







UNIDAD 4

Conceptos importantes en Javascript ES6









ÍNDICE:

- 1.- Introducción
- 2.- Control de Errores
- 3.- Uso de Rest
- 4.- Novedad en funciones ES6 (Arrow)
- 5.- Orientación a objetos
- 6.- Encadenar métodos
- 7.- Funcionalidad de Módulos
- 8.- Expresiones regulares
- 9.- Funciones anónimas autoejecutables
- **10.- Temporizadores**

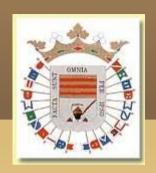


1.- Introducción

ECMAScript 6, también conocido como ES6, es la nueva versión de Javascript, aprobada en Junio 2015.

A partir de Emac Script 6 es posible nuevas funcionalidades y fundamentalmente crear clases y herencia en Javascript, muy usados en otros lenguajes como C# y Java.

Sin embargo esta nueva sintaxis de clases no introduce un nuevo modelo orientado a objetos, solamente simplifica la sintaxis para la herencia basada en prototipos que poseía javascript.



2.- Control de Errores

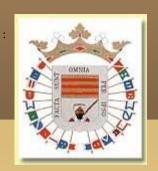
Nos va a permitir gestionar errores, capturando cualquier mensaje de error que lance nuestro código. La estructura típica del manejo de errores es:

```
try {
//sección donde se ejecuta y evalúa nuestro código
}
catch(error) {
// zona que captura cualquier error surgido o lanzado en el try
}
finally {
// zona para ejecutar cualquier código siempre al final de un bloque try-catch.
```



Ej. Control de errores

```
//errorDefinicion1; //error en definición variableDefinicion1
 trv {
      let numero='v':
      //lanzamos error si detectamos alguno
      if (isNaN(numero)) {
        throw new Error("No has introducido un número");
//error en definición variable errorDefinicion2,
en el momento en que se encuentra un error no sigue interpretando el código
         console.log("Esto ya no se interpreta porque se detectó error justo antes");
catch(error) {
    // zona que captura cualquier error surgido o lanzado en el try
       console.log(error); //imprimimos el error gestionado con catch
finally {
   // zona para ejecutar cualquier código siempre al final de un bloque try-catch.
    console.log("Esto se ejecuta siempre al final");
```



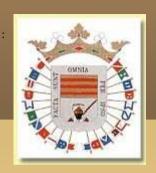
4.- Uso de Rest (spred operator)

Se usa fundamentalmente para pasar un array de valores como parámetro a una función, convirtiendo cada uno de esos valores en un argumento distinto.

Ejemplos:

```
function suma(num1,num2,...numx){
    let resultado=num1+num2;
    numx.forEach(function(num){
        resultado+=num;
    })
return console.log(resyltado);
}
suma(1,3,5,7,9);
```

Mediante Parámetros Resto podemos pasar un número indeterminado de parámetros, teniendo en cuanta que si la función tiene múltiples parámetros solo el último puede ser parámetro rest



4.- Uso de Rest (spred operator)

```
El spred operator (...) se puede usa para unir dos arrays.

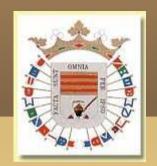
Ej:

const array1 =[10,20,30,40],

array2 =[50,60,70,80,90];

const array3=[...array1,...array2];

console.log(array3);
```



3.- Novedad en Funciones ES6 (Arrow)

Son funciones definidas usando una flecha =>

Sintaxis: ES5 ES6

Sin argumentos

Con 1 solo argumento (no se necesitan paréntesis)

```
var nombres = ['pepe', 'ana'];
nombres.forEach(function(parametro){
    document.write('nombre: ' + parame
tro+"<br>');});
var nombres = ['pepe', 'ana'];
nombres.forEach(parametro=>{
    document.write('nombre: '+parametro+"<br>'');});
+"<br/>);
```

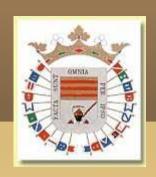
Con múltiples argumentos (paréntesis necesarios)

```
var suma = function(a, b){ return a+b;} var suma=(a,b)=>{ return a+b;}
console.log(suma(3,5)); alert(suma(3,5));
```

Una de las características que tienen las arrow function es que capturan el objeto this del contexto en el que se encuentran.

Ejemplos:

```
//this captura el objeto principal window del navegador
function saludar(){
    console.log (this);
   };
saludar();
//this en función declarada captura el objeto dentro del contexto en el que está
const alumno={ nombre: "pepe",
                apellidos: "Gil Robles",
                email: "pepe@inventado.com",
                matricular: function (){console.log(this);}
alumno.matricular();
//this en arrow function se salta el contexto del objeto y coge el padre en el que está
const alumno={ nombre: "pepe",
                apellidos: "Gil Robles",
                email: "pepe@inventado.com",
                matricular: ()=>{console.log(this);}
alumno.matricular();
Por tanto, mucho cuidado en usar arrow function dentro en métodos de objetos
```



Novedad en Funciones ES6(Parámetros)

Ya podemos incluir valores por defecto en los parámetros, incluso referenciando a otros parâmetros.

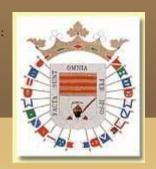
Parámetros por defecto



5.- Orientación a objetos (EMAC 6)

En Javascript ES5 no existía una manera específica de crear clases en POO con una declaración "class", aunque sí existían alternativas para crear componentes parecidos a lo que serían las clases (mediante prototype).

La versión reciente de Javascript ES6 incorpora la declaración de clases, aunque un poco diferentes a las de lenguajes como Java.



Orientación a objetos (ES6)

Declarar Clases

Se declara con la palabra reservada class y el nombre de la clase.

Constructor

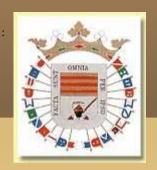
El constructor es un método especial que inicializa una instancia de la clase, con la sintaxis constructor([arguments]) { ... }

En JavaScript sólo puede haber un constructor (no existe el concepto de sobrecarga).

Métodos y propiedades

La definición de métodos ahora es más sencilla, ya no hay que escribir la palabra function, solo el nombre del método y su implementación.

Ejemplo:

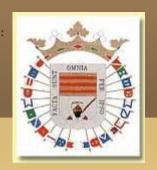


Método estáticos

Un método estático es aquel que se puede invocar sin tener que crear una instancia de una clase. Se crean anteponiendo la palabra static a la definición del método.

Ejemplo:

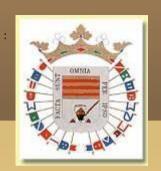
```
class Punto {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    static distancia(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.sqrt(dx * dx + dy * dy);
    }
}
const p1 = new Punto(1, 1);
const p2 = new Punto(3, 3);
alert(Punto.distancia(p1, p2));
```



Diferencia ES5 con preES6

Clases y herencia

```
Ejemplo de clases y herencia en ES6:
Ejemplo de clases y herencia en ES5:
                                                   class Documento {
function Documento(titulo, autor, publicado) {
                                                      constructor(titulo="",autor="",publicado=false) {
this.titulo=titulo:
                                                             this.titulo = titulo:
this.autor = autor;
                                                             this.autor = autor:
Documento.prototype.publicar = function publicar() {
this.publicado = true;
                                                   publicar(){ this.publicado = true; }
function Libro(titulo,autor,tema) {
                                                   class Libro extends Documento{
Documento.call(this, titulo,autor);
                                                      constructor(titulo,autor,tema){
Documento.prototype.publicar();
                                                             super(titulo,autor,false);
this.tema = tema;
                                                             this.publicar();
                                                             this.tema = tema;
Libro.prototype =
Object.create(Documento.prototype);
milibro= new Libro("El Quijote", "Cervante", "novela");
                                                   var milibro=new Libro("titulo1","autor1","novela");
alert(milibro.titulo+" "+milibro.publicado);
                                                   alert(milibro.titulo+" "+milibro.publicado);
```

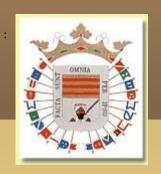


6.- Encadenar métodos (chaining)

En javascript es posible encadenar métodos con la notación del punto (.)

Sintaxis: objt.metodo1().metodo2().metodo3()

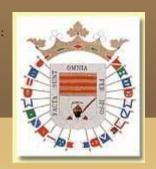
Esta sintaxis es propia de los lenguajes de programación orientados a objetos y simplifica la escritura del código.



7.- Funcionalidad de Módulos

ES6 incluye la funcionalidad de módulos, que permite a javascript importar y exportar objetos, funciones y clases desde el propio código y no tener que importarlos desde HTML como en ES5.

Es importante y surge como solución para cuando tenemos todas nuestras funciones en archivos js, donde unos importan a otros, debiendo hacerlo en el **orden correcto** para que se cargen en el orden adecuado y todo funcione correctamente. Por tanto, debemos pensar en módulos como archivos js que pueden compartir lo que deseemos.



Funcionalidad de Módulos

Veamos un ejemplo cuando la función a exportar es por defecto:

Supongamos que tenemos un archivo modulo.js donde yo quiero tener varias funciones y exportar (poner a disposición de) alguna o todas ellas.

JS modulo.is X JS main.is O ejemplo.html

Ahora tendré que importar esa función desde cualquier otro módulo (archivo js), por ejemplo desde main.js

Cualquier identificador para nombrar la exportación por defecto

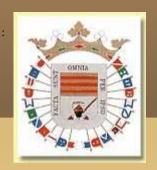
Ahora solo queda probar la ejecución:

```
Js modulo.js Js main.js × <> ejemplo.html

1 import sumar from './modulo.js';

2 sumar(3,4);
```

export default function(a,b){
 alert(a+" + | +b+" = "+ (a+b));



Funcionalidad de Módulos

Veamos el mismo ejemplo cuando la función a exportar sí lleva nombre:

Supongamos que tenemos un archivo modulo.js donde yo quiero tener varias funciones y exportar (poner a disposición de) alguna o todas ellas.

JS modulo.js • JS main.js • O ejemplo.html

1 export default function sumar(a,b){
2 alert(a+" + "+b+" = "+ (a+b));
3 }

sumar(3,4);

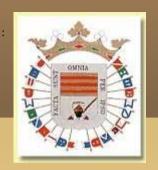
Ahora tendré que importar esa función desde cualquier otro módulo (archivo js), por ejemplo desde main.js

Es importante que al importar, elijamos la ruta correcta, porque si no, siempre buscará en el archivo node modules por defecto.

JS modulo, js main. js modulo, js main. js modulo. js main. js main. js modulo. js main. j

Ahora solo queda probar la ejecución:

import {sumar} from './modulo.is';



Funcionalidad de Módulos

Ejemplo exportando varias funciones nombradas de un módulo:

Supongamos que tenemos un archivo modulo.js donde yo quiero tener varias funciones y

exportar (poner a disposición de) alguna o todas ellas.

Ahora tendré que importar esas funciones desde cualquier otro módulo (archivo js), por

ejemplo desde main.js

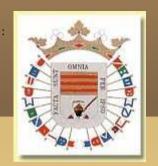
Ahora solo queda probar la ejecución:

```
JS modulo.js ● JS main.js × ⇔ ejemplo.html

1 import {sumar,multiplicar} from './modulo.js';

2 sumar(3,4);

3 multiplicar(3,4);
```



Funcionalidad de Módulos

Ejemplo exportando varias funciones nombradas y otra por defecto de un módulo:

Supongamos que tenemos un archivo modulo.js donde yo quiero tener varias funciones y

exportar (poner a disposición de) alguna o todas ellas.

Ahora tendré que importar esas funciones desde cualquier otro módulo (archivo js), por

ejemplo desde main.js

Cualquier identificador para nombrar la exportación por defecto

Ahora solo queda probar la ejecución:

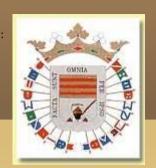
```
Js modulo.js ● Js main.js × ◇ ejemplo.html

1 import error,{sumar,multiplicar} from './modulo.js';

2 sumar(3,4);

3 multiplicar(3,4);

4 error("Salida de error");
```



Funcionalidad de Módulos

Ejemplo exportar un objeto con todos los métodos que queramos:

Supongamos que tenemos un archivo modulo.js donde yo quiero tener varias funciones y

exportar (poner a disposición de) alguna o todas ellas.

```
Js modulo, x Js main. is ⇔ ejemplo.html

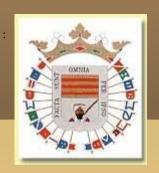
1 export function sumar(a,b){
2 alert(a+" + "+b+" = "+ (a+b));
3 }
4 export function restar(a,b)
5 {
6 alert(a+" - "+b+" = "+ (a-b));
7 }
8 export function multiplicar(a,b){
9 alert(a+" * "+b+" = "+ (a*b));
10 }
11 //objeto por defecto a exportar
12 export default
13 {
14 sumar:sumar,
15 restar:restar,
16 multiplicar:multiplicar
17 }
```

Ahora tendré que importar el objeto con cualquier identificador o alias

Ahora solo queda probar la ejecución:

```
Js modulo.js Js main.js × ⇔ ejemplo.html

1 import objeto from './modulo.js';
2 objeto.sumar(3,4);
3 objeto.restar(3,4);
4 objeto.multiplicar(3,4);
```

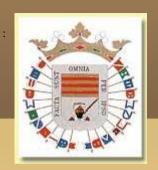


Funcionalidad de Módulos

Aquí se muestran todas las posibilidades de importar módulos que se hayan puesto en exportación:

```
import alias from "archivo.js";
import * as name from "archivo.js";
import { funcionnombrada } from "archivo.js";
import { funcionnombrada as alias } from "archivo.js";
import { funcionnombrada1 , funcionnombrada2 } from "archivo.js";
import { funcionnombrada1 , funcionnombrada2 as alias2 , [...] } from "archivo.js";
import alias, { funcionnombrada [ , [...] ] } from "archivo.js";
import alias, * as name from "archivo.js";
import "archivo.js";
```

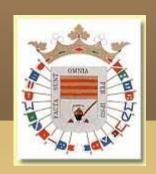
Para más información, visitar: https://es.javascript.info/import-export



8.- Expresiones regulares

Cualquier lenguaje en POO que se preste, debe hacer uso de las expresiones regulares (por ej., para validar). Como no podía ser de otra manera javascript puede hacer un uso excelente de expresiones regulares.

Caracteres especiales en expresiones regulares.	
Caracteres/construcciones	Artículo correspondiente
$\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ $	<u>Clases de caracteres</u>
^, \$, x(?=y), x(?!y), (?<=y)x, (? y)x, \b, \B</td <td><u>Aserciones</u></td>	<u>Aserciones</u>
(x), (?:x), (? <name>x), x y, [xyz], [^xyz], \<i>Number</i></name>	Grupos y rangos
*, +, ?, $x\{n\}$, $x\{n,\}$, $x\{n,m\}$	<u>Cuantificadores</u>
\p{UnicodeProperty}, \P{UnicodeProperty}	Escapes de propiedades Unicode



Método test con Expresiones regulares

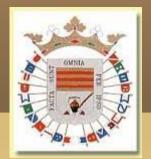
RegExp.prototype.test()

El método test() ejecuta la búsqueda de una ocurrencia entre una expresión regular y una cadena especificada. Devuelve true o false

```
Sintaxis: Hay dos posibles:
    new RegExp(patrón,bandera)
    regexObj.test(cadena) donde cadena es la cadena a comparar con la expresión regular
Ejs: var cadena = "hello world!";
    var result =
    var result = /^hello/.test(cadena);
    console.log(result); // devuelve true
```

let cadena= = 'Miguel Ángel León';

console.log(/Angel/.test(cadena));



9.- Funciones anónimas autoejecutables

Las funciones anónimas autoejecutables son aquellas que se ejecutan nada más definirse.

Sintaxis: (function () { Sentencias})();

Consta de dos partes bien diferenciadas:

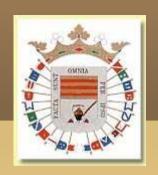
- La primera es la función anónima
- La segunda parte crea la expresión de función cuya ejecución es autoejecutable (), siendo interpretado directamente por el motor de javascript



Ejemplos:

```
(function(){
      console.log("Ejecutado directamente en la primera función autoejecutable");
      document.write("Ejecutado directamente en la primera función autoejecutable"+"<br/>))();

var funcionAnonimaAutoejecutable = (function () {
      console.log("Ejecutado directamente en la segunda función autoejecutable");
      document.write("Ejecutado directamente en la segunda función autoejecutable");
      var a=3
          return a;
      })();
    //console.log(a); //a no está definida
      console.log(funcionAnonimaAutoejecutable)
      </script>
```



10.- Temporizadores

Son dos sentencias javascript que van a permitir realizar una función callback cada cierto tiempo: SetTimeout y SetInterval. La única diferencia es que SetTimeout se ejecuta una sola vez con el tiempo estimado y SetInterval se ejecuta indefinidamente cada vez con el tiempo estimado.

```
setTimeout(()=>{},tiempo en milisegundos);
setInterval(()=>{}, tiempo en milisegundos);
```

Además, hay dos sentencias que sirven para detener a los temporizadores, clearTimeOut y clearInterval.

```
clearTimeOut(temporizador);
clearInterval(temporizador);
```



```
Ejemplo de temporizadores:
<script>
        setTimeout(()=>{console.log("Esto se ejecuta una sola vez a los 3 segundos");},3000);
        setInterval(()=>{console.log("Esto se ejecuta indefinidamente cada segundo");},1000);
        setInterval(()=>{});
</script>
<script>
       var temporizador1 = setTimeout(()=>{document.write("Esto se ejecuta una
sola vez a los 3 segundos"+"<br>");},3000);
        var temporizador2 = setInterval(()=>{document.write("Esto se ejecuta
indefinidamente cada segundo"+"<br>");},1000);
        clearTimeout(temporizador1);
        clearInterval(temporizador2);
</script>
```