Breve historia

- **JAVASCRIPT (December 4th 1995)** Netscape and Sun press release
- **ECMAScript Standard Editions:**
  - **ES1 (June 1997)** Object-based, Scripting, Relaxed syntax, Prototypes
  - **ES2 (June 1998)** Editorial changes for ISO 16262
  - **ES3 (December 1999)** Regexps, Try/Catch, Do-While, String methods
  - **ES5 (December 2009)** Strict mode, JSON, .bind, Object mts, Array mts
  - **ES5.1 (June 2011)** Editorial changes for ISO 16262:2011
  - **ES6 (June 2015)** Classes, Modules, Arrow Fs, Generators, Const/Let, Destructuring, Template Literals, Promise, Proxy, Symbol, Reflect
  - **ES7 (June 2016)** Exponentiation operator (**) and Array Includes
  - **ES8 (June 2017)** Async Fs, Shared Memory & Atomics

ES6, ES2015

- Breve historia
  - standard de JS es ECMA-262
  - no cambió desde publicación de 3a edición en 1999 hasta año 2007
  - se propuso un borrador de ECMAScript 4 (muy ambicioso)
  - grupo se separó y propuso ECMAScript 3.1 (incremental)
  - en 2008 creador de JS (Brendan Eich) quiso ordenar el caos y parte esfuerzo de unir 3.1 y 4 (Harmony)
  - 3.1 se convierte en ECMAScript 5 (4 nunca fue liberado)
  - 6 es el primer release de la iniciativa Harmony y se completó el año 2014

Muchos Cambios

- Pequeños o básicos (soporte completo de unicode, nuevos métodos de strings, mejora en expresiones regulares, etc)
- block bindings (let, constant)
- funciones (parámetros por defecto, resto, block level, notación flecha, etc)
- objetos (setPrototype, super, methods, etc)
- clases
- símbolos (Ruby)
- promesas (incluyendo encadenamiento)
- módulos

Block Bindings

- En JS el scope es de función
- Declaración *var* deja visible la variable desde el comienzo

```
function checkScope() {
"use strict";
  var i = "function scope";
  if (true) {
    i = "block scope";
    console.log("Block scope i is: ", i);
  }
  console.log("Function scope i is: ", i);
  return i;
}

checkScope();
```

CONSOLE

> Block scope i is: ,"block scope"
> Function scope i is: ,"block scope"

```
function checkScope() {
"use strict";
  let i = "function scope";
  if (true) {
    let i = "block scope";
    console.log("Block scope i is: ", i);
  }
  console.log("Function scope i is: ", i);
  return i;
}

checkScope();
```

CONSOLE

> Block scope i is: ,"block scope"
> Function scope i is: ,"function scope"

y esto ?

```
function checkScope() {
"use strict";
  // let i = "function scope";
  if (true) {
    var i = "block scope";
    console.log("Block scope i is: ", i);
  }
  console.log("Function scope i is: ", i);
  return i;
}

checkScope();
```

CONSOLE

> Block scope i is: ,"block scope"
> Function scope i is: ,"block scope"

```
function checkScope() {
"use strict";
  //let i = "function scope";
  if (true) {
    let i = "block scope";
    console.log("Block scope i is: ", i);
  }
  console.log("Function scope i is: ", i);
  return i;
}

checkScope();
```

CONSOLE

> Block scope i is: ,"block scope"
> ReferenceError: i is not defined (/index.js:11)

Const es similar a un let pero read-only

```
function printManyTimes(str) {
  "use strict";

  var sentence = str + " is cool!";

  sentence = str + " is amazing!"

  for(var i = 0; i < str.length; i+=2) {
    console.log(sentence);
  }

}
printManyTimes("freeCodeCamp");
```

CONSOLE

```
> freeCodeCamp is amazing !
> freeCodeCamp is amazing !
> freeCodeCamp is amazing !
> freeCodeCamp is amazing !
> freeCodeCamp is amazing !
> freeCodeCamp is amazing !
```

```
function printManyTimes(str) {
  "use strict";

  const sentence = str + " is cool!";

  sentence = str + " is amazing!"

  for(var i = 0; i < str.length; i+=2) {
    console.log(sentence);
  }

}
printManyTimes("freeCodeCamp");
```

CONSOLE

```
> Error: SyntaxError: unknown: "sentence" is read-only (/index.js:1)
```

y ahora ?

```
function printManyTimes(str) {
  "use strict";

  const sentence = str + " is cool!";

  for(let i = 0; i < str.length; i+=2) {
    console.log(sentence);
  }

}
printManyTimes("freeCodeCamp");
```

CONSOLE

```
> freeCodeCamp is cool !
> freeCodeCamp is cool !
> freeCodeCamp is cool !
> freeCodeCamp is cool !
> freeCodeCamp is cool !
> freeCodeCamp is cool !
```

Arrays y Objects son mutables

```
const s = [5, 7, 2];
function editInPlace() {
  "use strict";

  //s = [2, 5, 7];
  s[0] = 2;
  s[1] = 5;
  s[2] = 7;

}
editInPlace();

console.log(s)
```

CONSOLE

> [2, 5, 7]

```
function freezeObj() {
  "use strict";
  const MATH_CONSTANTS = {
    PI: 3.14
  };

  try {
    MATH_CONSTANTS.PI = 99;
  } catch( ex ) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}

const PI = freezeObj();

console.log(PI);
```

CONSOLE

> 99

Pero se puede congelar un objeto con freeze

```
function freezeObj() {
  "use strict";
  const MATH_CONSTANTS = {
    PI: 3.14
  };

  Object.freeze(MATH_CONSTANTS);

  try {
    MATH_CONSTANTS.PI = 99;
  } catch( ex ) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}

const PI = freezeObj();

console.log(PI);
```

CONSOLE

```
> TypeError: Cannot assign to read only property 'PI' of object '#<Object>' (/index.js:13)
> 3.14
```

Asignación Desestructural

- Ayuda a sacar fácilmente elementos desde datos estructurados (objetos o arrays)

```
let obj = { a: [{ foo: 123, bar: 'abc' }, {}], b: true };
let { a: [{foo: f}] } = obj;     // f = 123

let { x: x } = { x: 7, y: 3 };   // x = 7
let [x,y] = ['a', 'b', 'c'];     // x='a'; y='b';

let [x=3, y] = [];               // x = 3; y = undefined
let {foo: x=3, bar: y} = {}; // x = 3; y = undefined
```

Funciones

- default parameters
- rest parameters
- destructured parameters
- funciones dentro de blocks
- funciones arrow

Default Parameters

- En JS siempre se ha podido pasar un número variable de parámetros (arguments.length)
- En ECMAScript 6 se pueden inicializar)

```
function foo(x, y) {
    x = x || 0;
    y = y || 0;
    ...
}
```



```
function foo(x=0, y=0) {
    ...
}
```

```
function selectEntries(options) {
    options = options || {};
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
    ...
}
```

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
 ...
}
```

Rest Parameters

- ... antes del nombre hace que ese nombre sea un array conteniendo el resto de los parámetros

```
function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
```

```
function logAllArguments(...args) {
    for (let arg of args) {
        console.log(arg);
    }
}
```

El operador ... tambien se usa para separar arrays

- Se llama operador spread

```
> Math.max(-1, 5, 11, 3)
11
> Math.max(...[-1, 5, 11, 3])
11
> Math.max(-1, ...[-1, 5, 11], 3)
11
```

Block-level functions

- En ECMAScript 5 es un error declarar una función en un block
- En ECMAScript 6 se permiten
  - similar a let en que desaparecen al salir del block
  - distinta a let en que valen desde comienzo del block (hoisted)
- Las clases no son elevadas (non hoisted)

Ejemplo

```
{ // Enter a new scope

    console.log(foo());       // OK, due to hoisting
    function foo() {
        return 'hello';
    }
}
```

Arrow functions

- sintaxis simplificada
- comportamiento algo distinto también
  - binding léxico de this (donde se define la función no donde se usa)
  - no se puede hacer new de estas funciones (no tienen constructor)
  - no se puede cambiar this al interior de la función
  - no se puede usar arguments

Ejemplos de nueva sintaxis

```
function foo(x,y) {
    return x + y;
}
```

Se convierte en

```
const   foo = (x, y) => x + y;
```

Otros ejemplos:

```
const  f1 = () =>12;

const  f2 = x => x * 2;

const  f3 = (x, y) => {
    let z = x*2 + y;
    y++;
    x*=3;
    return(x+y+z)/2;
};
```

No solo sintaxis

- la notación de funciones con flechas cambia el objeto this a un scope léxico en vez de dinámico

ES5

```
var controller = {
    makeRequest: function(..) {
        var self = this;
        btn.addEventListener( "click", function(){
            // ..
            self.makeRequest(..);
        }, false );
    }
};
```

ES6

```
var controller = {
    makeRequest: function(..) {
        btn.addEventListener( "click",  () => {
            // ..
            this.makeRequest(..);
        }, false );
    }
};
```

Object literals

- cuando el nombre de una propiedad es igual al de la variable local puede omitirse

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}
```

```
function createPerson(name, age) {
    return {
        name,
        age
    };
}
```

- métodos pueden declararse como funciones

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

```
var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
    }
};
```

Métodos

- Anteriormente solo una propiedad que contiene una función
- En ECMAScript 6 método queda asociado a un objeto (propiedad HomeObject)

```
var obj = {
    foo: function () {
        ...
    },
    bar: function () {
        this.foo();
    }, // trailing comma is legal in ES5
}
```

```
let obj = {
    foo() {
        ...
    },
    bar() {
        this.foo();
    },
}
```

Clases

- muchos desarrolladores creen que no son necesarias pero
- muchas librerías las implementan
- ECMAScript 6 incorpora clases aunque son algo distintas a lo usual

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called '+this.name;
};
```

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called '+this.name;
    }
}
```

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

Nuevos métodos de Array

- Array.from permite convertir listas (por ejemplo resultado de operaciones del DOM a arrays)

```
let lis = document.querySelectorAll('ul.fancy li');
Array.from(lis).forEach(function (li) {
    console.log(node);
});
```

- Array.of permite generar un array a partir de una lista de parámetros

`Array.of(item_0, item_1, ···)` creates an Array whose elements are `item_0`, `item_1`, etc.

Maps

```
> let map = new Map(); // create an empty Map
> const KEY = {};

> map.set(KEY, 123);
> map.get(KEY)
123
> map.has(KEY)
true
> map.delete(KEY);
true
> map.has(KEY)
false
```

```
let map = new Map([
    [ 1, 'one' ],
    [ 2, 'two' ],
    [ 3, 'three' ], // trailing comma is ignored
]);
```

Sets

```
let s = new Set()
s.add("hello").add("goodbye").add("hello")
s.size === 2
s.has("hello") === true
```

Módulos en ES6

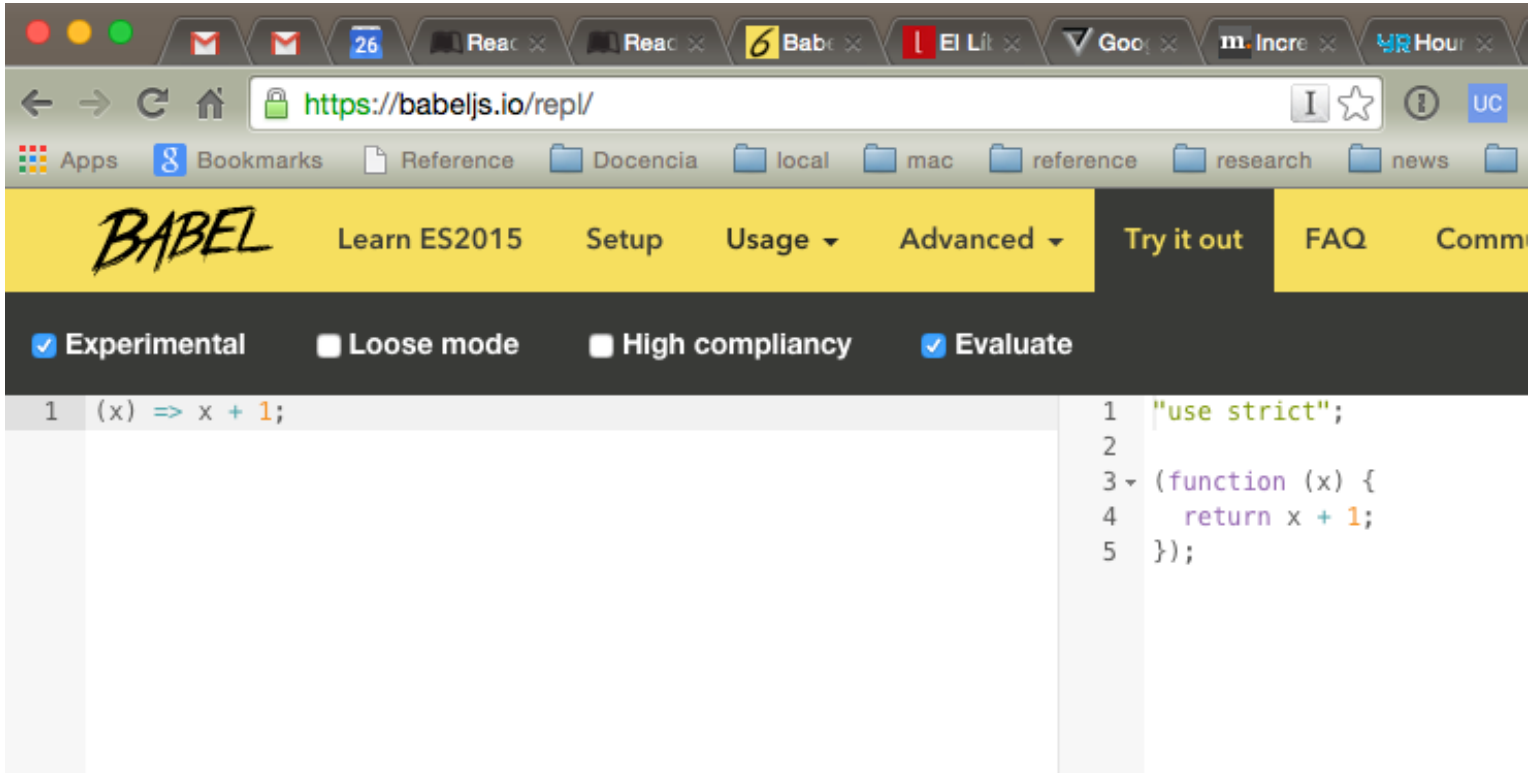
```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}
```

```
//------ main1.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

```
//------ main2.js ------
import * as lib from 'lib'; // (A)
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

Babel

- JavaScript to JavaScript compiler
- Permite usar ES6 con toda confianza
- Tiene un área para probar



Programación Asincrónica

- JS Engine - single thread event loop
- Solo ejecuta una piezas de código a la vez
- El código a ejecutar se pone en la cola de eventos

```
let button = document.getElementById("my-btn");
button.onclick = function(event) {
    console.log("Clicked");
};
```

- En el ejemplo, cuando se pincha el botón la función asignada a onclick se agrega a la cola
- A veces se requiere usar "callbacks"

Callbacks

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    console.log(contents);
});
console.log("Hi!");
```

- readFile () ejecuta de inmediato y pausa mientras se lee el archivo
- "Hi!" aparece de inmediato
- Función es ejecutada cuando se completa la lectura del archivo
- Finalmente se despliega el contenido

Callback Hell

```
method1(function(err, result) {

    if (err) {
        throw err;
    }

    method2(function(err, result) {

        if (err) {
            throw err;
        }

        method3(function(err, result) {

            if (err) {
                throw err;
            }

            method4(function(err, result) {

                if (err) {
                    throw err;
                }

                method5(result);
            });

        });

    });

});
```

Promesas (Promises)

- Un slot reservado para el resultado de una operación asíncrona
- La operación retorna una promesa
- Promesa parte en estado "pending" y luego de completar la operación queda "fulfilled" o "rejected"
- Método then permite tomar acción en cada caso

Ejemplo

```
let cleanRoom = function() {
  return new Promise(function(resolve, reject) {
    resolve('cleaned the room');
  });
};

let removeGarbage = function(message) {
  return new Promise(function(resolve, reject) {
    resolve(message + ' remove garbage');
  });
};

let winIcecream = function(message) {
  return new Promise(function(resolve, reject) {
    resolve( message + ' won icecream');
  });
};

cleanRoom().then(function(result){
        return removeGarbage(result);
}).then(function(result){
        return winIcecream(result);
}).then(function(result){
        console.log('finished ' + result);
})
```

CONSOLE

> finished cleaned the room remove garbage won icecream

Funciones Async (ES2017)

- Construido sobre las promesas
- Hace más claro el código cuando hay encadenamiento de promesas

```
addthis(5).then((val) => {
    return addthis(val*2)
}).then((val) => {
    return addthis(val/4)
}).then((val) => {
    return Promise.resolve(val) // return promise that resolves to final number
})
```

```
var a = await addthis(5)
var b = await addthis(a*2)
var c = await addthis(b/4)
return c // return promise that resolves to final number
```

¿ Como exactamente ?

- Se declara la función explícitamente como asíncrona

```
async function addAsync(x) {
  // code here...
}
```

- Se usa await para pausar el código hasta que la promesa se resuelva

```
async function addAsync(x) {
  const a = await doubleAfter2Seconds(10);
  const b = await doubleAfter2Seconds(20);
  const c = await doubleAfter2Seconds(30);
  return x + a + b + c;
}
```

ES2016 no agrega mucho a ES2015

- Array includes (reemplaza a array.indexOf)
- Ejemplos

```
[1, 2, 3].includes(2)  -->  true
[1, 2, 3].includes(4)  -->  false
[1, 2, NaN].includes(NaN)  -->  true
["a", "b", "c"].includes("a")  -->  true
["a", "b", "c"].includes("a", 1)  -->  false
```

- Operador de Exponenciación
  - x ** y equivalente a Math.pow(x, y)

Qué mas en ES2017 ?

- Object Entries y Object Values
  - Facilita iterar con objetos

```
Object.entries( {one: 1, two: 2} )      // [ ['one', 1], ['two', 2]  ]
Object.values( {one: 1, two: 2} )       // [1, 2]
```

  - String Padding
    - Se agregan métodos padStart y padEnd

```
'x'.padStart(5, 'ab')     // 'ababx'
'x'.padEnd(5, 'ab')     // 'xabab'
```

que mas ...

- shared array buffer y atomics (service workers)
- Object.getOwnPropertyDescriptors
  - retorna todas las propiedades definidas directamente para el objeto (no heredadas)

```
const obj = {
  get es7() { return 777; },
  get es8() { return 888; }
};
Object.getOwnPropertyDescriptors(obj);
// {
//   es7: {
//     configurable: true,
//     enumerable: true,
//     get: function es7(){}, //the getter function
//     set: undefined
//   },
//   es8: {
//     configurable: true,
//     enumerable: true,
//     get: function es8(){}, //the getter function
//     set: undefined
//   }
// }
```

ES2018

- async iteration
- for-await-of (versión async de loop for-off)
- async generators
- operador rest en desestructuración de objetos
- operador spread en objetos literales
- muchas cosas en expresiones regulares

ES2019 (Enero)

- nuevos métodos de Array: flat y flatMap
- nuevo método de Object: fromEntries
- nuevos métodos de String: trimStart, trimEnd
- Function.toString

Para seguirle la pista ...

- Repositorio del comité técnico 39 TC39
- https://github.com/tc39/proposals
  - Finished Proposals

RegExp Unicode Property Escapes	Mathias Bynens	Brian Terlson Daniel Ehrenberg Mathias Bynens	January 2018	2018
`Promise.prototype.finally`	Jordan Harband	Jordan Harband	January 2018	2018
Asynchronous Iteration	Domenic Denicola	Domenic Denicola	January 2018	2018
Optional `catch` binding	Michael Ficarra	Michael Ficarra	May 2018	2019
JSON superset	Richard Gibson	Mark Miller Mathias Bynens	May 2018	2019
`Symbol.prototype.description`	Michael Ficarra	Michael Ficarra	November 2018	2019
`Function.prototype.toString` revision	Michael Ficarra	Michael Ficarra	November 2018	2019
`Object.fromEntries`	Darien Maillet Valentine	Jordan Harband Kevin Gibbons	January 2019	2019
Well-formed `JSON.stringify`	Richard Gibson	Mathias Bynens	January 2019	2019
`String.prototype.{trimStart,trimEnd}`	Sebastian Markbåge	Sebastian Markbåge Mathias Bynens	January 2019	2019
`Array.prototype.{flat,flatMap}`	Brian Terlson Michael Ficarra	Brian Terlson Michael	January 2019	2019