



EcmaScript 6 y Typescript



Formador: Vladimir Bataller

<https://es.linkedin.com/in/vladimirbataller>
vladimirbataller@yahoo.es

Requisitos:

- HTML y CSS
- Javascript.



Ecmarkup 6

ES6 / ES2015

¿Como puedo probar ES6?

ES6 no está implementado a 100% en todos los navegadores, aunque los principales tienen la mayor parte de los navegadores están cerca de ese 100%.

Para probar la sintaxis se puede emplear node.js, para ello una vez instalado se puede ejecutar cualquier archivo ES6 (por ejemplo: prueba.js) en Node con el siguiente comando:

node prueba.js

La única condición es que solamente se pueden mostrar datos en la consola, ya que no existe ningún navegador donde renderizar datos, ni se puede acceder al DOM, ni existe un archivo HTML.

Si se desea probar en un navegador, otra opción es declarar los archivos de prueba con la extensión *.ts (de TypeScript, aunque de este solamente se emplee ES6) y transpilarlos a ES5. Para ver la instalación y configuración de TypeScript ir a la sección TypeScript de este documento.

Variables

Mediante *let* se pueden declarar variables locales al ambito (bloque) en el que están declaradas, cosa que no ocurría con *var* (que eran locales a la función en la que estaban declaradas).

ES5	ES6
<pre>function bombilla(){ if (true){ var luz = "on"; console.log("1.- " + luz); //on } console.log("2.- " + luz); //on luz="off"; console.log("3.- " + luz); //off }</pre>	<pre>function bombilla(){ if (true){ let luz = "on"; console.log("1.- " + luz); //on } console.log("2.- " + luz); //Reference Error: luz is not defined luz="off"; console.log("3.- " + luz); }</pre>

Constantes

Mediante la palabra reservada *const* se declaran constantes cuyo valor asignado no se puede modificar.

ES6

```
const MAX_USUARIOS= 100;  
MAX_USUARIOS = 200; //TypeError: Assignment to constant variable
```

Clases

Se declaran con la palabra reservada `class` y pueden heredar de otra clase indicandose con la palabra reservada `extends`. El constructor se declara con la palabra *constructor*. Tanto el constructor como los métodos no emplean la palabra `function`.

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

```
class Rectangulo extends Figura {  
  constructor(x, y, l1, l2) {  
    super(x, y);  
    this.l1 = l1;  
    this.l2= l2;  
  }  
  area() {  
    return this.l1 * this.l2;  
  }  
  toString(){  
    return "(" + this.x + ", " + this.y + ") -" +  
      "["+ this.l1 + ", " + this.l2 + "];"  
  }  
}
```

```
let r = new Rectangulo(2,2,3,4);  
console.log("La figura " + r);  
console.log("tiene un area:" +r.area());
```

Función flecha o Expresiones Lambda (I)

ES6 introduce el operador `=>` que permite simplificar el paso de funciones anónimas.

ES5	ES6
<pre>var matriz= [{...}, {...}, {...}, ...]; matriz.forEach(function(elemento){ // Iteración sobre cada elemento console.log(elemento); });</pre>	<pre>var matriz = [{...}, {...}, {...}, ...]; matriz.forEach(elemento => { // Iteración sobre cada elemento console.log(elemento); });</pre>
<pre>// Cuando la implementación es solamente una línea, se pueden omitir las llaves. var suma = function (a,b){ return a + b; };</pre>	<pre>// Se puede omitir el return var suma = (a,b) => a + b;</pre>

Función flecha o Expresiones Lambda (II)

El operador `=>` permite resolver el problema de acceso a `this` desde contextos anidados (por ejemplo en funciones de callback de eventos).

ES5

```
var obj = {
  foo : function() {...},
  bar : function() {
    var that = this;
    document.addEventListener("click", function(e) {
      that.foo();
    });
  }
}
```

```
var obj = {
  foo : function() {...},
  bar : function() {
    document.addEventListener("click", function(e) {
      this.foo();
    }.bind(this));
  }
}
```

ES6

```
var obj = {
  foo : function() {...},
  bar : function() {
    document.addEventListener("click", (e) =>
      this.foo());
  }
}
```

Template Strings

Se definen entre comillas invertidas y dentro de ellas se pueden emplear expresiones encerradas en `${}` que serán evaluadas y su resultado se incluirá en el string resultante. Además se pueden trocear en varias líneas.

ES6

```
let nombre = "Vladimir";  
let apellidos = "Bataller";  
let saludo = `¡¡¡Hola, soy ${nombre} ${apellidos}!!!`;  
alert(saludo);
```

```
let texto = `En un lugar de la mancha  
de cuyo nombre no quiero ...`;
```

Operador propagación

El operador propagación (spread operator) permite a una función o método que espera una lista de argumentos, pasarle una matriz y que se pasen todos sus elementos como argumentos. El operador consiste en tres puntos (...)

El operador propagación se antepone a una matriz para obtener una lista con los valores de la matriz.

ES5	ES6
<pre>// El método Math.max puede recibir dos o mas // argumentos y retornar el mayor de todos ellos, // pero no puede recibir una matriz. var lista=[1,2,3,4,5,6]; var mayor=0; for (var i=0;i<lista.length;i++) { mayor = Math.max(mayor, lista[i]); } console.log(mayor);</pre>	<pre>// Mediante el operador propagación se puede // convertir la matriz en la lista de valores a pasar // a la función Math.max. var lista=[1,2,3,4,5,6]; var mayor= Math.max(...lista); console.log(mayor);</pre>

Descomposición de matrices y objetos

ES6 permite leer datos que se encuentran en una matriz o en un objeto y pasarlos a varias variables de una sola vez.

ES6

```
var [a, b] = ["lunes", "martes"];  
alert("Hoy es " + a + " y mañana " + b);  
  
var direccion = { calle: "Goya", numero: 33};  
var {calle, numero} = direccion;  
alert("vivo en la calle " + calle + ", " + numero);
```

Valores por defecto en funciones

A los argumentos que recibe una función o método, se les puede pasar un valor por defecto.

ES5	ES6
<pre>function sumar (a,b){ a = a 25; // if(!a) a=25; b = b 31; return a + b; }</pre>	<pre>function sumar (a=25,b=32){ return a + b; }</pre>

Módulos

Los módulos es un mecanismo que permite importar desde un archivo js, elementos definidos en otros archivos js.

Los elementos (variables, funciones, clases) que se exportan de un módulo se indican con la palabra *export*.
Para importar un elemento de un módulo se emplea: `import { nombreElemento } from "ruta/nombreModulo";`

ES6

```
// archivo mensajes.js
export function saludar(nombre){
    alert("Hola " + nombre);
}
```

```
// archivo app.js
import {saludar} from "./mensajes";
saludar("Pepe");
```

for(let ... of ...)

La palabra reservada del lenguaje permite recorrer una matriz y asignar, en cada iteración, el elemento actual a la variable.

```
let matriz=[{...}, {...}, {...},... ];  
for (let variable of matriz) {  
    console.log(variable);  
}
```

Clase Map

Permite crear mapas asociando pares de elementos, uno llamado clave y el otro el valor. En ES5 existían las matrices asociativas, pero en ese caso las claves solamente podían ser String. En el caso de los Map, no existe esa restricción.

```
let mapa = new Map();
mapa.set('lunes', 123);
console.log(mapa.size); //1
console.log(mapa.get('lunes')); //123
console.log(mapa.has('lunes')); //true
console.log(mapa.delete('lunes')); //true
console.log(mapa.has('lunes')); //false
```

Concatenación:

```
let map = new Map()
    .set(1, 'uno')
    .set(2, 'dos')
    .set(3, 'tres');
```


callback hell

Cuando se trabaja de forma asíncrona, mediante funciones de callback, surge el problema de que al encadenar varias llamadas asíncronas, el código alcanza un nivel de anidamiento poco legible.

```
function esperar(callback){ // Función que responde con una función de callback
    setTimeout(callback,2000);
}
esperar(function(){
    elemento.style.backgroundColor="yellow";
    esperar(function(){
        elemento.style.backgroundColor=backgroundColor="lightblue";
        esperar(function(){
            elemento.style.backgroundColor="lightgreen";
        });
    });
});
});
```

Uso de promesas

Las promesas son objetos de la clase Promise que internamente contienen o contendrán en el futuro el resultado de una determinada operación (síncrona o asíncrona).

Uso de una promesa

Supongamos que tenemos una una función llamada **sumaDiferida** retorna una promesa:

```
var promesa = sumaDiferida(2,5);
```

las promesas tienen el método *then* para poder especificar la función que se ejecutará cuando el resultado esté disponible. Dicho resultado se le pasará a la función como argumento:

```
promesa.then(function(resultado){  
    console.log("La suma de 3 más 5 es igual a " + resultado);  
})
```

Creación de promesas

Las promesas son objetos de la clase Promise que internamente contienen o contendrán en el futuro el resultado de una determinada operación (síncrona o asíncrona).

Esos objetos permiten, mediante la función **resolve** indicar cuando la promesa está resuelta, lo que hará que se ejecute la función pasada a su método **then**.

```
function sumaDiferida(a,b){  
    return new Promise(function (resolve, reject){  
        setTimeout(function(){ resolve(a+b); },2000);  
    });  
}  
  
.....  
sumadiferida(3,5).then(function(resultado){  
    console.log("La suma de 3 más 5 es igual a " + resultado);  
});
```

Promesas encadenadas

Los objetos de la clase Promise contienen una promesa. Esos objetos, en su constructor reciben una **función** que se ejecuta de inmediato y que recibe como argumento la función **resolve**. Mediante **resolve** se indica cuando la promesa está resuelta, lo que hará que se ejecute la función pasada al método **then** de la promesa.

```
function esperar() {
    return new Promise(function (resolve, reject) {
        setTimeout(function() { resolve("fin"); }, 2000);
    }); // La promesa se retorna de inmediato, pero se resolverá dentro de 2 s.
}

esperar().then(function() { // En "then" se indica la función a ejecutar cuando se cumpla la promesa.
    console.log("yellow"); return esperar();
}).then(function() { // Si then retorna otra promesa, al método "then" se le puede encadenar otro "then".
    console.log("lightblue"); return esperar();
}).then(function() {
    console.log("lightgreen");
});
```

resolve y reject:
then y catch

La clase Promise, además de la función **resolve** (trabajo terminado correctamente), proporciona la función **reject** (se ha producido un error). Cuando la promesa ejecute una de estas, automáticamente ejecutará las funciones pasadas a **then** y **catch**.

```
function esperar(){
    var inicio = (new Date()).getTime();
    return new Promise(function (resolve, reject){
        function tic(){
            var ahora = (new Date()).getTime();
            if(ahora>=inicio + 5000) resolve("fin");
            else if(ahora<inicio) reject("Reloj estropeado");
            else {console.log("."); setTimeout(tic,500)};
        }
        tic();
    });
}

esperar().then(function(msg){console.log(msg);})
.catch(function(msg){console.log(msg);});
```

Copiar objetos

La clase **Object** proporciona el método estático **assign** que permite transferir de un objeto todos sus atributos y los valores asignados a este a otro objeto.

El método assign recibe como primer argumento el objeto destino al cual se quieren copiar todos los atributos y valores del objeto de origen. El objeto de origen se pasará como segundo argumento. Finalmente el método, por comodidad, retorna el objeto destino.

```
let persona = {dni:22 , nombre:"Daniel", apellidos:"Valiente", saldo:834.777};  
let nuevaPersona = Object.assign( {}, persona);  
console.log(nuevaPersona.nombre);// "Daniel"
```

Del mismo modo funciona con matrices:

```
let matrizPersonas[] = [  
    {dni:22 , nombre:"Daniel", apellidos:"Valiente", saldo:834.777}, {...}, {...}  
];  
let nuevaMatriz = Object.assign( [], matrizPersonas);  
console.log(nuevaMatriz[0].nombre);// "Daniel"
```

En muchos casos el objeto destino que se pasa como argumento se crea en el momento como en estos ejemplos.



TypeScript

typescriptlang.org

Características

TypeScript es un superconjunto de ES6 que mediante un transpilador se puede convertir a ES5 por lo que este último se puede ejecutar sin problema en todos los navegadores actuales.

- Permite la declaración de variables (y argumentos de funciones y métodos) con tipo.
- Obliga a declarar los atributos de una clase.
- Permite la declaración de interfaces e indicar que una clase o una función implementa dicha interface.
- Permite declarar atributos de solo lectura dentro de una clase o una interfaz.
- Permite la declaración y uso de decoradores.

Instalación y uso con VS Code

TypeScript es un superconjunto de ES6 que mediante un transpilador se puede convertir a ES5 por lo que este último se puede ejecutar sin problema en todos los navegadores actuales.

Para instalar en transpilador de TypeScript se debe tener instalado Node.js y ejecutar el comando:

```
npm install -g typescript
```

En la carpeta raíz del proyecto se debe indicar la configuración de la transpilación mediante el archivo **tsconfig.json**:

```
{"compilerOptions":{"target":"ES5", "module":"commonjs"}, "include": [ "*.ts" ] }
```

Para realizar la transpilación, se debe configurar el archivo task.json de Visual Studio Code de la siguiente manera y luego ejecutar Ctrl + shift + b:

```
{"version": "0.1.0", "command": "tsc", "isShellCommand": true, "args": ["-w", "-p", "."],  
  "showOutput": "silent", "isBackground": true, "problemMatcher": "$tsc-watch"}
```

En este modo, el transpilador queda a la escucha de los cambios del archivo y los compila sobre la marcha.

Tipos de datos

number: números enteros o con parte decimal. `let edad:number; edad=20;`

string: texto entre comillas (simples, dobles o invertidas)

boolean: puede tomar true o false.

matrices con tipo: `Array<number>` o `number[]`

tuplas: `[number, string, boolean]`

enumeraciones: `enum Color {Red, Green, Blue}; let c: Color = Color.Green;`

any: indica que admite cualquier tipo. Como una variable javascript clásica (sin tipo). `let dato:any;`

void: indica que una función no retorna nada.

null, undefined: son tanto tipos como valores representativos de esos tipos (ningún objeto, ningún dato).

never: indica que una función nunca termina retornando un valor sino que producirá una excepción.

Interfaces

Las interfaces permiten indicar que atributos o métodos debe tener un objeto que debe ser pasado a una función o método.

```
interface Imprimible{
    alto: number;
    ancho: number;
    grosorPapel?:number; // Mediante el signo ? indicamos que es un campo opcional.
}
function imprimir(documento:Imprimible){
    if(!documento.grosorPapel){
        mandar Almprimir(documento);
    }
}
```

atributos readonly

Los atributos de las interfaces y las clases pueden ir precedidos con ***readonly***, lo cual significa que, en la creación de un objeto que implemente esa interface, solamente en ese momento se le podrá asignar un valor al atributo.

```
interface Imprimible{  
    readonly alto: number;  
    readonly ancho: number;  
}  
  
let documento: Imprimible = {alto:600, ancho:800};  
documento.alto = 1000; // Error.
```

Así mismo existe el tipo `ReadonlyArray<>` que permite definir una matriz inmutable:

```
let mm: ReadonlyArray<number> = [1,2,3,4];  
mm[0] = 10; // Error.  
mm.push(5); // Error.  
let m = mm; // Error.
```

Clases que implementan una interface

```
class Factura implements Imprimible{  
    alto: number;  
    ancho: number;  
    constructor (ancho:number, alto:number){  
        ....  
    }  
}
```

Interfaces con métodos

Las interfaces también pueden declarar métodos que deberán implementarse en una clase que implemente dicha interfaz.

```
interface Operacion(){  
    calcular(a:number,b:number):number;  
}  
class Suma implements Operacion {  
    calcular(a:number,b:number):number{  
        return a + b;  
    }  
}
```

Modificadores de acceso

En las clases TypeScript se pueden emplear los modificadores de acceso public, protected y private. Los atributos y método por defecto tienen el modificador public.

public: accesible desde cualquier sitio.

private: accesible solamente desde la clase en la que está definido.

protected: accesible solamente desde la clase en la que está definido y sus subclases.

Argumentos- propiedades

Si un argumento pasado a un constructor va precedido de public, private, protected o readonly. Este se declara implícitamente como un atributo (o propiedad) de la clase.

```
class Circulo{  
    // Al indicar public o protected o private delante  
    // de un parámetro del constructor, automáticamente  
    // ese parámetro pasa a ser también un atributo implícitamente.  
    constructor(public x, public y, public r){  
    }  
    public area():number{  
        return Math.PI * this.r + this.r;  
    }  
}  
  
var cir = new Circulo(1,5,2);  
console.log("Tiene un área de: " + cir.area());
```


Decoradores

Los decoradores son mecanismos para que permiten anotar (con información adicional) o modificar clases o miembros de una clase. El elemento que distingue la llamada de un decorador es que su nombre va precedido de @ y seguido de ().

Los decoradores se emplean ubicándolos del elemento al que modifican, por ejemplo delante de una clase.

@Sealed()

```
class FechaVencimiento{
    milisegundos:number;
    constructor(milisegundos:number){
        this.milisegundos = milisegundos;
    }
}
```

Los decoradores se definen mediante una función que en el caso de las clases se ejecutará cuando se llame a su constructor:

```
function Sealed(constructor: Function){
    Object.seal(constructor); //Object.seal -> impide que pueda ser modificado.
    Object.seal(constructor.prototype); //Object.seal -> impide que la clase sea modificada.
}
```