ES6

Ángel Villalba Fdez-Paniagua

Table of Contents

1. Let y Const

En la nueva versión de JavaScript (ES6), se han introducido dos nuevas formas de declarar las variables.

- **let**
- **const**

1.1. Let

La palabra **let** es muy parecida a la palabra **var**, las dos nos permiten declarar variables locales. La principal diferencia entre ellas es el alcance de las variables que vamos a declarar.

Las variables declaradas usando *let* solo van a poder ser accesibles dentro del bloque donde se han declarado, mientras que las que se declaran con *var* son accesibles dentro de la función en la que se han declarado.

```
function getValor(param1) {
  let res1 = 0;
  var res2 = 0;
  if (param1 == 1) {
    let res1 = 1;
    var res2 = 1;
    console.log('LET: ' + res1);
    console.log('VAR: ' + res2);
  }
  console.log('LET: ' + res1);
  console.log('VAR: ' + res2);
  return res1 + res2;
}

let valor = getValor(1);
```

1.2. Const

La declaración de variables usando la palabra **const** actua igual que cuando usamos **let**, es decir, que solo son accesibles desde dentro del bloque en el que se han declarado, pero con la diferencia de que va a ser una *constante*, por lo que no se puede modificar el valor de esta variable.

```
const NUM_MAX = 5;
```

Al ser una constante, si intentamos asignarle un nuevo valor, nos va a dar un error.

```
const NUM_MAX = 5;
NUM_MAX = 4;
```

2. Bucle: FOR OF

Este bucle es como el bucle `for in` que hemos visto anteriormente, solo que esta vez en lugar de guardar en la variable la *posición* o la *clave*, se guarda el valor del elemento.

```
for (var k of [1, 2, 3, 4]) {
  console.log('Elemento del array en esta iteración' + k);
}
```

3. Template literals

Los **templates literals** son una nueva forma de crear *strings*. Para usarlos hay que añadir el texto que queremos crear entre las comillas \\``.

```
var texto = `Si eres bueno en algo, nunca lo hagas gratis`;
```

El principal problema a la hora de crear strings a los que les queremos asignar valores que nos devuelve alguna expresión, es que tenemos que ir concatenandolos junto al texto que queremos mostrar, y esto puede darnos bastante trabajo si tenemos que mostrar bastantes valores almacenados en variables.

Con los *template literals* vamos a evitar concatenar todo esto. Para ello, donde queramos mostrar una variable o el resultado de una expresión, vamos a añadir la expresión o la variable entre `${}`.

```
var cuenta = '2+2';
var resultado = `El resultado de ${cuenta} es ${2+2}`;
```

Por último, si queremos añadir saltos de línea, ahora lo podemos hacer de una forma mucho más sencilla que antes. Solo hay que saltar de línea dentro del *template literal*, en lugar de añadir `\n` como haríamos usando el método antiguo.

```
var textoMultiLinea = `Este texto aparece
en varias
líneas`;
```

4. Arrow functions (funciones flecha)

Las **arrow functions** que a veces se les llama funciones lambda, son funciones anónimas que se

usan muy amenudo en JavaScript.

Hay muchos métodos que reciben como parámetro una función anónima, y es muy posible que esta función anónima tenga solo una linea de código. Para esta sola linea de código estariamos usando la palabra reservada `function`, habria que abrir llaves `{}` y en caso de devolver un valor usar la palabra reservada `return`. Las arrow functions nos ahorran el tener que escribir todo ese código para solo una linea que va a contener en el cuerpo, y se llaman arrow functions porque el cuerpo y los parámetros van separados por una flecha (`⇒`).

```
let misPeliculas = [
  { titulo: 'Scary movie', genero: 'comedia' },
  { titulo: 'La jungla de cristal', genero: 'accion' },
  { titulo: 'Los mercenarios', genero: 'accion' },
  { titulo: 'Salvar al soldado Ryan', genero: 'belica' }
];

let peliculasComedia = misPeliculas.filter(function (pelicula) {
  return pelicula.genero === 'comedia';
});

let peliculasAccion = misPeliculas.filter(pelicula => pelicula.genero === 'accion');
```

Cuando estas funciones no reciben parámetros o reciben más de uno, entonces hay que ponerlos entre parentesis, mientras que si reciben solo uno, da igual si ponemos los parentesis o no. Y en caso de tener más de una linea de código, el cuerpo de la función tiene que ir entre llaves.

```
misPeliculas.forEach(() => console.log('Vista!'));
misPeliculas.forEach(pelicula => console.log(pelicula.titulo + ' vista!'));
misPeliculas.forEach((pelicula, index) => console.log(pelicula.titulo + ' (' +
pelicula.genero + ')' + ' vista!'));
misPeliculas.forEach((pelicula, index) => {
  let texto = pelicula.titulo + ' (' + pelicula.genero + ')' + ' vista!';
  console.log(texto);
});
```

Normalmente cuando necesitamos usar la variable **this** dentro de un *callback*, tenemos que asignarle su valor a otra variable que se suele llamar **self**. Con las arrow functions no hace falta capturar la variable **this** sino que te deja usarla directamente.

```
function pelicula() {
  let self = this;
  self.añoEstreno = 2000;
  setTimeout(function () {
    console.log(self.añoEstreno);
  }, 1500);
}
pelicula();

function peliculaArrow() {
  this.añoEstreno = 2000;
  setTimeout(() => {
    console.log(this.añoEstreno);
  }, 1500);
}
peliculaArrow();
```

5. Rest params

Los parámetros **rest**, se usan para recoger un conjunto de valores en un array. Este tipo de parámetros tienen que ir en la última posición cuando se definen los parámetros en las funciones.

```
function getNumeroLoteria(...nums) {
  return nums.join(', ');
}

let num = getNumeroLoteria(1, 5, 12, 22, 35, 37);
console.log(num);
```

6. Spread operators

El operador **spread** lo que nos permite es pasarle un array de valores a una función, y esta va a separarlos en distintos valores. Se usa igual que el anterior, al pasar el array a la función se le ponen los ⋯ delante del nombre de la variable, y esa función recibirá los elementos del array como valores independientes.

```
let numeros = [1, 3, 6, 2, 8, 0, 2];

let max = Math.max(...numeros);
console.log(max);
```

7. Destructuring Arrays/Objects

Cuando tenemos un array o un objeto con datos, y necesitamos sacar estos datos en variables independientes, teniamos que hacerlo de una en una.

```
let colores = ['Negro', 'Blanco'];
let negro = colores[0];
let blanco = colores[1];

let datosUsuario = {
  username: 'falco',
  password: 1234
};
let username = datosUsuario.username;
let password = datosUsuario.password;
```

Pero ahora ya podemos desestructurar un array u objeto para sacar los valores de una vez sin necesidad de escribir tanto código. Para ello solo tenemos que envolver las variables que van a recibir los valores entre los simbolos que representan al objeto del que van a salir estos valore, e igualarle el objeto u array.

En el caso de los objetos, las variables se tienen que llamar igual que las propiedades del objeto.

```
let colores = ['Negro', 'Blanco'];
let [negro, blanco] = colores;

let datosUsuario = {
  username: 'falco',
  password: 1234
};
let {username, password} = datosUsuario;
```

También podemos ignorar algún elemento al desestructurar un array, si no ponemos ninguna variable.

```
let colores = ['Negro', 'Blanco'];
let [, blanco] = colores;
```

Incluso podriamos obtener algunos elementos, y los elementos restantes guardarlos en una variable.

```
let colores = ['Negro', 'Blanco', 'Gris'];
let [negro, ...otros] = colores;
```

8. Clases

Hasta ahora en JavaScript las clases no se usaban como se hace en otros lenguajes porque no existían como tal, sino que se simulaban clases mediante las funciones.

Las clases son plantillas que nos van a permitir crear objetos. Todos los objetos que se crean con las clases van a tener las mismas propiedades y los mismos métodos. Las clases encapsulan una funcionalidad que se puede reutilizar y que se relaciona con una entidad (cualquier cosa).

```
class Serie {
  constructor(nombre, episodios, temporadas, episodiosVistos) {
    this.nombre = nombre;
    this.episodios = episodios;
    this.temporadas = temporadas;
    this.episodiosVistos = episodiosVistos;
  }
  episodiosPorVer() {
    return this.episodios - this.episodiosVistos;
  }
}

let vikings = new Serie('Vikings', 69, 5, 45);
console.log(vikings.episodiosPorVer());
console.log(vikings.finalizada());
```

Podemos añadir **métodos estáticos** añadiendo la palabra `static` delante del método. Y ahora no necesitamos una instancia del objeto para poder usar este método.

```
class Serie {
  constructor(nombre, episodios, temporadas, episodiosVistos) {
    this.nombre = nombre;
    this.episodios = episodios;
    this.temporadas = temporadas;
    this.episodiosVistos = episodiosVistos;
  }
  episodiosPorVer() {
    return this.episodios - this.episodiosVistos;
  }
  finalizada() {
    return this.episodiosPorVer() == 0;
  }
  static queEs() {
    return 'Serie';
  }
}

let vikings = new Serie('Vikings', 69, 5, 45);
console.log(Serie.queEs());
```

Ahora podemos hacer que una clase herede de otra de una forma más parecida a lo que estamos acostumbrados a ver en otros lenguajes.

Para indicar que una clase hereda de otra, solo hay que añadir `extends` seguido de la clase de la que va a heredar.

```
class Rectangulo {
  constructor(largo, ancho) {
    this.largo = largo;
    this.ancho = ancho;
  }
  getArea() {
    return this.largo * this.ancho;
  }
}

class Cuadrado extends Rectangulo {
  constructor(largo) {
    super(largo, largo);
  }
}

let cuadrado = new Cuadrado(3);
console.log(cuadrado.getArea());
```

Para añadir propiedades privadas, tenemos que declarar estas propiedades en el constructor usando la palabra **var**. Y para poder acceder a los datos o modificarlos, tendremos que crear unos

métodos que nos lo permitan.

```
class Persona {
  constructor(nombre, dni) {
    this.nombre = nombre;
    var _dni = dni;
    this.setDni = function (dni) {
      _dni = dni;
    }
    this.getDni = function () {
      return _dni;
    }
  }
}

let p = new Persona('Angel', 1111);
console.log(p.getDni());
p.setDni(22222);
console.log(p.getDni());
```

9. Modulos

Los módulos nos permiten agrupar bloques de código y exportarlos para poder usarlos en cualquier lugar de la aplicación cuando queramos.

Todo aquello que hemos puesto en un módulo solo va a poder ser usado en otros módulos si se está exportando, por lo tanto tendremos que indicar, que cosas hay que exportar con la palabra `export`. Hay dos formas de hacerlo.

```
export class Mascota {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
  }
  hazSonido () {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
```

```
class Mascota {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
  }
  hazSonido () {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}

export { Mascota };
```

Y para poder usar aquello que se está exportando desde un módulo, tendremos que importarlo en el archivo donde lo necesitemos usando la palabra `import` seguida de lo que se quiere importar y indicando desde que archivo hay que importarlo. A la hora de importarlo le podemos poner un alias a lo que se importa.

```
import { Mascota } from './mascota';

let perro = new Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```

Hay otra forma de importar módulos y es usando el asterisco, que importará todo aquello que se está exportando. A la hora de usar esta forma tendremos que darle un alias para poder acceder a las cosas que se están importando.

```
import * as masc from './mascota';

let perro = new masc.Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```

9.1. Exportando por defecto

Hay otra opción para exportar a parte de las vistas antes, y es la exportación por defecto que se usa cuando vamos a exportar solo una cosa del módulo. Aquí no hace falta ponerle nombre a lo que estamos exportando.

```
export default class {
  constructor(nombre, tipo, sonido) {
    this.nombre = nombre;
    this.tipo = tipo;
    this.sonido = sonido;
  }
  hazSonido() {
    console.log('El ' + this.tipo + ' hace ' + this.sonido);
  }
}
```

```
import Mascota from './mascota';

let perro = new Mascota('Toby', 'perro', 'guau');
perro.hazSonido();
```