

EXERCISES QUE TRABAJAREMOS EN EL CUE

- EXERCISE 1: INTRODUCCIÓN A DESTRUCTURING Y EL OPERADOR SPREAD.
- EXERCISE 2: SETS Y MAPS EN ES6.

EXERCISE 1: INTRODUCCIÓN A DESTRUCTURING Y EL OPERADOR SPREAD

En JavaScript usamos objetos para almacenar múltiples valores, como una estructura de datos compleja. Actualmente, ya son escasas las aplicaciones JavaScript que no ocupen los objetos.

Durante nuestros desarrollos, comúnmente extraemos valores desde las propiedades de un objeto, para usarlos en otras partes de los programas. Con ES6, JavaScript introdujo la **desestructuración** de objetos, para así facilitar la creación de variables a partir de las propiedades de uno.

En síntesis, **Destructuring** o la Desestructuración, es la sintaxis para extraer valores desde las propiedades de un objeto o de un **array**, y asignarlos a una variable. Veamos este concepto en práctica con un ejemplo.

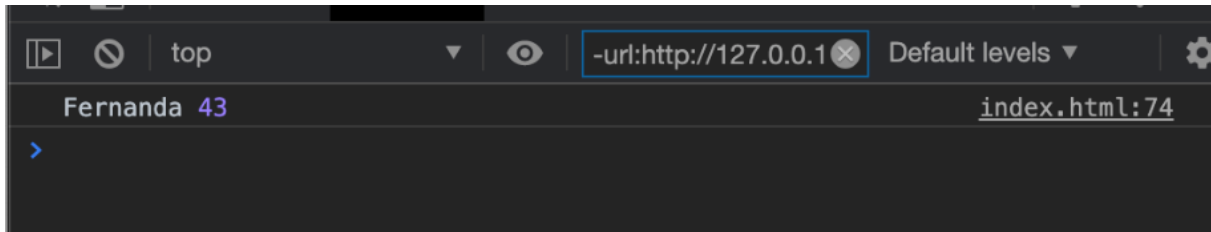
Supongamos que tenemos un objeto **Persona**:

```
1  const persona = {  
2      'nombre': 'Fernanda',  
3      'direccion': 'Avenida Central',  
4      'edad': 43  
5  }
```

Si quisiéramos extraer los valores de los atributos de este objeto para usarlos por separado, tendríamos que hacer algo similar a lo siguiente:

```
1  let nombre = persona.nombre;  
2  let edad = persona.edad;  
3  console.log(nombre, edad);
```

De esta forma, podríamos trabajar con los valores de **nombre** y **edad** por separado. Lo comprobaremos mediante la consola que muestra lo siguiente:



Ahora, nota la siguiente manera de llegar a exactamente el mismo resultado con el uso de **Destructuring**.

```
1 const { nombre, edad } = persona;  
2 console.log(nombre, edad);
```

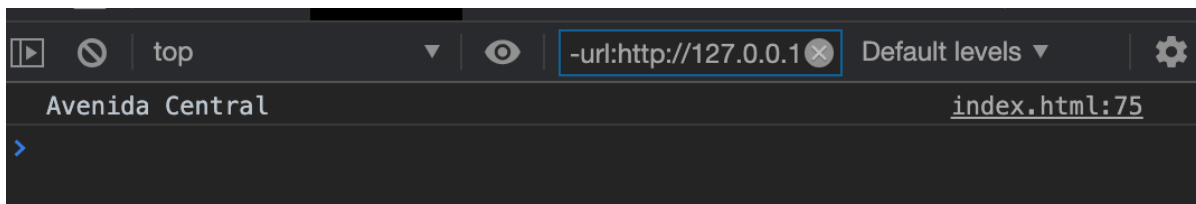
Al ver el resultado en nuestra consola, notaremos que fue exactamente el mismo, pero esta vez, con una sola línea de código.

¿Cómo funciona? En el lado izquierdo de la expresión, seleccionamos la clave de propiedad del objeto que deseamos extraer (en este caso `nombre`), y la colocamos dentro de las llaves `{ }`. Mientras que en el lado derecho de la expresión es donde especificamos el objeto del cual queremos extraer los valores.

Una forma alternativa de escribir la sintaxis de desestructuración es la que se muestra en el siguiente código:

```
1 let direccion  
2 ({direccion} = persona);  
3 console.log(direccion);
```

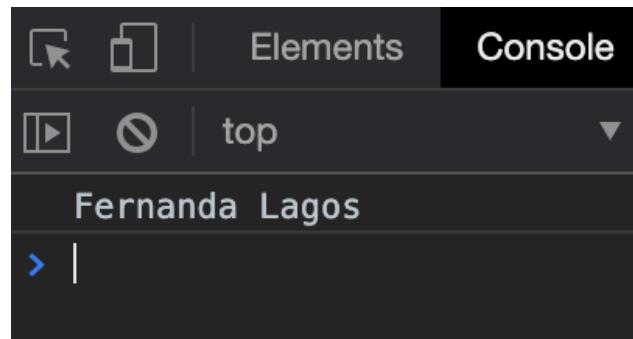
Aquí se parte especificando el objeto de destino que almacenará el valor de la propiedad “dirección”, obtenida del objeto `persona`. Es importante destacar que en esta sintaxis la declaración `let` y la desestructuración deben estar en distintas líneas, para que JS no lo interprete como una declaración de función. El resultado por consola será:



Ahora bien, Destructuring no solo sirve para extraer el valor de una propiedad, sino que también nos sirve para poder concatenar propiedades existentes con propiedades nuevas. Es importante especificar que las propiedades nuevas no son agregadas al objeto de destino, pues el Destructuring siempre extrae valores, no los puede añadir. Por ejemplo:

```
1 const {  
2     nombreCompleto = `${persona.nombre} Lagos`  
3 } = persona;  
4 console.log(nombreCompleto);
```

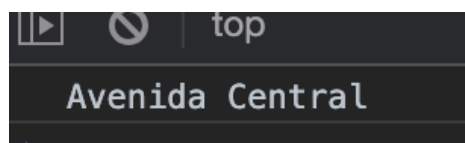
En nuestra consola el resultado es el siguiente:



Otra característica que proporciona el Destructuring es la posibilidad de poder dar un alias a las variables desestructuradas. Esto es especialmente útil si queremos evitar conflictos por los nombres de las variables. Considere el siguiente caso, donde asignamos el alias “domicilioLaboral” al atributo extraído dirección:

```
1 const {  
2     direccion: domicilioLaboral  
3 } = persona;  
4 console.log(domicilioLaboral);
```

El resultado en nuestro navegador es el siguiente:



También podemos aplicar Destructuring a objetos anidados. Por ejemplo, supongamos que tenemos el siguiente objeto que refleja una situación de la vida real:

```
1 const empleado = {
2   'nombre': 'Rodrigo',
3   'direccion': 'Pasaje Alba 324',
4   'edad': 32,
5   'departamento': {
6     'nombre': 'Ventas',
7     'turno': 'Mañana',
8     'direccion': {
9       'ciudad': 'Punta Arenas',
10      'calle': 'Calle Industrial 1020'
11    }
12  }
13 }
```

En este objeto tenemos un atributo departamento, cuyo valor es otro objeto, dentro del cual se encuentra el atributo dirección, que también es otro objeto. Si queremos extraer **dirección** de **departamento**, la sintaxis es colocar el nombre del que está en el exterior, seguido por dos puntos **“:”**, y luego el otro que está anidado entre llaves **{ }**.

```
1 const {
2   departamento,
3   departamento: {
4     direccion
5   }
6 } = empleado
7 console.log(departamento)
8 console.log(direccion)
```

En nuestra consola los dos objetos se ven así, logrando extraer ambos de manera exitosa:

```
index.html:87
{nombre: "Ventas", Turno: "Mañana", direccion: {...}} ⓘ
  Turno: "Mañana"
  ▶ direccion: {ciudad: "Punta Arenas", Calle: "Calle Industrial 1020"}
  nombre: "Ventas"
  ▶ __proto__: Object

index.html:88
{ciudad: "Punta Arenas", Calle: "Calle Industrial 1020"} ⓘ
  Calle: "Calle Industrial 1020"
  ciudad: "Punta Arenas"
  ▶ __proto__: Object
```

La última característica del Destructuring que vamos a analizar, corresponde a cómo usarlo en funciones. Por ejemplo, supongamos que tenemos un método que muestra por consola un mensaje similar: “Fernanda tiene 43 años y su hobby favorito es pintar”. Vamos a implementar dicho método usando el objeto persona como parámetro. Plantearemos el siguiente desarrollo:

```
1 const persona = {
2   'nombre': 'Fernanda',
3   'direccion': 'Avenida Central',
4   'edad': 43,
5   'hobby': 'pintar',
6   'oficio': 'desarrolladora Front End'
7 }
8 const infoSobrePersona = ({
9   nombre,
10  edad,
11  hobby
12 }) => {
13   console.log(`${nombre} tiene ${edad} años y su hobby favorito es
14   ${hobby}.`)
15 }
16 // Pasamos un objeto como parámetro:
17 infoSobrePersona(persona);
```

Si te fijas en la declaración del método, podrás ver que por parámetro tenemos entre llaves los atributos que queremos extraer desde el objeto persona. En este tipo de desestructuración, dicho objeto es pasado por parámetros, y no en la misma declaración del destructuring. Todo este desarrollo resulta en lo siguiente:



```
Fernanda tiene 43 años y su hobby favorito es pintar.
```

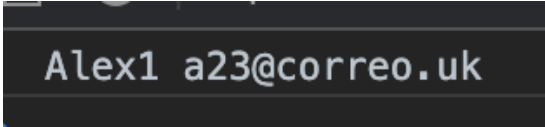
Ahora, cuando una función retorna un objeto y nosotros queremos usar los valores de las propiedades de ese objeto retornado, podemos usar la desestructuración para trabajarlos.

```
1 // método:
2 const obtenerUsuario = () => {
3   return {
4     'usuario': 'Alex1',
5     'correo': 'a23@correo.uk',
6     'edad': 22
7   }
8 }
```



```
7      }  
8  }  
9  // Desestructuramos el objeto retornado por el método:  
10 const {  
11     usuario,  
12     correo  
13 } = obtenerUsuario();  
14 console.log(usuario, correo);
```

Nuestra consola:



```
Alex1 a23@correo.uk
```

Ya que hemos analizado de manera completa el concepto de desestructuración, a continuación, estudiaremos otro elemento nuevo llamado: el operador **spread**.

En castellano, la palabra **Spread** puede traducirse a *propagar* o *expandir*. Tener esto en cuenta nos ayudará a explicar que éste *expande* de forma iterable a sus elementos individuales. Su sintaxis consta de solo tres puntos `...`.



```
function func(...argumentos){  
  
}
```

Para entenderlo mejor, vamos a plantear el siguiente ejercicio:

```
1 const suma = (x, y, z) => x + y + z;  
2 let numeros = [7, 8, 9]  
3 // Manera ES5:  
4 console.log(suma.apply(null, numeros)); //24  
5 // Manera ES6 con Spread:  
6 console.log(suma(...numeros)); //24
```

Si lo notas, antiguamente teníamos que depender de un método `apply()`, para poder incorporar los elementos del arreglo `numeros` como parámetro en el método `suma()`. Ahora podemos usar el operador `spread` `...`, antepuesto al nombre del arreglo, para poder usar todos sus elementos como parámetros.

En este punto, se podría estar pensando que ya hemos visto esto antes con el operador Rest, pero no es el caso. El operador Rest puede parecer similar, pero en términos de funcionalidad, hace exactamente lo contrario.

El operador Rest toma varios elementos individuales, y los “condensa o junta” en una matriz. Mientras que el operador Spread hace exactamente lo contrario, permitiéndonos tomar un elemento “condensado” como una matriz o un objeto, y solo usar dicho elemento por sus partes individuales, no como un todo.

Éste nos facilita muchas acciones que podemos hacer sobre los arreglos. Por ejemplo, nos permite copiarlos:

```
1 var a = [1, 2, 3, 4, 5]
2 var b = [6, 7, 8, 9, 10]
3 a = [...b]
4 console.log(a) // [6,7,8,9,10]
```

También, nos permite concatenar dos arreglos sin la necesidad del método `concat()`:

```
1 var a = [1, 2, 3, 4, 5]
2 var b = [6, 7, 8, 9, 10]
3 // Manera ES5:
4 console.log(a.concat(b)) // [1,2,3,4,5,6,7,8,9,10]
5 // Manera ES6 con Spread:
6 console.log([...a, ...b]) // [1,2,3,4,5,6,7,8,9,10]
```

Como podemos ver, basta con incluir dos Spread dentro de un arreglo para concatenarlos. De esta forma, hemos analizado cómo usar dos nuevos elementos de la revisión ES6: **Destructuring** y el operador **Spread**.

EXERCISE 2: SETS Y MAPS EN ES6

ES6 proporciona dos nuevos tipos o estructuras de datos, llamados: `Set` y `Map`. Vamos a partir analizando la primera estructura llamada `Set`, este objeto cumple con el objetivo de almacenar una colección de valores **únicos** de cualquier tipo de dato.

Para crear un nuevo **Set** vacío, se utiliza la siguiente sintaxis:

```
1 let objetoSet = new Set(objetoIterable);
```

En el constructor del **Set**, el objeto iterable es completamente opcional. Cuando se incluye, todos los elementos del objeto iterable se agregan al nuevo **Set**.

A este objeto se le pueden aplicar los siguientes métodos:

- **add(valor)**: agrega un nuevo elemento con un valor especificado al conjunto. Devuelve el objeto **Set**, por lo tanto, puede encadenar este método con otro.
- **delete(valor)**: elimina un elemento especificado por el valor.
- **clear()**: elimina todos los elementos del objeto **Set**.

Estos no son los únicos métodos disponibles. A continuación, pondremos en práctica este objeto, junto a otros métodos disponibles para ver el alcance de lo que podemos lograr con el **Set**.

Empezaremos viendo cómo crear un **Set** a partir de un **Array**:

```
1 let caracteres = new Set(['x', 'x', 'y', 'z', 'z']);  
2 console.log(caracteres);
```

Nuestro **Set** contiene un **array** con 5 letras, sin embargo, como éstos solo pueden contener valores únicos, el objeto descartará los que están repetidos y solo retornará 3, que son los que corresponden:

```
► Set(3) {"x", "y", "z"}
```

Podemos incluso confirmar esto usando el método **size()** para los **Sets**, considere el siguiente ejemplo:

```
1 console.log(caracteres);  
2 console.log(caracteres.size);
```

Como podemos ver en nuestra consola, efectivamente el Set solo almacenó los elementos únicos, dado que su tamaño es 3, el cual refleja la cantidad de elementos que hay:


```
► Set(3) {"x", "y", "z"}
3
```

Para eliminar un elemento de un Set, se usa el método `delete()`. La siguiente declaración elimina el valor `'y'` del `Set caracteres`:

```
1 let caracteres = new Set(['x', 'x', 'y', 'z', 'z']);
2 caracteres.delete('y')
3 console.log(caracteres);
```

El resultado es el siguiente:

```
▼ Set(2) {"x", "z"} ⓘ
  ▼ [[Entries]]
    ► 0: "x"
    ► 1: "z"
    size: (...)
    ► __proto__: Set
```

Como podemos ver, ahora ya no existe el elemento único `'y'`.

Ahora bien, para agregar un elemento al Set, se usa el método `add()`, y dado que éste se puede encadenar, puede agregar varios elementos a un conjunto usando una declaración en cadena:

```
1 let caracteres = new Set(['x', 'x', 'y', 'z', 'z']);
2 caracteres.add(1).add(2).add(3);
3 console.log(caracteres);
```

El resultado de la declaración en cadena es el siguiente:

```
▼ Set(6) {"x", "y", "z", 1, 2, ...} ⓘ
  ▼ [[Entries]]
    ▶ 0: "x"
    ▶ 1: "y"
    ▶ 2: "z"
    ▶ 3: 1
    ▶ 4: 2
    ▶ 5: 3
```

En esta imagen podemos notar un detalle importante, un objeto **Entries** o “entradas”. Si se lo aplicamos al **Set**, éste nos retornará un nuevo iterador que contiene una matriz **[valor, valor]**:

```
1 console.log(caracteres.entries());
```

Al aplicar este método, el resultado es:

```
SetIterator {"x" => "x", "y" => "y", "z" => "z", 1 => 1, 2 => 2, ...}
Index.html:67
```

Para verificar si un Set contiene un elemento específico, se usa el método **has()**. Éste devuelve **true** si el **Set** contiene el elemento, y de lo contrario, devuelve **false**. Considere el siguiente ejemplo:

```
1 let caracteres = new Set(['x', 'x', 'y', 'z', 'z']);
2 console.log(caracteres.has('z')); //verdadero
3 console.log(caracteres.has('w')); //falso
```

Dado que el **Set caracteres** contiene una **'z'**, esta declaración devuelve **true**, mientras que la otra retorna **false**:

```
true
false
```

Dado que ya hemos analizado varios métodos que podemos aplicar sobre un **Set**, ahora revisaremos el concepto de un **WeakSet**.

Un **WeakSet** es similar a un **Set**, excepto que solo contiene objetos. Éste es débil, lo que significa que las referencias a objetos dentro de él se mantienen así. Si no existen otras referencias a un objeto almacenado en el **WeakSet**, esos objetos se pueden recolectar como basura. El recolector de basura, o simplemente el recolector, intenta recuperar la memoria ocupada por objetos que el programa ya no usa. Por esa razón, **WeakSet** no tiene la propiedad de tamaño. De hecho, solo usa uno para verificar si un valor específico está en el conjunto.

La sintaxis de un **WeakSet**, junto con el método para ver si contiene un valor, es el siguiente:

```
1 let computador = {
2     tipo: 'laptop'
3 };
4 let servidor = {
5     tipo: 'servidor'
6 };
7 let equipment = new WeakSet([computador, servidor]);
8 if(equipment.has(servidor)) {
9     console.log('Tenemos un servidor');
10 }
```

El resultado en nuestro navegador es el siguiente:



Ahora que hemos visto los Sets, continuaremos viendo los **Maps** o “mapas”. Antes de ES6, cuando se necesitaba asignar claves a valores, a menudo se usaba un objeto, pues nos permite asignar una clave a un valor de cualquier tipo. Sin embargo, usar un objeto como mapa tiene algunas limitaciones, tales como:

- Un objeto siempre tiene una clave predeterminada como el prototipo.
- La clave de un objeto debe ser una cadena o un símbolo, no puede usar un objeto como clave.
- Un objeto no tiene una propiedad que represente el tamaño del mapa.

El tipo de colección **Map** aborda estas deficiencias. Por definición, un objeto **Map** contiene pares clave - valor, donde podemos usar cualquier tipo de dato como claves o valores. Además, éste recuerda el orden de inserción original de las claves.

La sintaxis de los **Maps** es la siguiente:

```
1 let mapa = new Map([iterable]);
```

Map() puede aceptar un objeto iterable cuyos elementos son pares clave - valor, pero este elemento es completamente opcional. Tiene métodos muy similares al **Set**. A continuación, analizaremos algunos de ellos.

Para empezar, vamos a plantear una lista de objetos que corresponden a usuarios, a los cuales les asignaremos roles usando el **Map** que declaramos a continuación:

```
1 //lista de objetos usuarios:
2 let juan = {
3     nombre: 'Juan Díaz'
4 },
5 lily = {
6     nombre: 'Lily Silva'
7 },
8 pedro = {
9     nombre: 'Pedro Salamanca'
10 };
11 let rolesUsuario = new Map(); //Nuevo Map
```

Para asignar un rol a un usuario, usamos el método **set()**:

```
1 rolesUsuario.set(juan, 'Administrador'); //asigna al usuario Juan el rol de
2 Administrador
```

Éste se puede encadenar, por lo que podemos asignar valores a cada uno de los miembros del **Map**:

```
1 userRoles.set(juan, 'Administrador') //asigna al usuario Juan el rol de
2 Administrador
3     .set(lily, 'Jefa de Area').set(pedro, 'Técnico');
```

Podríamos haber inicializado el Mapa con la misma información, usando un objeto iterable:

```
1 let rolesUsuario = new Map([
2     [juan, 'Administrador'],
3     [lily, 'Jefa de Area'],
4     [pedro, 'Técnico']
5 ]);
```

Para ver el rol de Juan, usamos el método **get()**; pero si buscamos un valor no existente, obtendremos **undefined**:

```
1 // Usando get() para obtener el rol
2 console.log(rolesUsuario.get(juan));
3 // Usando get() para obtener un valor no existente:
```



```
4 let noExiste = {  
5     nombre: 'abc'  
6 }  
7 console.log(rolesUsuario.get(noExiste));
```

En nuestra consola vemos el siguiente resultado:

```
Administrador      index.html:79  
undefined          index.html:84
```

Podemos comprobar la existencia de los objetos mediante el método `has()`:

```
1 console.log(rolesUsuario.has(lily));  
2 console.log(rolesUsuario.has(noExiste));
```

Estas líneas de código resultan en lo siguiente, indicando que el objeto `Lily` existe, mientras que `noExiste` realmente no existe:

```
true      index.html:86  
false     index.html:87
```

Para verificar la cantidad de objetos dentro de un `Map`, debemos usar la *propiedad* `size` (note que no es un método):

```
1 console.log(rolesUsuario.size);
```

La consola nos devuelve el valor:

```
3
```

Para obtener las claves de un objeto `Map`, se utiliza el método `keys()`, el cual devuelve un nuevo objeto iterador que contiene las claves de los elementos en el mapa.

Teniendo las claves, ya tenemos los objetos ingresados en el mapa. El siguiente ejemplo muestra el nombre de todos los usuarios en el mapa `rolesUsuario`.

```
1 for(let usuario of rolesUsuario.keys()) {  
2     console.log(usuario.nombre);  
3 }
```

Al iterar por el valor del nombre de usuario, tenemos el siguiente resultado:

Juan Diaz	index.html:90
Lily Silva	index.html:90
Pedro Salamanca	index.html:90

> |

De manera similar, podemos usar el método `values()`, para obtener un objeto iterador que contenga los valores de todos los elementos en el mapa:

```
1 for(let rol of rolesUsuario.values()) {  
2     console.log(rol);  
3 }
```

Con este método accedemos a lo siguiente por consola:

Administrador	index.html:90
Jefa de Area	index.html:90
Técnico	index.html:90

Además, el método `entries()` devuelve un objeto iterador que contiene una matriz `[clave, valor]` de cada elemento en el objeto `Map`:

```
1 for(let elementos of rolesUsuario.entries()) {  
2     console.log(`${elementos[0].nombre}: ${elementos[1]}`);  
3 }
```

En nuestra consola podremos apreciar lo siguiente, mostrando toda la información de nuestro `Map`:

Juan Diaz: Administrador	index.html:90
Lily Silva: Jefa de Area	index.html:90
Pedro Salamanca: Técnico	index.html:90

> |

Tal como se puede hacer con el objeto `Set`, `Map` utiliza los métodos `delete()` y `clear()`, para eliminar un valor y borrar todos los valores respectivamente.

Un `WeakMap` es similar a un `Map`, excepto que las claves de éste deben ser objetos (tal como sucede con los `WeakSets`). Esto significa que cuando una referencia a una clave (un objeto) está fuera de alcance, el valor correspondiente se libera automáticamente de la memoria del motor de JavaScript.

Un `WeakMap` solo tiene un subconjunto de los métodos del `Map`:

- `get()` la clave.
- `set (clave, valor)`
- `has(clave)`
- `delete(clave)`

Aquí están las principales diferencias entre un mapa y un `WeakMap`:

- Los elementos de un `WeakMap` no se pueden iterar.
- No se pueden borrar todos los elementos a la vez.
- No se puede comprobar el tamaño de un `WeakMap`.

Considere el siguiente ejemplo de un `WeakMap`:

```
1  let juan = {
2      nombre: 'Juan Diaz'
3  },
4  lily = {
5      nombre: 'Lily Silva'
6  },
7  pedro = {
8      nombre: 'Pedro Salamanca'
9  };
10 const mapaDebil = new WeakMap();
11 mapaDebil.set(juan, 'primero').set(lily, 'segunda').set(pedro,
12 'tercero');
13 console.log(mapaDebil)
```

Esto en nuestra consola se ve así:



```
index.html:73
▼ WeakMap {...} => "primero", {...} => "tercero", {...} => "segunda" ⓘ
  ▼ [[Entries]]
    ▼ 0: {Object => "primero"}
      ▶ key: {nombre: "Juan Diaz"}
      value: "primero"
    ▼ 1: {Object => "tercero"}
      ▶ key: {nombre: "Pedro Salamanca"}
      value: "tercero"
    ▼ 2: {Object => "segunda"}
    ▶ __proto__: WeakMap
```

De esta forma, queda demostrado cómo utilizar los **Sets** y **Maps** en nuestro desarrollo con ES6.