ES6

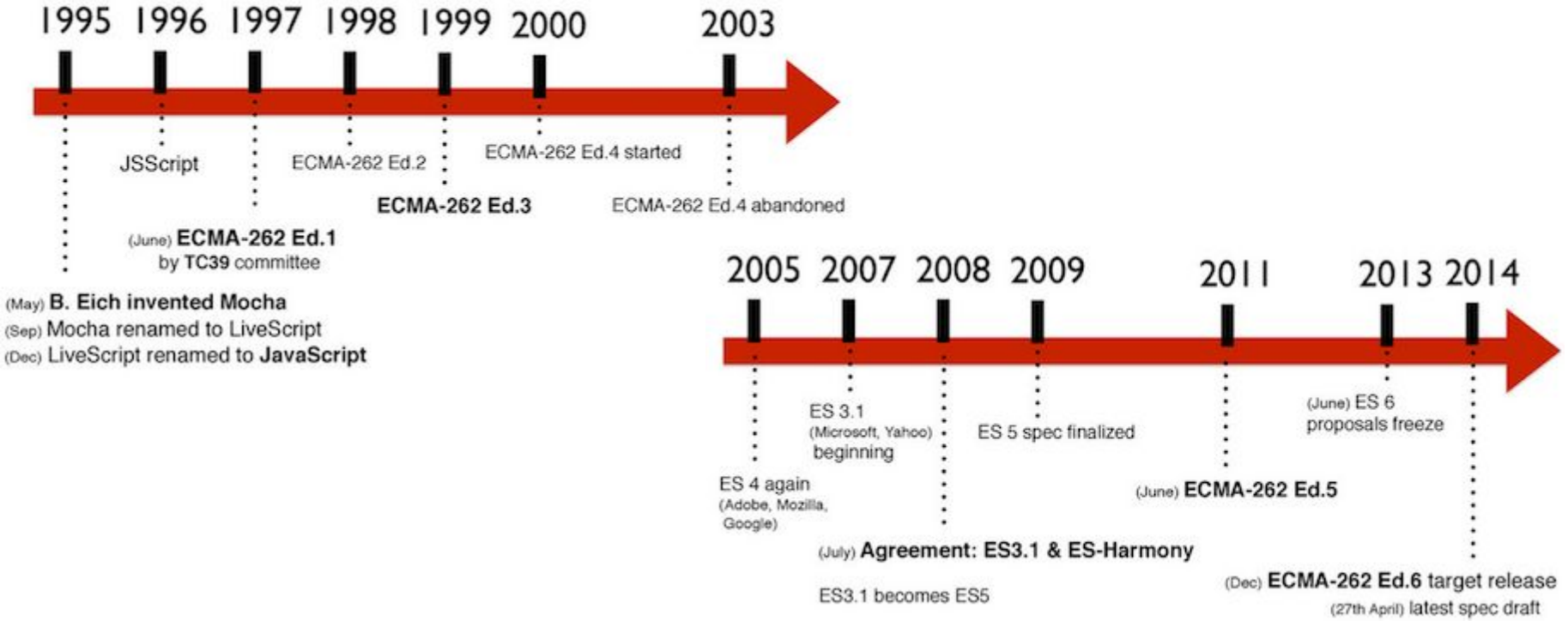
ECMAScript 2015

Programa:

- Breve historia de JavaScript
- Introducción a ES6
  - Variables var, const y let.
  - Templates de strings
  - Condicionales reducidas
  - Arrays (foreach, find, from, map, reduce, filter)
  - Funciones arrow
  - Destructuring
  - Valores por default, parametros rest
  - Spread operator (arrays y variables)
  - Clases y Herencias
  - Modules (export, import)
  - Promises

JavaScript

Breve historia



ES6

Navegadores

Safari 10 y Edge 14 fueron los primeros navegadores en dar soporte total a ES6.

				
Chrome 58	Edge 14	Firefox 54	Safari 10	Opera 55
Jan 2017	Aug 2016	Mar 2017	Jul 2016	Aug 2018

Podemos programar con ES6 y aun así utilizar navegadores que no lo soportan como Internet Explorer?

const y let

```
const PI;
PI = 3.1416;
// ERROR, porque el valor se debe asignar en la
// declaración

const PI = 3.15;
PI = 3.14159;
// ERROR de nuevo, porque es READ-ONLY
```

Podemos crear constantes que son READ-ONLY a lo largo del código y variables que no son accesibles más allá de su alcance.

```
//ES5
if(true) {
   var x = "hola mundo";
}
console.log(x);  // Imprime "hola mundo", porque //
"var" hace que sea global a la función;

//ES6
if(true) {
   let x = "hola mundo";
}
console.log(x);
//Da error, porque "x" ha sido definida dentro del
// "if"
```

Strings

```
//ES5
var nombre1 = "JavaScript";
var nombre2 = "genial";
console.log("Sólo quiero decir que " + nombre1 + "
es " + nombre2);
// Solo quiero decir que JavaScript es genial
```

```
//ES6
let nombre1 = "JavaScript";
let nombre2 = "genial";
console.log(`Sólo quiero decir que ${nombre1} es
${nombre2}`);
// Solo quiero decir que JavaScript es genial
```

Podemos interpolar strings de una manera más sencilla.

Condicionales reducidos

———

condición ? expr1 : expr2

```
//ES5
var cuota = "La Cuota es de:  ";
if(isMember){
   cuota += "$2.00";
}
else{
   cuota += "$10.00";
}
```

```
//ES6
const cuota = `La Cuota es de:  ${isMember ? "$2.00" :
"$10.00"}`;

//Otros ejemplos

const elvisLives = Math.PI > 4 ? "Sip" : "Nop";

//asignar valores por defecto
function savePerson (data){
   const person = {
       name: data.name,
       age: data.age,
       email: data.email ? data.email :
"default@gmail.com"
   }
   db.save(person)
}
```

Arrow functions

Proporcionan una sintaxis más compacta para la definición de funciones. Además conserva el valor de **this**

```
// ES5
setInterval(function () {
   console.log('hola mundo');
}, 100);

// ES6
setInterval(() => {
   console.log('hola mundo')
}, 100);

// ES5
var sum = function(a, b){
   return a+b;
}
// ES6
var sum = (a, b) => a + b;
```

```
// ES5
// data es un array de objetos
var data = [{...}, {...}, {...}, ...];
data.forEach(function(elem){
    // Tratamos el elemento
   console.log(elem)
});

// ES6
let data = [{...}, {...}, {...}, ...];
data.forEach(elem => {
    console.log(elem);
});
```

Arrays foreach, find

```
let array1 = ['a', 'b', 'c'];

array1.forEach(function(element) {
 console.log(element);
});

// expected output: "a"
// expected output: "b"
// expected output: "c"
```

Este método ejecuta la función indicada, una vez por cada elemento del array.

```
const inventory = [
   {name: 'apples', quantity: 2},
   {name: 'bananas', quantity: 0},
   {name: 'cherries', quantity: 5}
];
console.log(inventory.find(fruit => fruit.name ===
'cherries'));
// { name: 'cherries', quantity: 5 }
```

Este método tiene como parámetro una función que retorna el primer elemento que evalúa como verdadero.

Arrays from, map

`Array.from(arrayLike[, mapFn[, thisArg]])`

arrayLike: Objeto iterable para convertirlo en un array.

mapFn (Optional) : Función de mapa para llamar a cada elemento de la matriz.

thisArg (Optional): Valor para usar como this al ejecutar mapFn.

```
console.log(Array.from('foo'));
// expected output: Array ["f", "o", "o"]


console.log(Array.from([1, 2, 3], x => x + x));
// expected output: Array [2, 4, 6]
```

El método **map()** crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
  return x * 2;
});
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]


var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

Arrays reduce, filter

El método **reduce()** aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor.

```
var total = [0, 1, 2, 3].reduce(function(a, b){ return
a + b; });
// total == 6

var integrado = [[0,1], [2,3],
[4,5]].reduce(function(a,b) {
   return a.concat(b);
 });
 // integrado es [0, 1, 2, 3, 4, 5]
```

El método **filter()** crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

```
var words = ['spray', 'limit', 'elite', 'exuberant',
'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction",
"present"]
```

Destructuring

```
const [a, b] = ["hola", "mundo"];
console.log(a); // "hola"
console.log(b); // "mundo"

const obj = { nombre: "Carlos", apellido:
"Gonzalez" };
const { nombre } = obj;
console.log(nombre); // "Carlos"
```

```
let foo = () => {
    return ["175", "75"];
};

const [estatura, peso] = foo();
console.log(estatura); //175
console.log(peso); //75
```

Es una de las características nuevas más poderosas, nos facilita las cosas y hace el código más legible.

Valores por defecto

```
function f (x, y = 7, z = 42) {
   return x + y + z
}
f(1) === 50
f(1, 1) === 44
f(1, 1, 1) === 3
```

Por fin podemos incluir valores por defecto en nuestros parámetros, como en otros lenguajes de programación.

Parámetros rest

Nos proporcionan una manera de pasar un conjunto indeterminado de parámetros que la función agrupa en forma de Array. Solo puede ser parámetro rest el último argumento de la función.

```
// ES5
function findMax() {
   var i;
   var max = -Infinity;
   for (i = 0; i < arguments.length; i++) {
       if (arguments[i] > max) {
           max = arguments[i];
       }
   }
   return max;
}

console.log(findMax(1, 123, 500, 115, 44, 88));
// "500"
```

```
// ES6
function findMax(...numbers) {
   let max = Number.MIN_SAFE_INTEGER;
   numbers.forEach(n => {
       if (n > max) {
           max = n;
       }
   })
   return max;
}

console.log('max value ' + findMax(1, 123, 500, 115, 44, 88));
// "500"
```

Spread operators (operadores de distribución o propagación)

```
const codeburst = 'CODEBURST';
const characters = [ ...codeburst ];
console.log(characters)
// [ 'C', 'O', 'D', 'E', 'B', 'U', 'R', 'S',
'T' ]

const items = ['This', 'is', 'a',
'sentence'];
console.log(items) // [ 'This', 'is', 'a',
'sentence' ]
console.log(...items) // This is a sentence
```

```
// ES5
Math.max(5, 6, 8, 9, 11, 999);
// 999


// ES6
const numbers = [5, 6, 8, 9, 11, 999];
Math.max(...numbers)
```

Clases

Podemos definir clases y herencia de forma más clara y explícita

```
// ES5
//Clase
function Document(title, author, isPublished) {
   this.title = title;
   this.author = author;
   this.isPublished = isPublished;
 }
 Document.prototype.publish = function publish() {
   this.isPublished = true;
 };
 //Herencia
 function Book(title, author, topic) {
   Document.call(this, title, author, true);
   this.topic = topic;
 }
 Book.prototype = Object.create(Document.prototype);
```

```
// ES6
//Clase
class Document {
 constructor(title, author, isPublished) {
   this.title = title;
   this.author = author;
   this.isPublished = isPublished;
 }
 publish(){
       this.isPublished = true;
   }
}

//Herencia
class Book extends Document{
 constructor(title, author, topic){
   super(title, author, true);
   this.topic = topic;
 }
}
```

Módulos

```
//  lib/math.js
export function sum (x, y) { return x + y }
export const pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi,
math.pi))

//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

```
//  lib/mathplusplus.js
export * from "lib/math"
export var e = 2.71828182846
export default (x) => Math.exp(x)

//  someApp.js
import exp, { pi, e} from "lib/mathplusplus"
console.log("e^{π} = " + exp(pi))
```

ES6 incluye la funcionalidad de módulos, que nos permite exportar/importar objetos, funciones y clases desde código, sin tener que importarlos desde HTML.

Ejercicio: cómo lo escribirías en ES6?

---

1. Crear un módulo que tenga una función que sume todos sus argumentos y retorna la suma total y la cantidad de argumentos que tuvo (`{sum: 10, length: 2}`). Importar el módulo creado para sumar los números del 1 al 10.

Promises

Usos

1. Las promesas se utilizan para el manejo asíncrono de eventos.
2. Las promesas se utilizan para manejar solicitudes http asíncronas.

Beneficios de las promesas

- Mejora la legibilidad del código
- Mejor manejo de las operaciones asíncronas.
- Mejor flujo de definición de control en lógica asíncrona.
- Mejor manejo de errores

Promises

Estados de una promise:

fulfilled: Acción relacionada con la promesa realizada.

rejected: Falló la acción relacionada con la promesa

pending: la promesa aún está pendiente, es decir, aún no se ha cumplido o rechazado

settled: la promesa ha cumplido o rechazado

Constructor

```
var promise = new Promise(function(resolve, reject){
   //do something
});
```

Parámetros

El constructor de promesas toma solo un argumento, una función de callback.

El callback toma dos argumentos, `resolve` y `reject`

Promises

Al realizar operaciones dentro del callback si todo salió bien, entonces se llama a `resolve`.

Si las operaciones deseadas no van bien, entonces se llama a `reject`.

```
let promise = new Promise(function(resolve, reject) {
   const x = "thisisapromise";
   const y = "thisisapromise"
   if(x === y) {
     resolve();
   } else {
     reject();
   }
 });

 promise.
     then(function () {
         console.log('Success, You are a promise expert!');
     }).
     catch(function () {
         console.log('Some error has occured');
     });
//Success, You are a promise expert!
```

Promises consumers: then

Las promesas se pueden consumir registrando funciones usando los métodos **.then** y **.catch**.

then(): se invoca cuando una promesa se resuelve o se rechaza.

Toma dos funciones como parámetros:

- La primera función se ejecuta si la promesa se resuelve y se recibe un resultado.
- La segunda función se ejecuta si se rechaza la promesa y se recibe un error. (Es opcional y hay una mejor manera de eliminar el error usando el método .catch ()

```
promise.then(function(result){
   //handle success
}, function(error){
   //handle error
})
```

Promises consumers: then

```
let promise = new Promise(function (resolve, reject) {
   const random = Math.random() * 10; // returns a
random integer from 0 to 9
   if (random % 2) {
       resolve("SUCCESS!! Is odd number");
   } else {
       reject("ERROR: Is even number");
   }
});
```

```
promise
   .then(function (successMessage) {
       //success handler function is invoked
       console.log(successMessage);
   }, function (errorMessage) {
       console.log(errorMessage);
   })
```

Promises consumers: catch

---

catch() se invoca cuando se rechaza una promesa o se produce algún error en la ejecución.

Toma una función como parámetro. Esta función se utiliza para manejar errores o prometer rechazos. El método (**.catch()** llama internamente a **.then(null, errorHandler)**, es decir, **.catch()** es solo una abreviación de **.then(null, errorHandler))**

```
let promise = new Promise(function(resolve, reject) {
   reject('Promise Rejected')
})
 promise
   .then(function(successMessage) {
       console.log(successMessage);
   })
   .catch(function(errorMessage) {
      //error handler function is invoked
       console.log(errorMessage);
   });

//Promise Rejected
```

Consumir API rest

Fetch

```
fetch(url)
 .then(response => response.json())
 .then(data => console.log(data));
```

Proporciona una forma lógica y sencilla de consumir rest APIs que implica un proceso de **dos pasos** cuando se manejan datos JSON. Lo primero es hacer la solicitud real y luego lo segundo es llamar al método .json () en la respuesta.

Axios

```
axios.get(url).then(response => console.log(response));
axios
 .post("/user", {
   firstName: "Fred",
   lastName: "Flintstone"
 })
 .then(function(response) {
   console.log(response);
 })
 .catch(function(error) {
   console.log(error);
 });
```

Axios permite eliminar el paso intermedio de pasar los resultados de la solicitud http al método .json (). Simplemente devuelve el objeto de datos que esperamos.

https://www.npmjs.com/package/axios

Más información:

– – –

- http://www.ecma-international.org/
- https://babeljs.io/
- http://es6-features.org
- http://kangax.github.io/compat-table/es6/
- https://www.w3schools.com/js/js_es6.asp
- https://www.npmjs.com/package/axios

Consultas?