de bajo nivel para así poder ejecutarse en un ambiente de servidor

El “Hello World!”

```
console.log("Hello World!");
```

Node.js server

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hello World!');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

CARACTERÍSTICAS DEL LENGUAJE

JavaScript is a **dynamically** typed language

JavaScript is a **weakly** typed language

JavaScript es **multiparadigma**

object oriented

imperative/procedural

functional

JavaScript is an Object Oriented language

JavaScript doesn't have classes *

JavaScript has prototypes

* desde ES2015 existe *syntax sugar* para clases

¿Object Oriented sin clases?

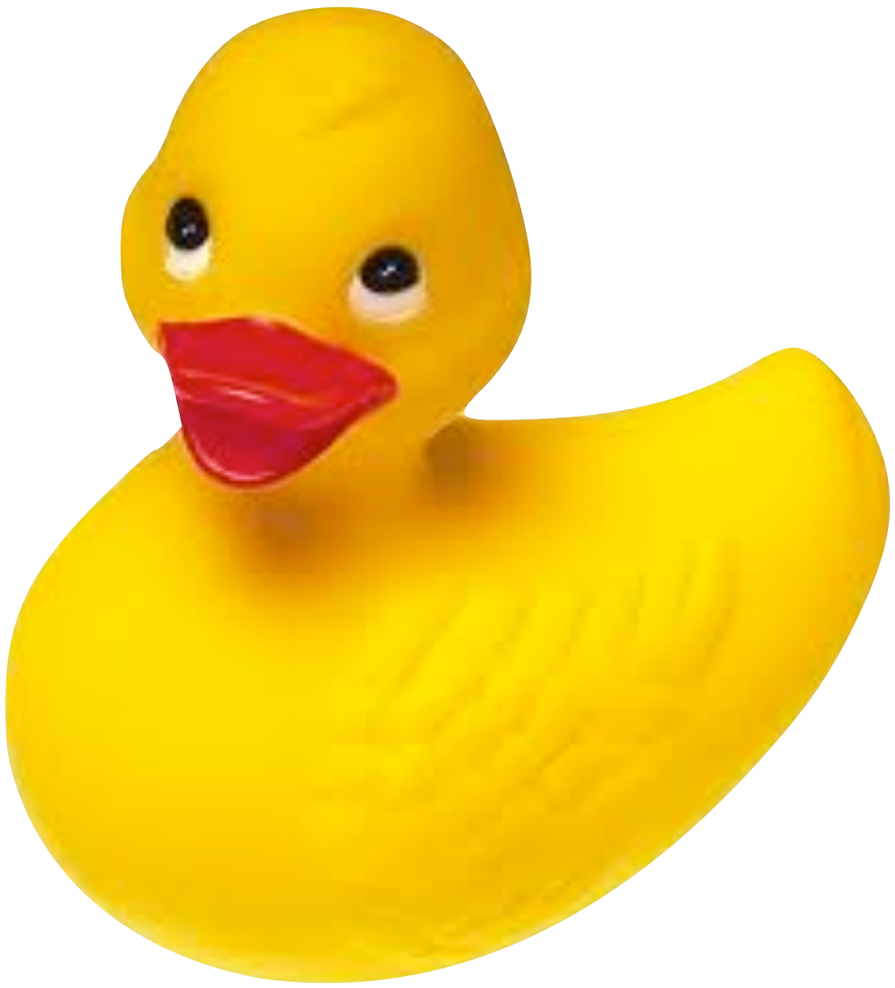
Constructor functions + prototypal inheritance

JavaScript has First class functions

... y con closures

Las variables se declaran con **let** o **const**

Sino, pertenecen al Global Object



Duck Typing

Volvamos al Node.js server...

Node.js server

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hello World!');
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

JavaScript es **single thread**

¿Puede el server responder múltiples peticiones simultáneamente?

Implementación de Node.js se basa en
non-blocking I/O + event loop

una llamada non-blocking permite al JS runtime
continuar ejecución del stack de ejecución actual

al terminar una llamada non-blocking se agrega
código a ejecutar en una cola

cuando el stack de ejecución se vacía, el event loop
agrega al stack lo que esté primero en la cola

esto permite ejecución "paralela" y asíncrona

¿Más detalles y visualización del event loop?

¿Cómo sabe el JS runtime qué código ejecutar luego de una llamada asíncrona?

callbacks

```
function callback(data) {
  console.log(data);
}

getDataAsync(callback);
```

```
function getDataAsync(callback) {
  // get the data asynchronously
  // and then execute callback passing the data
  const data = 'Hello World!';
  callback(data);
}
```

Pero...

```
doAsync1(function (value1) {
  doAsync2(value1, function (value2) {
    doAsync3(value2, function (value3) {
      doAsync4(value3, function (value4) {
        console.log(value4);
      });
    });
  });
});
```

"Callback hell" o "Pyramid of doom"

Una promesa de rescate...

```
doAsync1()
  .then(function (value1) {
    return doAsync2(value1);
  })
  .then(function (value2) {
    return doAsync3(value2);
  })
  .then(function (value3) {
    return doAsync4(value3);
  })
  .then(function (value4) {
    console.log(value4);
  });
```

En lugar de recibir un callback, la función retorna una promesa por un valor, que será resuelta (o rechazada) en el futuro

La promesa tiene un método then, que nos permite agregar un callback a ejecutar cuando ésta se resuelva

```
const promiseForHello = getHelloAsync();

promiseForHello.then(function (hello) {
  console.log(hello);
});
```

... y el valor de retorno de then es, además, una promesa del valor que retorne el callback

```
const promiseForHello = getHelloAsync();

const promiseForHelloWorld =
promiseForHello.then(function (hello) {
  return hello + ' World!';
});

promiseForHelloWorld.then(function (message) {
  console.log(message);
});
```