4.4.9 DESESTRUCTURACIÓN DE ARRAYS

La versión estándar ES2015 del lenguaje trajo una capacidad muy interesante relacionada con los arrays, pero que realmente lo que hace es añadir facilidades para manipular varios datos a la vez y recogerles en las estructuras idóneas. No afecta solo a los arrays, pero en este apartado nos centraremos en la sintaxis de desestructuración relacionada con los arrays.

Una primera capacidad que aporta es la de asignar valores a varias variables a la vez. Veámoslo con un ejemplo:

```
let [saludo,despedida,cierre]=["Hola","Adiós","Hasta nunca"];
console.log(saludo);   //Escribe Hola
console.log(despedida);//Escribe Adiós
console.log(cierre);   //Escribe Hasta nunca
```

Como vemos, se asignan de golpe valores para tres variables.

Las expresiones que se asignan pueden ser tan complejas como queramos:

```
let [n1,n2]=[10,Math.random()*20];
```

Incluso se facilita hacer la operación clásica de ***swap*** de variables. Es decir, intercambiar los valores de dos variables:

```
[a,b]=[b,a];
```

La variable ***a*** valdrá lo que valía ***b*** y viceversa.

Hay también un operador llamado operador de propagación (**spread** en inglés) que permite convertir un array en variables y viceversa. Este operador se usa con tres puntos seguidos (...) Ejemplo:

```
let array=[1,2,3];
let [x,y,z]=[...array];
```

La variable ***x*** valdrá uno, ***y*** valdrá dos y ***z*** valdrá tres. Incluso es válida este código:

```
let array=[1,2,3];
```

haces una desestructuración y asignas valores independientes a variables (x, y). Esto extrae elementos de un array.
let [x,y] = [...array]; es útil para extraer elementos y trabajar con ellos como variables independientes.

la desestructuración efectivamente crea variables nuevas donde "guardas" esos valores extraídos del objeto o array. Si no usas const, let o var para declararlas, JavaScript te dará un error porque tienes que declarar variables para poder usarlas.

Esto permite crear tres variables a=1, b=2, array=[3,4,5]

```
let [x,,y]=[...array];
```

Ahora ***x*** vale uno e ***y*** vale dos.

También es válida esta sintaxis:

```
let a,b,array;
[a,b,...array] = [1,2,3,4,5];
```

En este código, la variable ***a*** valdrá ***1***, ***b*** valdrá ***2*** y ***array*** será efectivamente un array con tres elementos que valdrán 3, 4 y 5 respectivamente.

Esta nueva sintaxis es muy poderosa, pero conviene adaptarse a ella poco a poco.

4.5 ESTRUCTURAS DE TIPO SET

4.5.1 INTRODUCCIÓN

Los **Sets** (que se pueden traducir con **conjuntos**) son una estructura de datos (son objetos realmente) que permiten, de forma similar a los arrays, almacenar datos. Es una estructura de datos que apareció en el estándar ES2015-

A diferencia de los arrays, no admiten valores duplicados y esa es su virtud. Es muy habitual recoger datos, pero eliminando aquellos que ya están repetidos. En los arrays esta tarea es pesada de realizar, pero con los conjuntos es muy sencilla ya que no hay que implementar nada, ya vienen preparados para esta labor.

4.5.2 DECLARACIÓN Y CREACIÓN DE CONJUNTOS

Podemos declarar e iniciar un conjunto vacío de valores:

```
const lista=new Set();
```

Esta forma declarar e iniciar variables deja claro que los conjuntos son objetos.

Añadir nuevos valores al conjunto requiere del uso del método add:

```
lista.add(8);
lista.add(6);
lista.add(5);
lista.add(5);
lista.add(6);
console.log(lista);
```

Lo cual muestra el siguiente resultado:

```
Set { 8, 6, 5 }
```

Dejando claro que los valores duplicados no se mantienen en el conjunto.

Como el método **add** devuelve una referencia al propio conjunto, se puede realizar la misma acción de esta forma:

```
lista.add(8).add(6).add(5).add(5).add(6);
console.log(lista);
```

Podemos también iniciar la lista a partir de un array:

```
const lista=new Set([5,6,4,5,6,5,5,6,4,6,6]);
console.log(lista);
```

Muestra:

```
Set { 5, 6, 4 }
```

Aunque los datos procedan de un array, no se permiten valores duplicados.

Es posible también iniciar el conjunto con un texto:

```
const lista=new Set("Conjunto");
console.log(lista);
```

Sin embargo, el resultado es sorprendente:

```
Set { 'C', 'o', 'n', 'j', 'u', 't' }
```

Cada elemento del conjunto es una letra y, como siempre, no se repiten los mismos caractere

Sí es posible añadir strings largos, pero mediante el método add:

```
const lista=new Set();
lista.add("Conjunto");
console.log(lista);
```

Resultado:

```
Set { 'Conjunto' }
```

4.5.3 MÉTODOS DE LOS CONJUNTOS

4.5.3.1 TAMAÑO DEL CONJUNTO

La propiedad **size** permite saber el tamaño de un conjunto:

```
const lista=new Set([2,4,6,8,10]);
console.log(lista.size);
```

Escribe el número cinco, ya que hay cinco elementos en el conjunto.

4.5.3.2 ELIMINAR VALORES

El método **delete** elimina el valor indicado de un conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
```

```
lista.delete(6);
console.log(lista);
```

Escribe:

```
Set { 1, 2, 3, 4, 5, 7, 8, 9 }
```

El número 6 ha desaparecido del conjunto.

Hay un método más agresivo: **clear**. Este método elimina todo el conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
lista.clear();
console.log(lista);
```

Escribe:

```
Set {}
```

Indicando que el conjunto está vacío.

4.5.3.3 BUSCAR VALORES

El método **has** permite que se le indique un valor y devuelve **true** si dicho valor forma parte del conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
console.log(lista.has(7)); //escribe true
console.log(lista.has(10)); //escribe false
```

4.5.4 CONVERTIR CONJUNTOS EN ARRAYS

JavaScript posee un operador muy interesante conocido como operador de **propagación** (en inglés se llama operador **spread**). Este operador usa tres puntos seguidos detrás de los cuales podemos indicar el conjunto a convertir.

```
const lista=new Set([1,2,3,4,5,6,7,8,9])
const array=[...lista];
console.log(array);
```

Escribe:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

El conjunto se ha convertido en un array.

4.5.5 RECORRER CONJUNTOS

Al igual que ocurre con los arrays, es habitual necesitar recorrer cada elemento de un array. Para ello, disponemos de los bucles **for..of** que recorren cada elemento del conjunto:

```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
```

```
for(let numero of lista){
    console.log(numero);
}
```

El resultado escribe cada número en una línea distinta. Solo necesitamos en este tipo de bucles indicar una variable para que vaya recogiendo los valores de cada elemento del conjunto. Esta variable se declara justo antes de la palabra **of**. El nombre del conjunto a recorrer se coloca después de esta misma palabra.

4.6 MAPAS

4.6.1 ¿QUÉ SON LOS MAPAS?

Los mapas permiten, en JavaScript, crear estructuras de tipo **clave-valor**, en las cuales las claves no se pueden repetir y tienen asociado el valor. Los mapas son también parte del estándar ES2015.

Tanto las claves como los valores pueden ser de cualquier tipo. En un mismo mapa no puede haber dos elementos con la misma clave, pero sí pueden repetir valor.

4.6.2 DECLARAR MAPAS

Para declarar un mapa e iniciarlo sin contenido basta el siguiente código:

```
const provincias=new Map();
```

Como los arrays y los conjuntos, realmente los mapas son objetos.

4.6.3 ASIGNAR VALORES A MAPAS

4.6.3.1 MÉTODO SET

El método **set** es el que permite asignar nuevos elementos. Este método requiere la clave del nuevo elemento y después el valor con el que asociamos dicha clave:

```
const provincias=new Map();
provincias.set(1,"Álava");
provincias.set(28,"Madrid");
provincias.set(34,"Palencia");
provincias.set(41,"Sevilla");
console.log(provincias);
```

El resultado es:

```
Map {
  1 => 'Álava',
  28 => 'Madrid',
  34 => 'Palencia',
```

```
  41 => 'Sevilla'
}
```

Si volvemos a añadir un elemento con la misma clave, este sustituye al anterior ya que no puede haber claves repetidas.

Es posible encadenar los métodos **set**:

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
    .set(41,"Sevilla");
console.log(provincias);
```

4.6.3.2 USO DE ARRAYS PARA CREAR MAPAS

También podemos utilizar un array donde cada elemento es otro array, en el que el primer elemento es la clave y el segundo el valor de esa clave. Podemos, a partir de dicho array, crear un mapa con las claves y valores del array:

```
const personas=new Map([[1,"Jose"],[2,"María"],[3,"Elena"]]);
console.log(personas);
```

Escribe:

```
Map { 1 => 'Jose', 2 => 'María', 3 => 'Elena' }
```

4.6.4 OPERACIONES SOBRE MAPAS

4.6.4.1 OBTENER VALORES DE UN MAPA

En los mapas es posible obtener el valor de una clave a partir del método **get** al que se le indica la clave del elemento que queremos obtener. La potencia de los mapas está en que obtener el valor referente a una clave es una operación muy rápida.

```
console.log(provincias.get(34)); //Escribe Palencia
```

4.6.4.2 BUSCAR UNA CLAVE EN UN MAPA

El método **has** permite buscar una clave en un mapa. Si la encuentra, devuelve **true** y si no, devuelve **false**.

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
    .set(41,"Sevilla");
console.log(provincias.has(34)); //Escribe true
console.log(provincias.has("Palencia"));//Escribe false
```

En el código anterior la búsqueda de la clave ***34*** produce un valor de **true** porque hay un elemento en el mapa que tiene esa clave. Al buscar el texto ***"Palencia"*** se devuelve false porque Palencia es un valor, pero no una clave.

4.6.4.3 BORRAR VALORES

El método **delete** permite eliminar un elemento del mapa. Para ello, hay que indicar la clave de dicho elemento:

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
  .set(41,"Sevilla");
provincias.delete(34); //Elimina el elemento con valor "Palencia"
console.log(provincias);
```

Resultado:

```
Map { 1 => 'Álava', 28 => 'Madrid', 41 => 'Sevilla' }
```

4.6.4.4 OBTENER OBJETOS ITERABLES

Un objeto iterable es un tipo de objeto semejante a un array, ya que es una colección de valores que se pueden recorrer mediante bucles de tipo **for...of**. Sin embargo, los objetos iterables no son realmente arrays y, por lo tanto, se manipulan de forma distinta.

Los mapas permite crear objetos iterables que contienen solo las claves y objetos iterables solo con los valores. El método **keys** obtiene las claves y el método **values** obtiene los valores.

Ejemplo:

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
    .set(41,"Sevilla");
let claves=provincias.keys();
for(let k of claves){
    console.log(k);
}
console.log("----------");
let valores=provincias.values();
for(let v of valores){
    console.log(v);
}
```

Resultado:

```
1
28
34
41
----------
Álava
Madrid
Palencia
Sevilla
```

4.6.5 CONVERTIR MAPAS EN ARRAYS

Al igual que ocurre con los conjuntos, el operador de propagación de JavaScript es el ideal para esta labor:

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
   .set(41,"Sevilla");
console.log(...provincias);
```

El resultado de este código permite ver el array que produce el operador:

```
[ 1, 'Álava' ] [ 28, 'Madrid' ] [ 34, 'Palencia' ] [ 41, 'Sevilla' ]
```

4.6.6 RECORRER MAPAS

El bucle **for...of** es el ideal para recorrer el contenido de los mapas. Cada valor que recoge este bucle es un array de dos elementos: el primero es la clave y el segundo el valor. La forma de recorrer los índices de un array es la siguiente:

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
   .set(41,"Sevilla");
for(let elemento of provincias){
   console.log(elemento);
}
```

Usando la forma habitual de recorrido de **for..of**, resulta que se nos proporciona en la variable que recoge cada elemento, un array con dos elementos: el primero es la clave del elemento y el segundo es el valor. Por ello, el código anterior mostraría este resultado:

```
[ 1, 'Álava' ]
[ 28, 'Madrid' ]
[ 34, 'Palencia' ]
[ 41, 'Sevilla' ]
```

También podemos desestructurar el array de los elementos para separar en dos variables la clave y el valor. Este podría ser otra versión del bucle para el mismo array:

```
for(let [clave,valor] of provincias){
   console.log(`Clave: ${clave}, Valor: ${valor}`);
}
```

La salida de este código es:

```
Clave: 1, Valor: Álava
Clave: 28, Valor: Madrid
Clave: 34, Valor: Palencia
Clave: 41, Valor: Sevilla
```

Este bucle es más versátil por separar de forma más cómoda la clave y el valor.

Los métodos **keys** y **values** también nos permiten recorrer las claves y los valores por separado. Para las claves sería:

```
for(let clave of provincias.keys()){
    console.log(clave);
}
```

Para los valores:

```
for(let valor of provincias.values()){
    console.log(valor);
}
```

Son muchas posibilidades, lo que otorga una gran versatilidad a los programadores a la hora de utilizar mapas.

RECOMENDACIÓN: FUNCIONES CALLBACK EN ARRAYS, CONJUNTOS Y MAPAS

- En el tema siguiente se estudiarán más métodos sobre arrays, mapas y conjuntos que realizan tareas muy avanzadas, pero que requieren del uso de funciones callback. Como este tipo de funciones se explican en el tema siguiente, será en ese momento cuando veamos estos métodos (véase 5.4.4 *"Uso de métodos avanzados para manipular estructuras de datos"*, en la página 185