UD02 - Arrays y programación Orientada a Objetos en Javascript

---

1. Arrays

1.1 Introducción

Son un tipo de objeto y no tienen tamaño fijo sino que podemos añadirle elementos en cualquier momento.

Podemos crearlos como instancias del objeto Array:

```
let a = new Array()         // a = []
let b = new Array(2,4,6)    // b = [2, 4, 6]
```

pero lo recomendado es crearlos usando notación JSON (recomendado):

```
let a = []
let b = [2,4,6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array. Si no está definido un elemento su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0])  // imprime 'Lunes'
console.log(a[4])  // imprime 6
a[7] = 'Juan'         // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7])  // imprime 'Juan'
console.log(a[6])  // imprime undefined
```

1.2 Operaciones con Arrays

Vamos a ver los principales métodos y propiedades de los arrays.

1.2.1 lenght

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a.length)  // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad:

```
a.length = 3  // ahora a = ['Lunes', 'Martes', 2]
```

1.2.2 Añadir elementos

Podemos añadir elementos al final de un array con `push` o al principio con `unshift`:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
a.push('Juan')    // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']
a.unshift(7)      // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

1.2.3 Eliminar elementos

Podemos borrar el elemento del final de un array con `pop` o el del principio con `shift`. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let ultimo = a.pop()          // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6
let primero = a.shift()       // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

1.2.4 splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3)        // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56)    // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56)    // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO 01_UD2: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz lo siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos Naranjas y Sandía (debe quedar Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos (debe quedar Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

1.2.5 slice

Devuelve un subarray con los elementos indicados pero sin modificar el array original (sería como hacer un `substr` pero de un array en vez de una cadena). Sintaxis:

```
Array.slice(posicion, num. de elementos a devolver)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3)        // ahora a = ['Lunes', 'Martes', 2, 4, 6] y subArray = ['Martes', 2, 4]
```

Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()        // ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

1.2.6 Arrays y Strings

Cada objeto (y los arrays son un tipo de objeto) tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola. En realidad `console.log(a)` ejecuta `console.log(a.toString())`. En el caso de los arrays esta función devuelve una cadena con los elementos del array dentro de corchetes y separados por coma.

Además podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-')        // cadena = 'Lunes-Martes-2-4-6'
```

Este método es el contrario del m `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-')         // arrayNotas = [5, 3.9, 6, 9.75, 7.5, 3]
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ')     // arrayPalabras = ['Que`, 'tal', 'estás']
let arrayLetras = cadena.split('')        // arrayLetras = ['Q','u','e`,' ','t',a',l',' ','e',s',t',á',s']
```

1.2.7 sort

Ordena **alfabéticamente** los elementos del array

```
let a = ['hola','adios','Bien','Mal',2,5,13,45]
let b = a.sort()        // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

También podemos pasarle una función que le indique cómo ordenar que devolverá un valor negativo si el primer elemento es mayor, positivo si es mayor el segundo o 0 si son iguales. Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola','adios','Bien','Mal']
let b = a.sort(function(elem1, elem2) {
  if (elem1.toLocaleLowerCase > elem2.toLocaleLowerCase)
    return -1
  if (elem1.toLocaleLowerCase < elem2.toLocaleLowerCase)
    return 1
  return 0
})       // b = ["adios", "Bien", "hola", "Mal"]
```

Como más se utiliza esta función es para ordenar arrays de objetos. Por ejemplo si tenemos un objeto *persona* con los campos *nombre* y *edad*, para ordenar un array de objetos persona por su edad haremos:

```
let personasOrdenado = personas.sort(function(persona1, persona2) {
  return persona1.edad-persona2.edad
})
```

Usando *arrow functions* quedaría más sencillo:

```
let personasOrdenado = personas.sort((persona1, persona2)  =>
persona1.edad-persona2.edad)
```

Si lo que queremos es ordenar por un campo de texto podemos usar la función *toLocaleCompare*:

```
let personasOrdenado = personas.sort((persona1, persona2)  =>
persona1.nombre.localeCompare(persona2.nombre))
```

EJERCICIO 02_UD2: Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10].

1.2.8 Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

- `.concat()`: concatena arrays

```
let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b)       // c = [2, 4, 6, 'a', 'b', 'c']
```

- `.reverse()`: invierte el orden de los elementos del array

```
let a = [2, 4, 6]
let b = a.reverse()       // b = [6, 4, 2]
```

- `.indexOf()`: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array
- `.lastIndexOf()`: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array

1.2.9 Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo. El ejemplo más claro es que intenta evitar los bucles *for* y *while* sobre arrays o listas de elementos. Normalmente cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programing* es que a la función que debe hacer eso además de pasarle como parámetro la lista sobre la que debe actuar se le pasa como segundo parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 javascript incorpora métodos de *functional programing* en el lenguaje, especialmente para trabajar con arrays:

1.2.9.1 filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica. Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array. Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar). Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo. La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado y **false** para el resto.

Ejemplo: dado un array con notas devolver un array con las notas de los aprobados. Esto usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0 i++ i < arrayNotas.length) {
  let nota = arrayNotas[i]
  if (nota > =  5) {
    aprobados.push(nota)
  }
}       // aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota > =  5) {
    return true
  } else {
    return false
  }
})       // aprobados = [5.2, 6, 9.75, 7.5]
```

Podemos refactorizar esta función para que sea más compacta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  return nota > =  5     // nota > =  5 se evalúa a 'true' si es cierto o 'false' si no lo es
})
```

Y usando funciones lambda la sintaxis queda mucho más simple:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(nota  => nota > =  5)
```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

> **EJERCICIO 03_UD2**: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

1.2.9.2 find

Como *filter* pero NO devuelve un **array** sino el primer **elemento** que cumpla la condición (o *undefined* si no la cumple nadie). Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let primerAprobado = arrayNotas.find(nota  => nota > =  5)     // primerAprobado = 5.2
```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar la persona con DNI '21345678Z' dentro de un array llamado personas cuyos elementos son objetos con un campo 'dni' haremos:

```
let personaBuscada = personas.find(persona  => persona.dni = = = '21345678Z')    // devolverá el objeto completo
```

EJERCICIO 04_UD2: Dado un array con los días de la semana obtén el primer día que empieza por 'M'

1.2.9.3 findIndex

Como *find* pero en vez de devolver el elemento devuelve su posición (o -1 si nadie cumple la condición). En el ejemplo anterior el valor devuelto sería 0 (ya que el primer elemento cumple la condición). Al igual que el anterior tiene más sentido con arrays de objetos.

EJERCICIO 05_UD2: Dado un array con los días de la semana obtén la posición en el array del primer día que empieza por 'M'

1.2.9.4 every / some

La primera devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario. La segunda devuelve **true** si **ALGÚN** elemento del array cumple la condición. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota  => nota > =  5)    // false
let algunAprobado = arrayNotas.some(nota  => nota > =  5)      // true
```

EJERCICIO 06_UD2: Dado un array con los días de la semana indica si algún día empieza por 'S'. Dado un array con los días de la semana indica si todos los días acaban por 's'

1.2.9.5 map

Permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados. Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = arrayNotas.map(nota  => nota + nota * 10%)
```

EJERCICIO 07_UD2: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas

1.2.9.6 reduce

Devuelve un valor calculado a partir de los elementos del array. En este caso la función recibe como primer parámetro el valor calculado hasta ahora y el método tiene como 1º parámetro la función y como 2º parámetro al valor calculado inicial (si no se indica será el primer elemento del array).

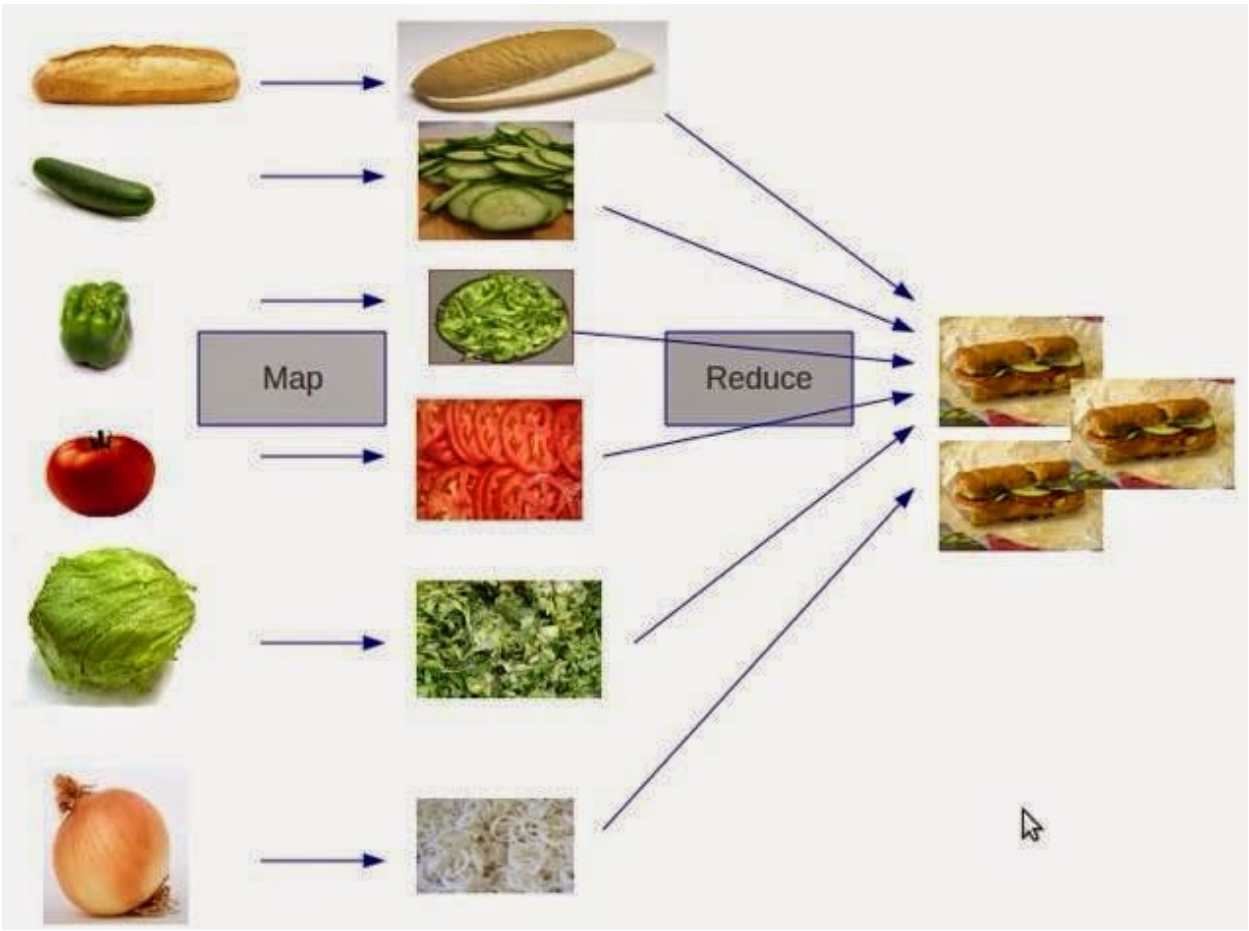
Ejemplo: queremos obtener la suma de las notas:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let sumaNotas = arrayNotas.reduce((total,nota)  => total + =  nota, 0)
 // total = 35.35
// podríamos haber omitido el valor inicial 0 para total
let sumaNotas = arrayNotas.reduce((total,nota)  => total + =  nota)    //
total = 35.35
```

Ejemplo: queremos obtener la nota más alta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota)  => nota > max ? nota : max)
 // max = 9.75
```

En el siguiente ejemplo gráfico tenemos un "array" de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sandwich) con todas ellas:



EJERCICIO 08_UD2: Dado el array de notas anterior devuelve la nota media

1.2.9.7 forEach

Es el método más general de los que hemos visto. No devuelve nada sino que permite realizar algo con cada elemento del array.

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.forEach((nota, indice)  => {
  console.log('El elemento de la posición ' + indice + ' es: ' + nota)
})
```

1.2.9.8 includes

Devuelve **true** si el array incluye el elemento pasado como parámetro. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.includes(7.5)     // true
```

EJERCICIO 09_UD2: Dado un array con los días de la semana indica si algún día es el 'Martes'

1.2.9.9 Array.from

Devuelve un array a partir de otro al que se puede aplicar una función de transformación (es similar a *map*). Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = Array.from(arrayNotas, nota  => nota + nota * 10%)
```

Puede usarse para hacer una copia de un array, igual que *slice*:

```
let arrayA = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayB = Array.from(arrayA)
```

También se utiliza mucho para convertir colecciones en arrays y así poder usar los métodos de arrays que hemos visto. Por ejemplo si queremos mostrar por consola cada párrafo de la página que comience por la palabra 'If' en primer lugar obtenemos todos los párrafos con:

```
let parrafos = document.getElementsByTagName('p')
```

Esto nos devuelve una colección con todos los párrafos de la página (lo veremos más adelante al ver DOM). Podríamos hacer un **for** para recorrer la colección y mirar los que empiecen por lo indicado pero no podemos aplicarle los métodos vistos aquí porque son sólo para arrays así que hacemos:

```
let arrayParrafos = Array.from(parrafos)
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo  => parrafo.textContent.startsWith('If'))
.forEach(parrafo  => alert(parrafo.textContent))
```

IMPORTANTE: desde este momento se han acabado los bucles *for* en nuestro código para trabajar con arrays. Usaremos siempre estas funciones!!!

1.3 Referencia vs Copia

Cuando copiamos una variable de tipo *boolean*, *string* o *number* o se pasa como parámetro a una función se hace una copia de la misma y si se modifica la variable original no es modificada. Ej.:

```
let a = 54
let b = a       // a = 54 b = 54
b = 86          // a = 54 b = 86
```

Sin embargo al copiar objetos (y los arrays son un tipo de objeto) la nueva variable apunta a la misma posición de memoria que la antigua por lo que los datos de ambas son los mismos:

```
let a = [54, 23, 12]
let b = a       // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3        // a = [3, 23, 12] b = [3, 23, 12]
let fecha1 = new Date('2018-09-23')
let fecha2 = fecha1          // fecha1 = '2018-09-23'   fecha2 = '2018-09-
23'
fecha2.setFullYear(1999)   // fecha1 = '1999-09-23'   fecha2 = '1999-09-
23'
```

Si queremos obtener una copia de un array que sea independiente del original podemos obtenerla con `slice` o con `Array.from`:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()       // ahora ambos arrays contienen lo mismo
pero son diferentes
let otraCopiaDeA = Array.fom(a)
```

En el caso de objetos es algo más complejo. ES6 incluye `Object.assign` que hace una copia de un objeto:

```
let a = {id:2, name: 'object 2'}
let copiaDeA = Object.assign({}, a)       // ahora ambos objetos contienen lo mismo pero son diferentes
```

Sin embargo si el objeto tiene como propiedades otros objetos estos se continúan pasando por referencia. Es ese caso lo más sencillo sería hacer:

```
let a = {id: 2, name: 'object 2', address: {street: 'C/ Ajo', num: 3} }
let copiaDeA =  JSON.parse(JSON.stringify(a))       // ahora ambos objetos contienen lo mismo pero son diferentes
```

> **PRUEBA 01_UD2**: Dado el array arr1 con los días de la semana haz un array arr2 que sea igual al arr1. Elimina de arr2 el último día y comprueba quá ha pasado con arr1. Repita la operación con un array llamado arr3 pero que crearás haciendo una copia de arr1.

También podemos copiar objetos usando *rest* y *spread*.

1.4 Rest y Spread

Permiten extraer a parámetros los elementos de un array o string (*spread*) o convertir en un array un grupo de parámetros (*rest*). El operador de ambos es **...** (3 puntos).

Para usar *rest* como parámetro de una función debe ser siempre el último parámetro:

```
function notaMedia(...notas) {
  let total = notas.reduce((total,nota)  => total + =  nota)
  return total/notas.length
}

console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3))    // le pasamos un número variable de parámetros
```

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*. Por ejemplo el objeto *Math* proporciona métodos para trabajar con números como *.max* que devuelve el máximo de los números pasados como parámetro. Para saber la nota máxima en vez de *.reduce* como hicimos en el ejemplo anterior podemos hacer:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = Math.max(...arrayNotas)    // maxNota = 9.75
// hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número
```

Estas funcionalidades nos ofrecen otra manera de copiar objetos (pero sólo a partir de ES-2018):

```
let a = {id: 2, name: 'object 2'}
let copiaDeA = { ...a}       // ahora ambos objetos contienen lo mismo pero son diferentes

let b = [2, 8, 4, 6]
let copiaDeB = [ ...a ]       // ahora ambos objetos contienen lo mismo pero son diferentes
```

1.5 Desestructuración de arrays

Similar a *rest* y *spread*, permiten extraer los elementos del array directamente a variables y viceversa. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let [primera, segunda, tercera] = arrayNotas   // primera = 5.2, segunda = 3.9, tercera = 6
let [primera, , , cuarta] = arrayNotas         // primera = 5.2, cuarta = 9.75
let [primera, ...resto] = arrayNotas           // primera = 5.2, resto = [3.9, 6, 9.75, 3]
```

También se pueden asignar valores por defecto:

```
let preferencias = ['Javascript', 'NodeJS']
let [lenguaje, backend = 'Laravel', frontend = 'VueJS'] = preferencias  // lenguaje = 'Javascript', backend = 'NodeJS', frontend = 'VueJS'
```

La desestructuración también funciona con objetos. Es normal pasar un objeto como parámetro para una función pero si sólo nos interesan algunas propiedades del mismo podemos desestructurarlo:

```
const miProducto = {
    id: 5,
    name: 'TV Samsung',
    units: 3,
    price: 395.95
}

muestraNombre(miProducto)

function miProducto({name: name, units: units}) {        // Mejor pondríamos function miProducto({name, units}) {
    console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}
```

También podemos asignar valores por defecto:

```
function miProducto({name, units = 0}) {
    console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre({name: 'USB Kingston')
// mostraría: Del producto USB Kingston hay 0 unidades
```

1.6 Map

Es una colección de parejas de [clave,valor]. Un objeto en Javascript es un tipo particular de *Map* en que las claves sólo pueden ser texto o números. Se puede acceder a una propiedad con **.** o **[propiedad]**. Ejemplo:

```
let persona = {
  nombre: 'John',
  apellido: 'Doe',
  edad: 39
}
console.log(persona.nombre)      // John
console.log(persona['nombre'])   // John
```

Un *Map* permite que la clave sea cualquier cosa (array, objeto, ...). No vamos a ver en profundidad estos objetos pero podéis saber más en MDN o cualquier otra página.

1.7 Set

Es como un *Map* pero que no almacena los valores sino sólo la clave. Podemos verlo como una colección que no permite duplicados. Tiene la propiedad **size** que devuelve su tamaño y los métodos **.add** (añade un elemento), **.delete** (lo elimina) o **.has** (indica si el elemento pasado se encuentra o no en la colección) y también podemos recorrerlo con **.forEach**.

Una forma sencilla de eliminar los duplicados de un array es crear con él un *Set*:

```
let ganadores = ['Márquez', 'Rossi', 'Márquez', 'Lorenzo', 'Rossi', 'Márquez', 'Márquez']
let ganadoresNoDuplicados = new Set(ganadores)    // {'Márquez, 'Rossi', 'Lorenzo'}
// o si lo queremos en un array:
let ganadoresNoDuplicados = Array.from(new Set(ganadores))    // ['Márquez, 'Rossi', 'Lorenzo']
```

2. Programación Orientada a Objetos (POO)

2.1 Introducción

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes: clases, herencia, ...

Se pueden crear con **new** o creando un *literal object* (usando notación **JSON**). Con *new*:

```
let alumno = new Object()
alumno.nombre = 'Carlos'     // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz'    // se crea la propiedad 'apellidos'
alumno.edad = 19
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + 'años'
}
```

Creando un *literal object* (es la forma recomendada) el ejemplo anterior sería:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: function() {
        return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + 'años'
    }
};
```

2.2 Propiedades de un objeto

Podemos acceder a las propiedades con `.` (punto) o `[ ]`:

```
console.log(alumno.nombre)        // imprime 'Carlos'
console.log(alumno['nombre'])     // imprime 'Carlos'
let prop = 'nombre'
console.log(alumno[prop])         // imprime 'Carlos'
console.log(alumno.getInfo())     // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años'
```

Podremos recorrer las propiedades de un objecto con `for..in`:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

En ES6 si el valor de una propiedad es una función podemos ponerlo como:

```
    ...
    getInfo() {
        return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + 'años'
    }
    ...
```

o en forma de *arrow function*:

```
    ...
    getInfo: () => 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + 'años'
    ...
```

Además si queremos que el valor de una propiedad sea el de una variable que se llama como ella desde ES2015 no es necesario ponerlo:

```
let nombre = 'Carlos'

let alumno = {
    nombre,                          // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
```

PRUEBA 02_UD2: Crea un objeto llamado tvSamsung con las propiedades nombre (TV Samsung 42"), categoria (Televisores), unidades (4), precio (345.95) y con un método llamado importe que devuelve el valor total de las unidades (nº de unidades * precio)

2.3 Clases

Desde ES2015 funcionan como en la mayoría de lenguajes:

```
class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre
        this.apellidos = apellidos
        this.edad = edad
    }
    getInfo() {
        return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(cpo.getInfo())     // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años'
```

PRUEBA 03_UD2: Crea una clase Productos con las propiedades y métodos del ejercicio anterior. Además tendrá un método getInfo que devolverá: 'Nombre (categoría): unidades uds x precio € = importe €'. Crea 3 productos diferentes.

2.3.1 Ojo con *this*

Dentro de una función se crea un nuevo contexto y la variable *this* pasa a hacer referencia a dicho contexto. Si en el ejemplo anterior hiciéramos algo como esto:

```
class Alumno {
    ...
    getInfo() {
        return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años'
        function nomAlum() {
            return this.nombre + ' ' + this.apellidos      // Aquí this no es el objeto Alumno
        }
    }
}
```

Este código fallaría porque dentro de *nomAlum* la variable *this* ya no hace referencia al objeto Alumno sino al contexto de la función. Este ejemplo no tiene mucho sentido pero a veces nos pasará en

manejadores de eventos. Si debemos llamar a una función dentro de un método (o de un manejador de eventos) tenemos varias formas de pasarle el valor de *this*:

1. Pasárselo como parámetro

```
    getInfo() {
        return 'El alumno ' + nomAlum(this) +' tiene ' + this.edad + ' años'
        function nomAlum(alumno) {
            return alumno.nombre + ' ' + alumno.apellidos
        }
    }
```

2. Guardando el valor en otra variable (como *that*)

```
    getInfo() {
        let that = this;
        return 'El alumno ' + nomAlum() +' tiene ' + this.edad + ' años'
        function nomAlum() {
            return that.nombre + ' ' + that.apellidos      // Aquí this no es el objeto Alumno
        }
    }
```

3. Usando una *arrow function* que no crea un nuevo contexto por lo que *this* conserva su valor

```
    getInfo() {
        return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años'
        let nomAlum = () => this.nombre + ' ' + this.apellidos
    }
```

4. Haciendo un *bind* de *this* (lo varemos al hablar de eventos)

Esto nos puede ocurrir en las funciones manejadoras de eventos que veremos en próximos temas.

2.3.2 Herencia

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos. Podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** -es lo que haremos si creamos un constructor en la clase hija-).

```
class AlumnInf extends Alumno{
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad)
```

```
    this.ciclo = ciclo
  }
  getInfo() {
    return super.getInfo() + ' y estudia el Grado ' + (this.getGradoMedio
? 'Medio' : 'Superior') + ' de ' + this.ciclo
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase === 'SMX')
      return true
    return false
  }
}

let cpo = new AlumnInf('Carlos', 'Pérez Ortiz', 19, 'DAW')
console.log(cpo.getInfo())     // imprime 'El alumno Carlos Pérez Ortíz
tiene 19 años y estudia el Grado Superior de DAW'
```

PRUEBA 04_UD2: crea una clase Televisores que hereda de Productos y que tiene una nueva propiedad llamada tamaño. El método getInfo mostrará el tamaño junto al nombre

2.3.3 Métodos estáticos

En ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente utilizando el nombre de la clase y no tienen acceso al objeto *this* (ya que no hay objeto instanciado).

```
class User {
    ...
    static getRoles() {
        return ["user", "guest", "admin"]
    }
}

console.log(User.getRoles()) // ["user", "guest", "admin"]
let user = new User("john")
console.log(user.getRoles()) // Uncaught TypeError: user.getRoles is not
a function
```

2.3.4 toString() y valueOf()

Al convertir un objeto a string (por ejemplo al concatenarlo con un String) se llama al método ***.toString()*** del mismo, que devuelve `[object Object]`. Podemos sobrecargar este método para que devuelva lo que queramos:

```
class Alumno {
    ...
    toString() {
        return this.apellidos+', '+this.nombre
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
console.log('Alumno:' + cpo)     // imprime 'Alumno: Pérez Ortíz, Carlos'
```

Este método también es el que se usará si queremos ordenar una array de objetos (recordad que *.sort()* ordena alfabéticamente para lo que llama al método *.toString()* del objeto a ordenar). Por ejemplo, tenemos el array de alumnos *misAlumnos* que queremos ordenar alfabéticamente. Si la clase *Alumno* no tiene un método *toString* habría que hacer como vimos en el tema de Arrays:

```
misAlumnos.sort(function(alum1, alum2) {
    if (alum1.apellidos > alum2.apellidos)
      return -1
    if (alum1.apellidos < alum2.apellidos)
      return 1
    return alum1.nombre < alum2.nombre
});
```

(nota: si las cadenas a comparar pueden tener acentos u otros caracteres propios del idioma deberíamos usar el
método `.localeCompare()` para comparar las cadenas)

Pero con el método *toString* que hemos definido antes podemos hacer directamente:

```
misAlumnos.sort()
```

Al comparar objetos (con >, <, ...) se usa el valor devuelto por el método *.toString()* pero si sobrecargamos el método ***.valueOf()*** será este el que se usará en comparaciones:

```
class Alumno {
    ...
    valueOf() {
        return this.edad
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
let aat = new Alumno('Ana', 'Abad Tudela', 23)
console.log(cpo < aat)     // imprime true ya que 19<23
```

PRUEBA 05_UD2: modifica las clases Productos y Televisores para que el método que muestra los datos del producto se llame de una manera más adecuada

EJERCICIO 10_UD2: Crea 5 productos y guárdalos en un array. Crea las siguientes funciones (todas reciben ese array como parámetro):

- prodsSortByName: devuelve un array con los productos ordenados alfabéticamente
- prodsSortByPrice: devuelve un array con los productos ordenados por importe
- prodsTotalPrice: devuelve el importe total del los productos del array, con 2 decimales
- prodsWithLowUnits: además del array recibe como segundo parámetro un nº y devuelve un array con todos los productos de los que quedan menos de los unidades indicadas
- prodsList: devuelve una cadena que dice 'Listado de productos:' y en cada línea un guión y la información de un producto del array

2.4 POO en ES5

Las versiones de Javascript anteriores a ES2015 no soportan clases ni herencia. Este apartado está sólo para que comprendamos este código si lo vemos en algún programa pero nosotros programaremos como hemos visto antes.

En Javascript un objeto se crea a partir de otro (al que se llama *prototipo*). Así se crea una cadena de prototipos, el primero de los cuales es el objeto *null*.

Si queremos emular el comportamiento de las clases, para crear el constructor se crea una función con el nombre del objeto y para crear los métodos se aconseja hacerlo en el *prototipo* del objeto para que no se cree una copia del mismo por cada instancia que creemos:

```
function Alumno(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
}
Alumno.prototype.getInfo = function() {
        return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(cpo.getInfo())     // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años'
```

Cada objeto tiene un prototipo del que hereda sus propiedades y métodos (es el equivalente a su clase, pero en realidad es un objeto que está instanciado). Si añadimos una propiedad o método al prototipo se añade a todos los objetos creados a partir de él lo que ahorra mucha memoria.

Bibliografía

- Curso 'Programación con JavaScript'. CEFIRE Xest. Arturo Bernal Mayordomo
- Curso de JavaScript y TypeScript de Arturo Bernal en Youtube
- MDN Web Docs. Moz://a. https://developer.mozilla.org/es/docs/Web/JavaScript
- Introducción a JavaScript. Librosweb. http://librosweb.es/libro/javascript/
- Curso de Javascript (Desarrollo web en entorno cliente). Ada Lovecode - Didacticode (90 vídeos)
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). García Barea, Sergi
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). Segura Vasco, Juan. CIPFP Batoi.