6

Javascript
Conceptos avanzados

Desarrollo de aplicaciones web con React JS

fundación esplai
ciudadanía comprometida

Var, Let, Const

var, let y const

var

- Las variables declaradas con var son "elevadas" (hoisted) al principio de la función o del proceso global.
- El scope de una variable declarada con var dentro de una función, es su contexto de ejecución.
- El scope de una variable declarada fuera de la función es global.

```
function z(){
  console.log(i); // ok, undefined
  console.log(x); // error!

  var i = 1;
  console.log("antes: " + i); //1

  for (var i=0; i<10;i++){
    // cualquier cosa...
  }
  console.log("después: " + i); //10

}

z();
```

var, let y const

let

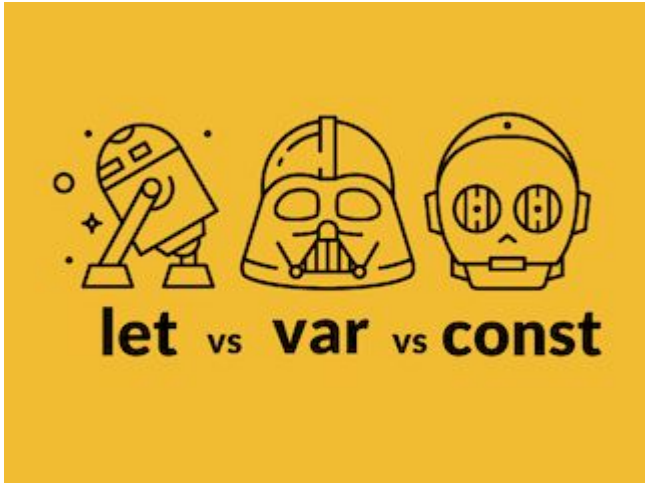
- Una variable no declarada siempre dará error
- El scope de una variable declarada con let es el bloque de código en el que está incluida.
- Si se declara fuera de una función, será global.
- Desde ES6, podemos prescindir de var, las variables declaradas con let son más predecibles...

```
function z(){
  console.log(i); // error!
  console.log(x); // error!

  let i = 1;
  console.log("antes: " + i); //1

  for (let i=0; i<10;i++){
    // cualquier cosa...
  }
  console.log("después: " + i); //1

}

z();
```

var, let y const

const

- Exactamente el mismo comportamiento que let, pero con valores constantes (no permite la reasignación)



ECMAScript 6 (ES6, ES2015)

ES6, ECMAScript 6, ES2015

ECMA (European Computer Manufacturers Association) es una organización que se dedica a definir estándares a nivel nacional e internacional.

ECMAScript es la especificación que esta organización ha publicado con el fin de estandarizar JavaScript.

ECMAScript 6 se publicó en 2015, por ello también se denomina ES2015. Es la última versión con cambios sustanciales, aunque también se han publicado la ES2016, 17 y 18 con mejoras menos importantes.

Importante: No todos los navegadores aceptan las últimas especificaciones ES.

Babel es un compilador que traduce sintaxis nueva en otra más antigua para asegurar la ejecución del código (requiere node/npm).

Principales novedades ES6:

- **Clases y módulos**
- **Operadores Rest/Spread**
- **Desestructuración**
- **Arrow functions**
- **Array Helpers**

Clases /1

ES6 introduce el concepto de “clase”, como una ampliación sintáctica (sigue siendo un lenguaje basado en prototipos).

Definimos la clase mediante el keyword "class", y creamos los objetos mediante "new".

La clase puede tener un constructor, el cual recibirá los parámetros que acompañan a "new", normalmente para inicializar sus atributos.

```
class Punto {
    constructor(x,y){
        this.x = x;
        this.y = y;
    }
}

punto1 = new Punto(2,4);
punto2 = new Punto(6,3);

console.log(punto1.x); // 2
console.log(punto2.y); // 3
```

Clases /2

Hasta ES6:

```
function Circulo(x,y,radio) {
     this.x = x;
     this.y = y;
     this.radio = radio;
    this.area = function(){
      return Math.PI*this.radio**2;
    }
}

a = new Circulo(0,0,2);
console.log(a.area()); //12.566
```

ES 6:

```
class Circulo {
  constructor(x,y,radio){
    this.x = x;
     this.y = y;
     this.radio = radio;
  }
  area(){
    return Math.PI*this.radio**2;
  }
}
```

Clases: herencia

Una clase puede crearse a partir de otra clase mediante "extends":

class Gorila **extends** Animal {...}

En este caso, Gorila heredará todos los atributos y métodos de Animal, y sus objetos los podrán utilizar.

En la definición de la clase hija, **es necesario invocar al constructor de la clase principal** mediante **super().**

(Ver ejemplo completo)

```
 class Animal {
   constructor(nombre, peso) {
     this.nombre = nombre;
     this.peso = peso;
     }
}

class Gorila extends Animal {
   constructor(nombre, peso, color)
{
      super(nombre, peso);
     this.color = color;
   }
}
```

Clases: herencia /2

super también permite referirse a métodos de la clase original, en la definición de la clase heredada.

```
class Punto{
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return '(' + this.x + ', ' + this.y + ')';
    }
}
```

```
class PuntoColor extends Punto {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return super.toString() + ' es ' + this.color;
    }
}

let cp = new PuntoColor (25, 8, 'verde');
console.log(cp.toString()); // '(25, 8) es verde'

console.log(cp instanceof PuntoColor); // true
console.log(cp instanceof Punto); // true
```

Ejercicio con clases

- Crear una clase **Figura** con los atributos **x, y**
- Crear una clase **Rectangulo** que extienda Figura e implemente el atributo **lado1** y **lado2** el método **area**. El método area debe devolver la superficie del rectángulo.
- Crear una clase **Triangulo**, extendiendo Figura e implementando los atributos **base, altura** y el método **area**.
- Crear una clase **Cuadrado** que extienda Rectángulo, con un constructor que reciba un solo lado y lo asigne a los lados del rectángulo mediante super()

Probar todo en JS BIN

Módulos

Módulos

Los módulos son unidades de código reusable. Se trata de archivos javascript (extensión js) con clases, funciones o variables que podemos invocar desde otros archivos.

Para que cualquier contenido de un módulo sea utilizable "desde fuera" se debe exportar con **export**

De modo similar, para incorporar contenido de un módulo en otro archivo, se debe importar con **import**

Si se trata de clases normalmente tendremos un archivo único para la clase, y lo exportaremos mediante **export default**

En este caso, al importar con import podemos indicar directamente el nombre de la clase.

Si no se ha efectuado export default, sino solo export, debemos indicar el nombre del elemento a importar entre llaves {}

Módulos: ejemplo

Archivo Moto.js

```
class Moto {
 constructor (modelo, cc, cv){
      this.modelo = modelo;
      this.cc = cc;
      this.cv = cv;
      }
}

export default Moto;
```

Archivo Test.js

```
import Moto from "./Moto.js";
let moto1 =
      new Moto("Scoopy", 125, 12);

console.log(moto1.modelo);
```

Módulos: ejemplo

Archivo Util.js

```
export const THE_SERVER =
'http://localhost:3001/';

export const doble = a =>a+a;
```

Archivo Test.js

```
import {doble} from "./Util";

console.log(doble(2)); //4
```

Módulos: uso en navegador

Los módulos pueden usarse en el navegador, aunque son más útiles en entornos como React o Node JS.

Para usarse en el navegador debe indicarse con **<script type="module">** tal como se muestra en el ejemplo.

Nota: en Chrome el ejemplo no funcionará por un tema de seguridad (CORS). Probad en Edge.

Archivo Test.html

```
<script type="module">
     import Moto from './Moto.js';
     let moto1 = new Moto("Scoopy",
     125, 12);
     console.log(moto1.cc);
</script>
```

Archivo Moto.js

```
export default class Moto {
    constructor (modelo, cc, cv){
   this.modelo = modelo;
   this.cc = cc;
   this.cv = cv;
   }
}
```

Módulos: Node JS

A pesar de que import i export forman parte de las especificaciones ES6, Node en su versión actual todavía no los soporta. Hay que utilizar una forma anterior: require.

El ejemplo anterior en formato "require" sería así:

Test.js

```
const Moto = require('./Moto.js');
let moto1 = new Moto("Scoopy", 125,
12);
console.log(moto1.cc);
```

Moto.js

```
class Moto {
  constructor (modelo, cc, cv){
   this.modelo = modelo;
   this.cc = cc;
   this.cv = cv;
   }
}
module.exports = Moto;
```

Ejecución:

node Test.js

Rest, Spread

Rest/Spread

Una nueva sintaxis permite recibir argumentos múltiples y componer arrays de forma simple, utilizando "..."

Ejemplo 1:

```
letras = ["a", "b", "c"];
numeros = [1,2,3];
mix = [...letras, ...numeros];
// ["a", "b", "c", 1, 2, 3]
```

Ejemplo 2:

```
function uneDespues(lista, ...elementos){
  return [...lista, ...elementos];
}

function uneAntes(lista, ...elementos){
  return [...elementos, ...lista];
}

l1 = uneAntes([1,2,3], 5,6,7);
l2 = uneDespues([1,2,3], 5,6,7);
console.log(l1); //[5, 6, 7, 1, 2, 3]
console.log(l2); //[1, 2, 3, 5, 6, 7]
```

Desestructuración

Se permite asignar varias variables a la vez a los valores de un array. Ejemplo:

```
lista=[1,3,4];
[a,b,c]=lista;
console.log(a,b,c);
```

```
1
3
4
```

```
let motos = [ ["scoopy", 125],
["burgman", 400], ["tmax", 530]];

for(i in motos){
     [modelo, cc]=motos[i];
     console.log(`${modelo} cubica
     ${cc}`);
}
```

```
"scoopy cubica 125"
"burgman cubica 400"
"tmax cubica 530"
```

Arrow functions

Arrow functions

Las arrow functions son una nueva sintaxis para simplificar la definición de funciones, especialmente utilizadas en funciones anónimas que se pasan como parámetro (callbacks).

Se utiliza el símbolo => para su creación, que puede tener varios casos:

```
(param1, param2, ...) => {sentencias}
(param1, param2, ...) => expresión
// equivale a { return expresión; }
```

```
Paréntesis opcionales cuando un
solo param:
(unParam) => {sentencias}
unParam => {sentencias}
unParam => expresión

Sin parámetros:
() => {sentencias}

Podemos usar rest y default:
(param1, param2, ...rest) => {sentencias}
(param1=default1, param2...) => { }
```

Array Helpers
Uso avanzado de arrays

Repaso de arrays

Definir array:

```
let lista = [];
let lista1 = [1,2,3];
```

Añadir elementos a un array:

```
z=[]; //crea array
z[0]="a"; //inserta en pos 0
z[3]="b"; // inserta en pos 3
z.push("c"); //inserta al final
z.unshift("x"); //inserta al inicio
console.log(z);
//["x", "a", undefined, undefined,
"b", "c"]
```

Eliminar elementos de un array:

```
// elimina primer elemento  y lo retorna
primero = z.shift();
// elimina último elemento y lo retorna
ultimo = z.pop();
```

Encontrar elemento en un array:

a = [1,4,2,7,9]

a.indexOf(4); // 1 (índice empieza por 0)

a.indexOf(3); // -1 (no existe)

Otras:

Repaso de arrays

Obtener subarray:

```
let lista = [1,2,3,4,5,6];
let parte1 = list.slice(0,3);
let parte2 = list.slice(3);
```

slice devuelve un nuevo array, sin modificar el original, con los elementos entre el primer índice y el segundo, sin incluir a éste.

Convertir de texto a array: split

```
let frase = "esto es una frase";
let palabras = frase.split(" ");
console.log(palabras);
//["esto", "es", "una", "frase"]
```

Convertr un array en texto: join

```
let lista = ["yamaha", "honda", "ducati"]
let marcas = lista.join(", ");
console.log(marcas);
```

Array Helpers. Filter.

Para gestionar datos de forma eficiente con Javascript es muy útil conocer los siguientes métodos relacionados con el manejo de listas:

- filter
- map
- forEach
- sort

Todos los métodos pertenecen al objeto Array. Normalmente les pasaremos una **arrow function**, pero también puede ser una función "normal".

FILTER: Aplica una condición sobre cada elemento de una lista, devolviendo una lista nueva con los elementos de la primera que cumplen la condición. Ej:

```
var lista=[1,2,3,4,5,6];
var pares = lista.filter(el => el%2===0);
console.log(pares); // [2,4,6]
```

La arrow function:

```
el => el%2===0
```

equivaldría a:

```
function es_par(el){
    return (el%2===0);
}
```

Array Helpers. Map y Foreach

MAP: Aplica una función a cada elemento de una lista, devolviendo una lista nueva con los resultados. Ej:

```
var lista=[1,2,3,4,5,6];
var dobles= lista.map(el => el*2);

console.log(dobles); // [2,4,6,8,10,12]
```

FOREACH: Aplica una función a cada elemento de una lista, **sin devolver nada**. Ej:

```
var lista=["barcelona", "girona", "mataró"];
lista.foreach( (el, indice) => {
        console.log("elemento: " +el);
        console.log("indice: "+indice);
});
```

Generaría la siguiente salida en consola:

elemento: barcelona
índice: 0
elemento: girona
índice: 1
elemento: mataró
índice: 3

Array Helpers. Sort.

SORT: Ordena una lista. Atención: modifica la lista directamente, no crea una nueva!

```
var lista=[4,2,5,1,3,6];
console.log(lista); // [4,2,5,1,3,6]

lista.sort();
console.log(lista); // [1,2,3,4,5,6]
```

Array.sort funciona directamente para elementos numéricos o strings, pero no para objetos. En este caso deberemos crear una función que compare dos objetos a,b y devuelva 1,0,-1 segun a>b, a=b, a<b respectivamente.

Por ejemplo, tenemos una lista de objetos del tipo: [{ nombre: xx, edad: yy], ...] y queremos ordenarla por edad, debemos crear una función que compare objetos:

```
function comparaEdad(a,b) {
        if (a.edad===b.edad) return 0;
        if (a.edad>b.edad) return 1;
        return -1;
}
```

Y finalmente podemos ordenar la lista "personas" utilizando esta función:

```
personas.sort(comparaEdad);
```

Ejemplo Sort aplicado a objetos

```
personas = [{nombre:"anna", edad:26},
{nombre: "joan", edad:25},
{nombre:"marta", edad:22}];

function ordenarEdad(a,b){
  if (a.edad>b.edad) { return 1; }
  else if (a.edad<b.edad){ return -1;
}
  else return 0;
}

function ordenarNombre(a,b){
  if (a.nombre>b.nombre) { return 1; }
  else if (a.nombre<b.nombre){ return
-1; }
  else return 0;
}
```

```
personas.sort(ordenarEdad);
personas.forEach(p =>
console.log(p.nombre));

personas.sort(ordenarNombre);
personas.forEach(p =>
console.log(p.nombre));
```

Ejercicio con listas /1

A partir de un listado de motos facilitado, obtener los siguientes datos:

- La moto más cara y más barata.
- ¿Cuántas motos hay con menos de 30.000 km de la marca HONDA?
- ¿Cuántas motos hay con menos de 30.000 km de más de 240cc?
- ¿Qué moto tiene menos de 25.000 km, más de 350cc de cilindrada y cuesta entre 1.800 y 2.200 eur?
- Una lista de marcas distintas con el número de motos de cada una.

Nota: recordad que para obtener el tamaño de un array utilizamos "length":

```
let a = [1,2,3,4];
console.log(a.length); // 4
```

Ejercicio con listas /2

A partir del ejercicio de conexión a la base de datos de bicing, y utilizando las funciones map, sort, filter...

Función que nos devuelva la estación con más bicicletas libres. (sort con función compare personalizada, de más a menos bicis libres, tomar primer elemento)

Función que nos devuelva un array de nombres de estaciones que no tienen bicicletas libres (filter).

Función que devuelva las X estaciones más cercanas a la ubicación 41.388163, 2.179769.

Cálculo de distancia entre (x,y) y (x',y'):

- Raíz cuadrada de la suma de los cuadrados de (y-y') y (x-x').

Utilizar sort basado en esta distancia.

Promises (ES5)

Promises: callback hell

Callback Pattern

```
driver.get("http://www.google.com", function() {
 driver.findElement(By.name("q"), function(q) {
   q.sendKeys("webdriver", function() {
     driver.findElement(By.name("btnG"), function(btnG) {
       btnG.click(function() {
         driver.getTitle(function(title) {
           assertEquals("webdriver - Google Search", title);
         });
       });
     });
   });
 });
});
```

Callback Hell
or
Pyramid of Doom

Promises

```
driver.get("http://www.google.com").
   then(function() {
     return driver.findElement(By.name("q"));
   }).
   then(function(q) {
     return q.sendKeys("webdriver");
   }).
   then(function() {
     return driver.findElement(By.name("btnG"));
    }).
   then(function(btnG) {
     return btnG.click();
    }).
   then(function() {
     return driver.getTitle();
   }).
   then(function(title) {
     assertEquals("webdriver - Google Search", title);
   })
   .catch(function(err) {
     console.log(err);
   });
```

Promises

Una Promise es un objeto obtenido a partir de la clase Promise.

Este objeto recibe una función que debe gestionar dos parámetros: resolve y reject.

Estos parámetros son en realidad funciones que deberán recibir un dato de salida, la primera (resolve) si la promesa se cumple, la segunda (reject) si se incumple.

```
const conecta = new Promise(
    function (resolve, reject) {
        if (hay_conexion) {
            const conn = {
                server: 'url_servidor',
                error: 'false'
            };
            resolve(conn); // conseguido!
        } else {
            const motivo =
                  new Error('no hay conexión');
            reject(motivo); // reject
        }

    }
);
```

Promises /2

Para ejecutar la promise debemos invocar el objeto y utilizar sus métodos "then" i "catch" para definir las funciones que gestionarán el objeto devuelto por resolve y reject respectivamente.

A menudo utilizamos las arrow functions para simplificar el uso de las Promises.

Las promises suelen encadenarse, y normalmente gestionan funciones asíncronas.

```
conecta
.then( (conn) => {
    console.log("conexion obtenida");
})
.catch( (error) => {
    console.log(error.message);
});
```

Ver ejemplo completo "promises.js" para observar el encadenamiento.