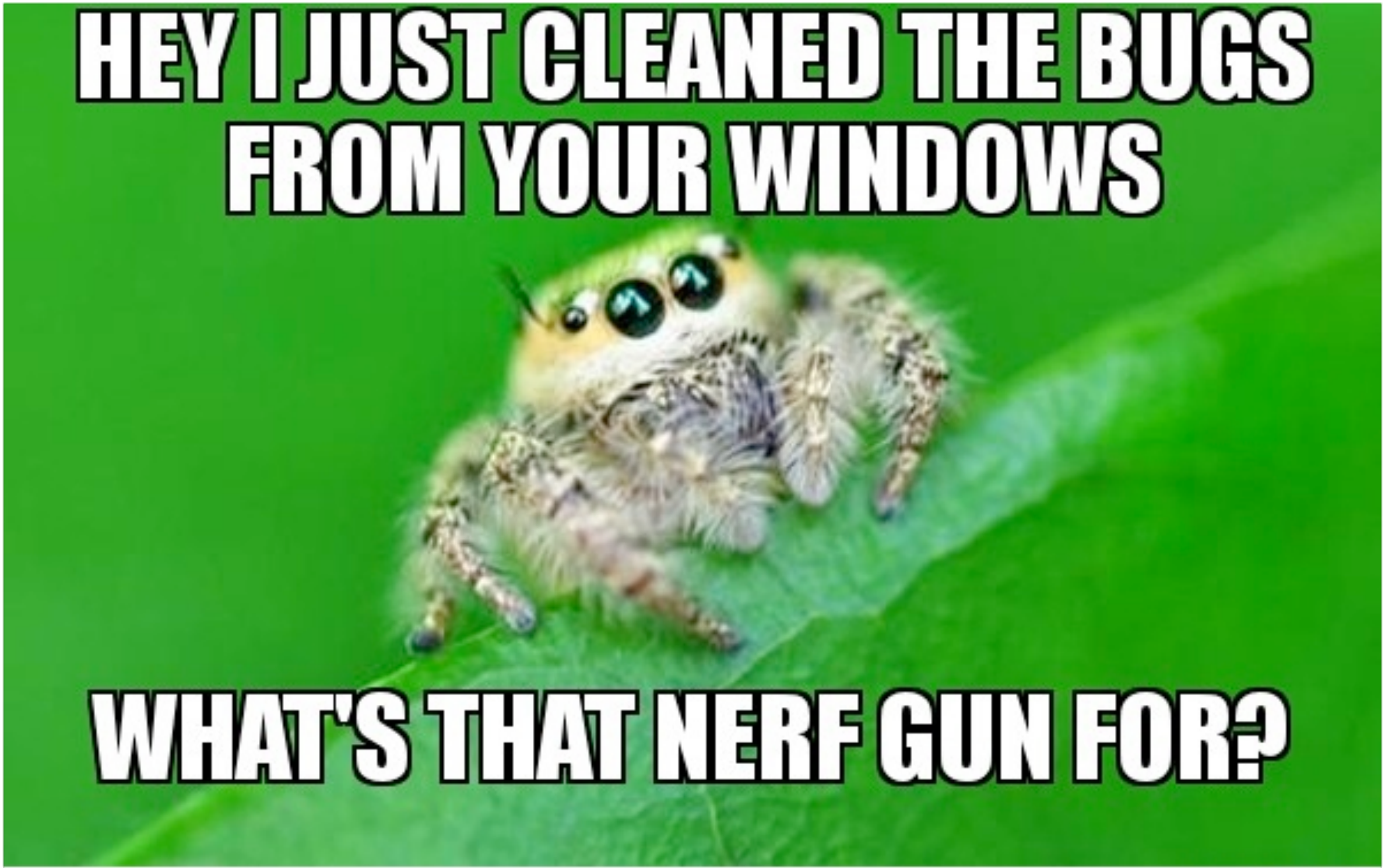
Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

JavaScript

... y un poco de Node.js

Raúl Montes T.


PEOPLE SAY I DON'T CARE...
BUT I DO.


HEY I JUST CLEANED THE BUGS FROM YOUR WINDOWS
WHAT'S THAT NERF GUN FOR?
WWW.IMGHUT.COM
FEEL IT - CREATE IT - SHARE IT
imgHUT


TRIED TO HELP SOMEONE STUCK IN A CAGE
GOT SHOT WITH A SPEAR GUN
WeKnowMemes

i smell blood i should rescue the injured - Misunderstood Shark

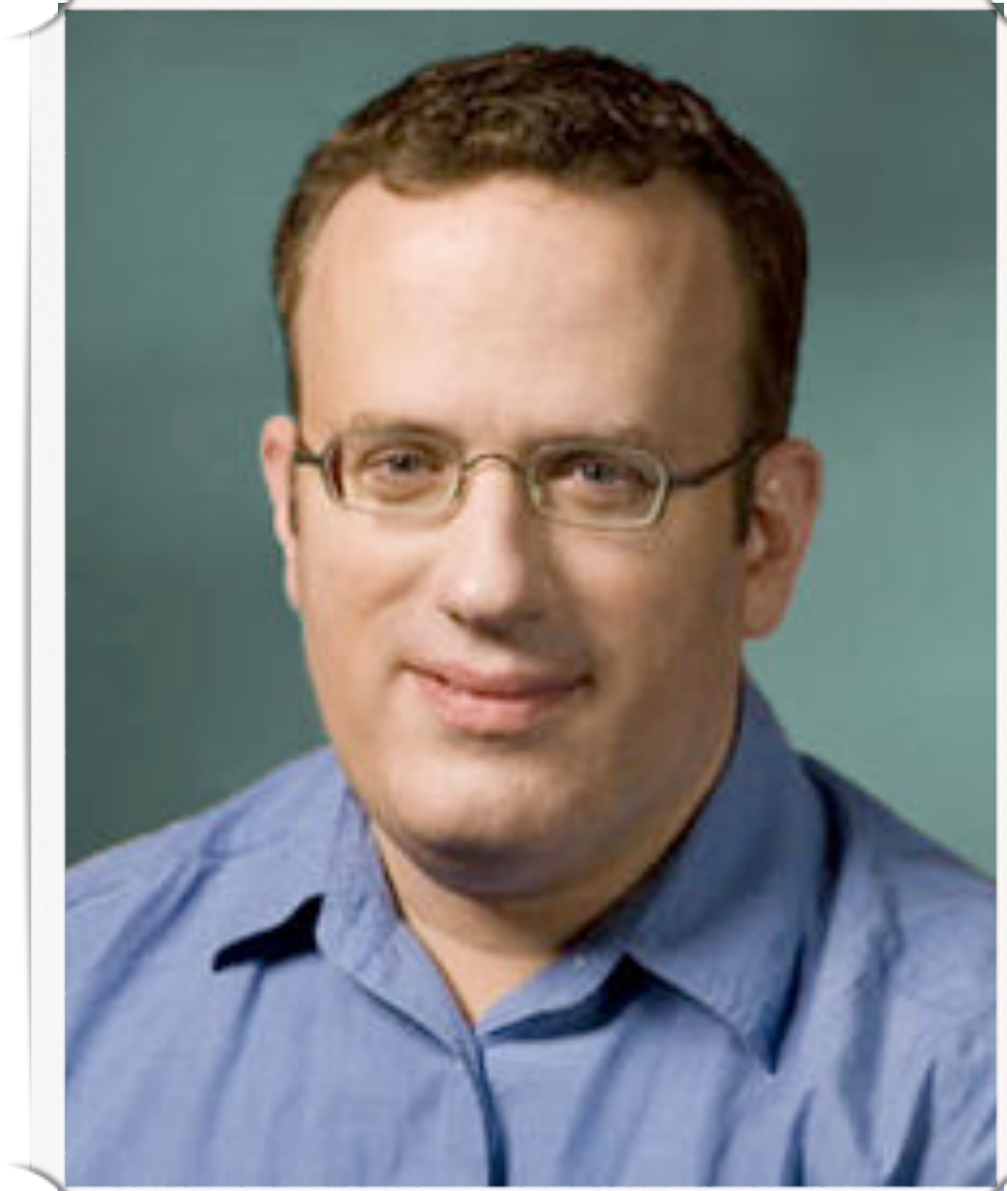

Everyone hates him during 7 movies
Turns out to be a good guy
MEMEBASE.com

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

ECMAScript 9th Edition, Junio 2018

...

¿Qué versión usamos?

Depende del contexto...

Browser ➜ ES5*

latest Node.js ➜ ~ES2017

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

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

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/');
});
```

JavaScript is a **dynamically** typed language

JavaScript is a **weakly** typed language

Weak Typing...

```
0 == ''                 // true
0 == '0'                // true
'' == '0'               // false
false == 'false'        // false
false == '0'            // true
false == undefined      // false
false == null           // false
null == undefined       // true
' \t\r\n ' == 0         // true
```

Recomendación: usar **===** y **!==** en lugar de == y !=

Si el tipo no es el mismo, retornan false de inmediato, por lo que son menos confusos y propensos a errores

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

Sino, pertenecen al Global Object

JavaScript es **multiparadigma**

object oriented

imperative/procedural

functional

JavaScript has First class functions

... y con closures

JavaScript is an Object Oriented language

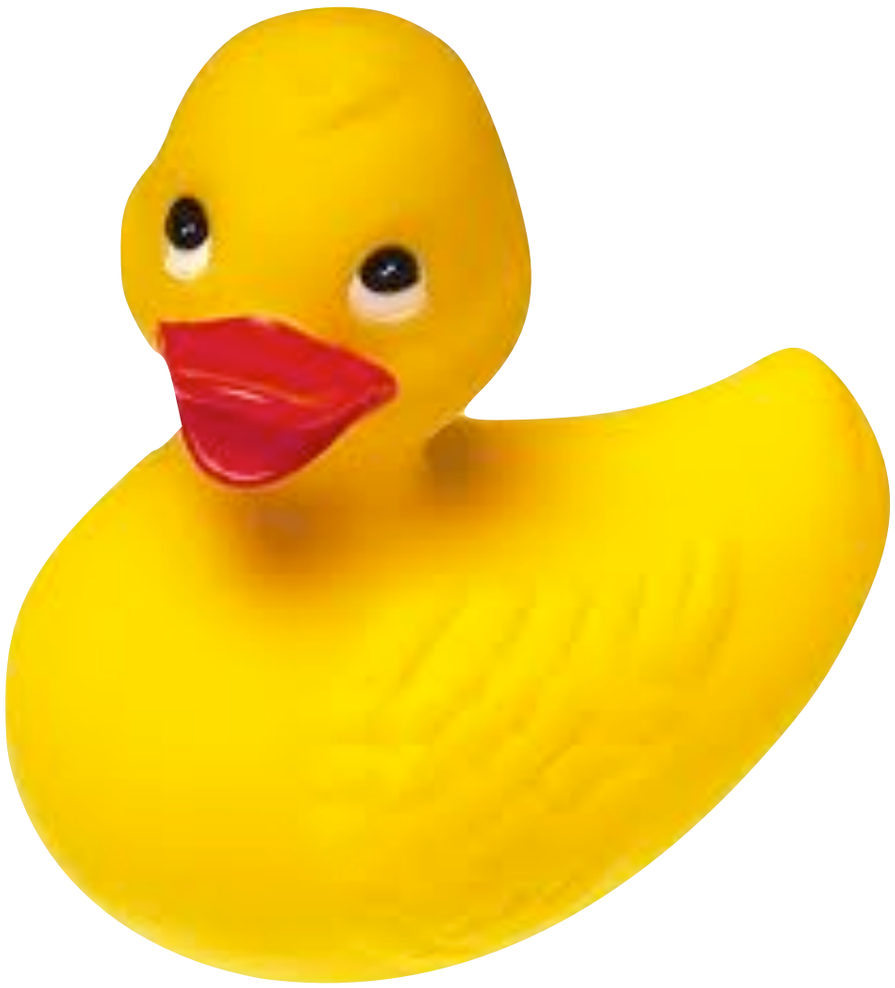
JavaScript doesn't have classes *

JavaScript has prototypes

* desde ES2015 existe *syntax sugar* para clases

¿Object Oriented sin clases?

Constructor functions + prototypal inheritance



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 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?

¡Pruébalo tú!

¿Cómo se sabe 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);
}
```

¿Cómo se sabe qué código ejecutar luego de una llamada asíncrona?

callbacks

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

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

¿Cómo se sabe qué código ejecutar luego de una llamada asíncrona?

callbacks

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

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

¿Cómo se sabe qué código ejecutar luego de una llamada asíncrona?

callbacks

```
getDataAsync((data) => {
  console.log(data);
});
```

```
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);
});
```

Más características de JS

Conversión rápida con operadores

```
+"100" -> 100
!!"100" -> boolean true
!!0 -> boolean false
```

If/else if/else y while

```
if(condicion) {
  // sentencias...
} else if(condicion 2) {
  // sentencias...
} else {
  // sentencias...
}

while(condicion) {
  // sentencias...
}
```

For y for..in

```
for(var i = 0; i < object.length; i++) {
  // sentencias...
}
for(var prop in object) {
  // en prop tendremos cada una de las
  // propiedades/atributos de object
}
```

OR y AND
como Default y Guard

```
var foo = bar && bar.length;
// si bar es interpretable a false no se pedirá el largo

foo = bar || "default";
// si bar es interpretable a false se usará el valor “default”

function ejemplo(param1) {
  var valorParam1 = param1 || "valor default para param1"
}
```

Hoisting

Declaraciones se interpretan primero

Equivale a siempre declarar las variables al principio del scope

```
console.log(foo);
var foo;
// equivalente a
var foo;
console.log(foo);
```

Functions

```
// ambas son equivalentes casi siempre
function ejemplo() {
  alert("Hola!");
}
var ejemplo = function() {
  alert("Hola!");
}
```

La diferencia es el efecto del hoisting

En el segundo caso sólo la declaración se interpreta primero. La asignación al ejecutar el programa

Function params

```
// - los params no identifican a la funcion
// - se pueden entregar más o menos params de los declarados
// - si uno no se entrega => valor undefined
// - si se entrega uno no declarado, se ignora
function ejemplo(prefijo, nombre) {
  alert("Hola, " + prefijo + " " + nombre + "!");
}

// se pueden obtener todos los params entregados con arguments
function ejemplo() {
  if (arguments.length > 1) {
    alert("Hola, " + arguments[0] + " " + arguments[1] + "!");
  } else {
    alert("Hola, " + arguments[0] + "!");
  }
  return arguments.length;
}
```

Arreglos

```
foo = []; // creamos un arreglo
foo[0] = 1;
foo[1] = 2;
foo[10] = "diez"; // se "rellenan" los espacios con undefined
foo.push("chao"); // agregado al final...pop saca del comienzo, como un stack
foo.splice(index, quantity); // elimina, desde index, quantity elementos
foo = new Array(); // otra forma de crear un arreglo
```

Objetos

```
foo = {}; // creamos un objeto
// son solo key/value pairs
// las keys pueden ser texto libre
foo = {uno: 1, dos: 2, "ciento ocho": 108}
// las keys se pueden acceder como en un arreglo asociativo
foo["uno"] -> 1
foo["ciento ocho"] -> 108
// o como "dot notation" para las keys simples
foo.uno -> 1
```

Métodos

```
// los métodos son sólo funciones asignadas
// como una de las propiedades de un objeto
foo.metodo = function() {
  alert("Método de instancia: " + this.uno);
}
// lo ejecutamos
// this será una referencia al objeto en el cual se llama
foo.metodo();
```

Otra forma de llamar funciones...

```
var f = function() {
  // cuerpo de la función
};

// "call" llama a la función y permite elegir el "this" que usará
// y además entregar los params como en una llamada normal
f.call(referenciaAThis, param1, param2, ...);
// "apply" es lo mismo, pero los params se entregan en un Array
f.apply(referenciaAThis, paramsArray);
```

Scope

```
// scope de función, no de bloque
function f() {
  var v1 = 1;
  if (v1 > 0) {
    var v2 = 2;
  }
  console.log(v2);
}
f();  // muestra 2, funciona
```

```
// el scope incluye el scope externo
function f1() {
  var v1 = 1;
  function f2() {
    console.log(v1);
  }
  f2();
}
f1();  // muestra 1
```

Closure

El **closure** de una función contiene todas las variables existentes en el scope en el cual fue declarada la función

```
function f1() {
  var v1 = 1;
  var f2 = function() {
    console.log(v1);
  };
  return f2;
}
var unaFuncion = f1();
unaFuncion(); -> muestra 1
```

La variable local v1 forma parte del closure de f2

Constructores

Cualquier función puede ser usada como **Constructor** de objetos. Tendrá a this como referencia al objeto que se está construyendo. Se crea el objeto con new

```
function Perro(nombre, raza) {
  this.nombre = nombre;
  this.raza = raza;
}
var perro = new Perro(“Snoopy", "Beagle");
```

Construir objetos con métodos

```
function Perro(nombre, raza) {
  this.nombre = nombre;
  this.raza = raza;
  this.habla = function(veces) {
    while (veces-- > 0) {
      console.log("Guau!");
    }
  }
}
var perro = new Perro(“Snoopy", "Beagle");
perro.habla(5);
```

Y esto?

```
var perro1 = new Perro("Pluto", "QuienSabe");
var perro2 = new Perro("Snoopy", "Beagle");

perro1.habla === perro2.habla -> false
// no son el mismo método... estamos creando la función cada vez
// un poco tonto o no?
```

Prototype

Todos los objetos (salvo el objeto base) tienen un **prototipo**.

El **prototipo** es... un objeto... que tiene prototipo...

Las propiedades de un objeto se buscarán en el objeto mismo y, si no están ahí, en su prototipo, y prototipo del prototipo y... => **herencia**!

Prototype

Todas las funciones, además de su prototipo como objeto, tienen un prototype para asignárselo a los objetos que construyen

```
var perro1 = new Perro("Pluto", "QuienSabe");
// para obtener el prototipo de un objeto
// sólo la segunda es 100% segura en browsers "no modernos" (sí, IE... :P)
perro.__proto__, perro.constructor.prototype, Object.getPrototypeOf(perro)
// para obtener el prototipo que una función asigna a sus objetos:
Perro.prototype
// este mismo prototipo es el que se asigna a los objetos
Perro.prototype === perro1.__proto__ -> true
```

Prototype

Entonces, si modificamos el prototype que se le asigna a los objetos...

```
Perro.prototype.habla = function(veces) {
  while(veces-- > 0) {
    console.log(this.nombre);
  }
}
perro1.habla === perro2.habla -> true
```

El objeto función es el mismo, pero dependiendo desde donde se llame, **this** será un diferente objeto :-)

Herencia con prototipos

```
function Mamifero(nombre) {
  this.nombre = nombre;
}
Mamifero.prototype.habla = function(veces) { console.log("..."); }
// subclase de Mamifero
function Perro(nombre, raza) {
  Mamifero.call(this, nombre); // llamamos al constructor de la super "clase"
  // (como llamar a un "super()"
  this.raza = raza;
}
// hacemos que el prototipo que asigne este constructor sea un objeto de la superclase
// así los Perros tendrán todos los métodos de los Mamíferos y además podremos
// agregar/sobreescribir otras propiedades
// Además asignamos Perro como el constructor del prototipo
Perro.prototype = new Mamifero();
Perro.prototype.constructor = Perro;
var perro = new Perro("Odie", "Teckel");
perro.habla === perro1.habla -> true
PerroEducado.prototype.habla = function(veces) {
  while(veces-- > 0) {
    console.log(this.nombre);
  }
};
perro.habla === mamifero.habla -> false
```

Atributos/Métodos privados, privilegiados y públicos

```
var Perro = function(nombre, raza) {
  // variables locales como nombre o raza, independiente
  // de si almacenan una referencia a un objeto cualquiera
  // o una función, serán propiedades de visibilidad privada, sólo
  // accesibles por quien tenga estas variables en su scope/closure

  // funciones definidas dentro del constructor pero
  // asignadas a una propiedad del objeto resultante (this),
  // serán de visibilidad "privilegiada" (de acceso público,
  // pero con acceso a propiedades privadas)
  this.getNombre = function() {return nombre;}

}

// propiedades definidas al objeto o a su prototipo pero fuera de la
// función constructora, serán de visibilidad pública, pero no tendrán
// acceso a propiedades privadas (pero sí a privilegiadas)
Perro.prototype.ladra = function(veces) {
  while(veces-- > 0) {
    console.log(this.getNombre() + ": Guau!");
  }
}
```