UD4. ESTRUCTURAS BÁSICAS DE DATOS

1. STRINGS

1.1. USO DE STRINGS

- Los valores de tipo string tienen que delimitarse con comillas, bien sean dobles (“), simples (‘) o invertidas (`).
- Es posible que los strings contengan comillas como carácter literal, siempre que no coincidan con el delimitador del string. Ejemplos:

```
let s1=”Miguel dijo ‘venid’ en voz baja”;
let s2=”’Miguel dijo \’venid’ en voz baja’;
```

En el ejemplo ambos strings tienen el mismo contenido.

- Se pueden escapar caracteres para representar caracteres no visibles, que no están en el teclado o que sean problemáticos:

```
let s3=”Primera línea\nSegunda línea”;
let s4=”Mi sonrisa: \u{1F600}”; //s4 vale: Mi sonrisa: xxx
```

- Dentro de valores de string delimitados por comillas invertidas se pueden colocar directamente expresiones del lenguaje siempre que se coloquen dentro de los símbolos ${}.

Ejemplo:

```
let x=8, y=9;
console.log(`La suma de x e y es: ${x+y}
//Escribe: la suma de x e y es: 17
```

1.2. COMPARACIÓN DE STRINGS

Los textos se pueden comparar igual que los números.

```
let texto1=”ana”;
let texto2=”Ana”;
```

```
if(texto1==texto2)
console.log("son iguales");
else
console.log("son distintos");
```

Por pantalla aparece el texto son distintos porque se distingue entre mayúsculas y minúsculas.

Al comparar qué texto es mayor, se usa el orden que tienen los caracteres en la tabla *Unicode*. Así **Casa** es menor que **Caso**. Pero, hay que tener en cuenta que las minúsculas, en la tabla *Unicode*, son mayores que las mayúsculas y por eso JavaScript considera que **casa** es mayor que **Caso**. También hay problemas con los caracteres especiales de cada lengua (los que no posee el inglés) ya que **Ñu** se considera mayor que **Oso**, porque en la tabla Unicode la **ñ** tiene un código mayor que la letra **O.**

1.2.1 MÉTODO LOCALECOMPARE

Es un método que permite comparar sin discriminar entre mayúsculas y minúsculas, y considerando la forma de ordenar cada lengua.

Este método recibe como parámetro el texto con el que deseamos comparar. Podemos indicar un segundo parámetro que es el código oficial de idioma. Sin ese segundo parámetro, se utiliza la configuración local del equipo que ejecuta el código.

Este método devuelve un número positivo si el primer texto es mayor en orden alfabético, un número negativo si el segundo texto es mayor que el primero, y 0 si son iguales.

```
let texto3="Oso";
let texto4="Ñu";
console.log(texto3.localeCompare(texto4));
```

Este código devuelve el número 1, porque considera a la **ñ** más pequeña en orden alfabético que la letra **o**. Es lo deseable al ordenar español. Podemos indicar en el segundo parámetro el código del país cuyas reglas usamos al ordenar (es para inglés de EE.UU.).

```
console.log(texto3.localeCompare(texto4,"en-US"));
```

1.3. MÉTODOS DE LOS STRINGS

JavaScript aporta numerosos métodos y propiedades a los strings (los cuales forman parte de la programación orientada a objetos (POO)) permitiendo el uso de funciones que nos ayudarán en el uso de estos.

Acceder a los métodos y propiedades es simplemente poner el nombre de la variable o el valor de tipo string, colocar un punto y escribir el nombre del método o función. Los métodos pueden requerir más información para hacer su labor, estos datos extra se conocen como parámetros.

1.3.1. TAMAÑO DEL STRING

La propiedad **length** devuelve el tamaño del texto:

```
let texto="Nabucodonosor";
console.log(texto.length); //El resultado es el número 13
```

1.3.2. OBTENER UN CARÁCTER CONCRETO

El método **charAt** permite extraer un carácter concreto del texto. Basta con indicar la posición del carácter deseado, teniendo en cuenta que el primer carácter es el número 0, el siguiente el 1 y así sucesivamente. Ejemplo:

```
let texto=”Nabucodonosor”;
console.log(texto.charAt(5)); // Escribe la letra “o”
```

Hay otro método charCodeAt que funciona igual, pero devuelve el código del carácter de esa posición:

```
let texto=”Nabucodonosor”;
console.log(texto.charcodeAt(5)); // Escribe el valor 99, código Unicode de la letra “o”
```

1.3.3. CONVERTIR MAYÚSCULAS A MINÚSCULAS

Lo realizan los métodos **toUpperCase** (mayúsculas) y **toLowerCase** (minúsculas)

```
let texto=”Nabucodonosor”;
console.log(texto.toUppercase()); // Escribe “NABUCODONOSOR”
```

Para evitar problemas debido a la forma específica de pasar a mayúsculas de algunas lenguas disponemos de la función **toLocaleUpperCase** y **toLocaleLowerCase**. Se puede indicar un segundo parámetro con el código específico de país (es, fr, en ,etc.) pero, hoy en día, las funciones habituales (**toUpperCase** y **toLowerCase**) funcionan correctamente en casi cualquier idioma.

1.3.4. MÉTODOS DE BÚSQUEDA DE CARACTERES

indexOf

texto.**indexOf**(texto [,inicio])

Devuelve la posición del texto indicado en la variable, empezando a buscar desde el lado derecho. Si aparece varias veces, se devuelve la primera posición en la que aparece. El segundo parámetro (inicio) es opcional y nos permite empezar a buscar desde una posición concreta. Ejemplo:

```
var var1=”Dónde está la x, busca, busca”
document.write(var1.indexOf(“x”)); //Escribe 14
```

En el caso de que no se encuentre, devuelve -1

lastIndexOf

Como en el método anterior, pero en este caso, devuelve la última posición en la que aparece el texto (o –1, si no lo encuentra).

```
texto.lastIndexOf(texto [,inicio]);
```

endsWith

```
texto.endsWith(textoABuscar [,tamaño])
```

Devuelve verdadero si el texto finaliza con el texto que indiquemos en el parámetro **textoABuscar.**

```
var texto=”Esto es una estructura estática”;
console.log(texto.endsWith(“estática”)); //Escribe true
```

El primer parámetro tamaño solo mira si termina el string con el texto indicado considerando que el tamaño del texto es el indicado:

```
var texto=”Esto es una estructura estática”;
console.log(texto.endsWith(“estructura”),22);      //Escribe true porque los 22 primeros caracteres
                                                    //terminan con el texto estructura
```

startsWith

Texto.startsWith(textoABuscar [,posición])

Parecido al anterior, solo que este método busca si el string comienza con el texto indicado. Si es así, devuelve true. El parámetro posición mira si el texto comienza por el indicado, pero empezando a mirar desde la posición indicada.

```
var texto=”Esto es una estructura estática”;
console.log(texto.startsWith(“Esto”)); //Escribe true
console.log(texto.startsWith(“es”,5)); //Escribe true
```

1.3.5. EXTRAER Y MODIFICAR SUBCADENAS

replace

```
texto.replace(textoBuscado, textoReemplazado)
```

Busca en el string el texto indicado en el primer parámetro (**textoBuscado**) y lo cambia por el segundo texto (**textoReemplazado**). Ejemplo:

```
var texto=”Esto es una estructura estática”;
console.log(texto.replace(“st”,”xxtt”)); //Escribe “Exxtto es una estructura estática
```

trim

Es un método que no necesita argumentos. Se encarga de quitar los espacios en blanco a derecha e izquierda del texto. Ejemplo:

```
let texto=”      texto con muchos espacios      ”;
console.log(“-”+texto+”-”);      // Escribe      -      texto con muchos espacios      -
console.log(“-”+texto.trim+”-”)); // Escribe      -texto con muchos espacios-
```

slice

```
texto.slice(inicio [,fin])
```

Extrae del texto los caracteres desde la posición indicada por **inicio**, hasta la posición fin (sin incluir esa posición). Si no se indica fin, se toma desde inicio hasta el final.

```
let texto.=”Esto es una estructura estática”
console.log(texto.slice(3,10); // Escribe “o es un”
```

El parámetro fin puede usar números negativos, en ese caso, los números indican la posición desde el final del texto. Ejemplo:

```
let texto.=”Esto es una estructura estática”
console.log(texto.slice(3,-5); // Escribe “o es una estructura est”
```

substring

texto.**substring**(inicio [,fin])
Funciona igual que **slice** anterior, pero no admite usar números negativos.

substr

Texto.**substr(**inicio [,tamaño])
Extrae del texto el número de caracteres indicados por el parámetro **tamaño**, desde la posición indicada por el número asociado al carácter **inicio**. Si no se indica tamaño alguno, se extraen los caracteres desde la posición indicada por ***inicio*** hasta el final.

```
let texto.=”Esto es una estructura estática”
console.log(texto.substr(3,10); // Escribe “o es una et”
```

split

texto.**split**([textoDelim [,límite])
Es un potente método capaz de dividir el texto en un array de textos.

Sin indicar parámetro alguno, se genera un array donde cada elemento del array será cada carácter del string. Si se indica el parámetro textoDelim, este se usa como texto delimitador; es decir, se divide el texto en trozos separados por ese delimitador. Ejemplo:

```
let texto=”Esto es una estructura estática”
console.log(texto.split(“ “)); //Escribe [ ‘Esto’, ‘es’, ‘una’, ‘estructura’, ‘estática’ ]
```

El segundo parámetro pone un límite tope de divisiones. Por ejemplo:

```
let texto=”Esto es una estructura estática”
console.log(texto.split(“ “,3)); //Escribe [ ‘Esto’, ‘es’, ‘una’ ]
```

El texto delimitador puede ser una expresión regular en lugar de un texto, eso permite dividir el texto en bas a criterios más complejos.

1.3.6. CONVERTIR CÓDIGO A TEXTO

El método **fromCharCode** permite que indiquemos una serie de números de código de la tabla Unicode y nos devolverá un string formado por los caracteres corresondientes a estos códigos.

Este método es estático, lo que significa que se usa indicando la clase String, es decir String.**fromCharCode**. Lo cual es lógico porque este método crea un string. Ejemplo:

```
console.log(String.fromCharCode(65,66,67));
```

El código anterior escribe el texto **ABC**, porque 65 es el código de la letra A, 66 el de B y 67 el de C.

2. ARRAYS

2.1. ¿QUÉ ES UN ARRAY?

Todos los lenguajes de programación disponen de un tipo de variable que es capaz de manejar conjuntos de datos. A este tipo de estructuras se las llama **arrays** de datos. También se las llama listas, vectores o arreglos, pero estos nombres tienen connotaciones que hacen que se puedan confundir con otras estructuras de datos. Por ello, es más popular el nombre sin traducir: array.

Los *arrays* aparecieron en la ciencia de la programación de aplicaciones para solucionar un problema habitual al programar. Supongamos que deseamos almacenar 25 notas de alumnos para ser luego manipuladas dentro del código JavaScript. Sin *arrays*, necesitamos 25 variables distintas. Manejar estos datos significaría estar continuamente indicando, de forma individual, el nombre de esas 25 variables. Los *arrays* permiten manejar las 25 notas bajo un mismo nombre.

En definitiva, los *arrays* son variables que permiten almacenar, usando una misma estructura, una serie de valores. Para acceder a datos individuales dentro del array, hay que indicar su posición, conocida como índice. Así, por ejemplo, nota[4] es el nombre que recibe el quinto elemento de la sucesión de notas. La razón por la que nota[4] se refiere al quinto elemento y no al cuarto es porque el primer elemento tiene índice cero.

Esto, con algunos matices, funciona igual en casi cualquier lenguaje. Sin embargo, en JavaScript los *arrays* son objetos. Es decir, no hay un tipo de datos array, si utilizamos *typeof* para averiguar el tipo de datos de un array, el resultado será la palabra *object*.

En algunos lenguajes como **C**, el tamaño del array se debe anticipar en la creación del array y no se puede cambiar más adelante. Este tipo de arrays son estáticos. Los arrays de JavaScript totalmente dinámicos, su tamaño se puede modificar después de la declaración a voluntad.

nota

nombre del array permite referirnos al array entero

7	nota[0]
8	nota[1]
6	nota[2]
6	nota[3]
5	nota[4]
4	nota[5]
3	nota[6]
9	nota[7]

Otro detalle importante, que diferencia a JavaScript respecto a lenguajes más formales, es que en JavaScript los arrays son heterogéneos. Los lenguajes como **C** o **Java** usan arrays homogéneos, que son arrays que solo pueden almacenar valores del mismo tipo. JavaScript admite, por ejemplo, que un elemento sea un número y otro un string. Cada elemento del array puede ser de un tipo distinto.

2.2. CREACIÓN DE ARRAYS

2.2.1. DECLARACIÓN Y ASIGNACIÓN DE VALORES

Hay muchas formas de declarar un array. Por ejemplo, si deseamos declarar un array vacío, la forma habitual es:

```
let a=[];
```

Los corchetes vacíos tras la sginación, significan array vacío. Un array que se llama simplemente **a** y que está listo para indicar valores.

Equivalente a ese código, podemos hacer:

```
let a=new Array();
```

Lo cual ya nos anticipa que los arrays, en realidad, son objetos. El operador new sirve para crear objetos.

Para colocar valores en el array se usa el índice que indica la posición del array en la que colocamos el valor. El primer índice es el 0, por ejemplo:

```
a[0]=”Antonio”;
```

En este caso asignamos el texto ***Antonio*** al primer elemento del array.

Podemos seguir:

```
a[1]=”Luis”;
a[2]=”Marta”;
a[3]=”Sofía”;
```

Si quisiéramos mostrar un elemento concreto por consola haríamos:

```
console.log(a[2]); // Muestra: Marta
```

Podemos asignar valores en la propia declaración del array;

```
let nota=[7,8,6,6,5,4,3,9];
```

Disponemos también de esta otra forma de declarar:

```
let nota=new Array(7,8,6,6,5,4,3,9);
```

En este caso, la variable nota es un array de 10 elementos, todos números. Para mostrar el primer elemento:

```
console.log(nota[0]); // Muestra: 7
```

El método **console.log** tiene capacidad para mostrar todo un array:

```
console.log(nota); //Muestra: [7,8,6,6,5,4,3,9]
```

Sin embargo, salvo para tareas de depuración, no es conveniente este uso. Lo lógico es utilizar bucles de recorrido de arrays para mostrar el valor del array en la forma que deseemos.

2.2.2. USO DE CONST Y LET EN ARRAYS

Otro detalle a comentar; es que es muy habitual declarar *arrays* con la palabra clave *const* en lugar de con *let*.

```
const datos=[4.5,6.78,7.12,9.123]
```
Hemos declarado un array de 4 valores decimales. Lo hemos declarado con const, lo que en principio indica que el valor de la variable datos no puede variar. Sin embargo, este código es válido:

```
datos[0]=4.671;
```
Hemos modificado el primer elemento del array. Incluso es válido este otro:

```
datos[4]=3.87;
```
Hemos añadido un nuevo elemento al array. Luego, aun declarando las variables con la palabra clave **const**, se ha modificado el contenido del array.

La razón es que, realmente, los arrays son objetos. De hecho, este código:

```
console.log(typeof datos); //Como resultado escribe object.
```

Los objetos serán tratados más adelante, pero adelantamos que cuando se crea un objeto, la variable a la que se asigna el array realmente es lo que nos sirve para llegar al valor del array, es una referencia al array, pero no es el array en sí. Es decir, en el ejemplo anterior, la variable (aunque en realidad es una constante) en realidad es un identificador para acceder a los datos del array. Para entender la idea, hemos de saber que este código sí provoca un error.

```
const datos=[4.5,6.78,7.12,9.123];
datos=[9.18,4.95];
```
En la primera línea declaramos **datos** como la forma de acceder al array que tiene valores: **[4.5,6.78,7.12,9.123]**. Esta variable es un enlace o una referencia al array. Pero en la segunda línea decimos que la variable **datos** es un enlace a otro array. Por lo tanto, ahora sí estamos modificando la referencia y eso no se permite porque la variable **datos** se ha declarado con la palabra **const**.

Cuando modificamos los elementos del array, o eliminamos elementos o incluso cuando los añadimos, el array sigue siendo el mismo. La referencia no cambia.

2.2.3. OPERACIÓN DE ASIGNACIÓN EN ARRAYS

Aun hemos de entender esta idea con otro detalle:

```
const datos=[4.5,6.78,7.12,9.123];
const datos2=datos;
datos2[0]=400;
console.log(datos[0]); // Escribe 400
```
Cuando asignamos la variable **datos2** al array datos, no se hace una copia del array. Tanto **datos** como **datos2** son 2 variables que hacen referencia al mismo array. Por eso, cuando cambiamos el primer elemento del array **datos2**, también cambiamos el del array **datos**, ya que en realidad es el mismo array.

2.2.4. VALORES INDEFINIDOS

Veamos este código:

```
let a=["Saúl",”Rocío”];
a[3]=”María”;
console.log(a[2]); //Escribe undefined
```

En la definición del array, estamos creando el array llamado a y dando valor a los 2 primeros elementos (a[0] y a[1]). Luego, damos valor al 4º elemento del array (a[3]). En ningún momento hemos dado valor al elemento a[2] y por eso el código provoca la escritura del texto **undefined**.

El resumen es que podemos dejar elementos sin definir en un array. Es más, incluso en la propia declaración se pueden dejar elementos vacíos.

```
let a=["Saúl",”Rocío”,,”María”];
```
Las 2 comas seguidas en la definición hacen que el tercer elemento se ignore.

2.2.5. ELIMINAR ELEMENTOS DE UN ARRAY

Para borrar un elemento se utiliza la palabra **delete** detrás del cual se indica el elemento a eliminar.

```
const dias=["Lunes”,”Martes”,”Miércoles”,”Jueves”,”Viernes”,”Sábado”,”Domingo”];
delete dias[2];
console.log(dias);
```
Aparece:

```
[‘Lunes’,
‘Martes’,
<1 empty item>,
‘Jueves’,
‘Viernes’,
‘Sábado’,
‘Domingo’]
```
Ha desaparecido el miércoles, pasando a ser indefinido ese elemento.

2.2.6. ARRAYS HETEROGÉNEOS

Los *arrays* en JavaScript son heterogéneos. Admiten mezclar en el mismo array valores de diferente tipo:

```
const a=[3,4,”Hola”,true,Math.random()];
```
Incluso podemos definir arrays de este tipo:

```
const b=[3,4,”Hola”,[99,55,33]];
```
El cuarto elemento (b[3]) es un array. Es decir, se pueden colocar arrays dentro de otros arrays. Es más, los elementos de un array pueden ser de cualquier tipo, hay arrays de todo tipo de objetos. De modo que la siguiente instrucción es perfectamente posible:

```
console.log(b[3][1]); // Escribe 55
```
En el ejemplo anterior, b[3][1] hace referencia al 2º elemento del cuarto elemento de **b**.

En definitiva, las posibilidades de uso de *arrays* en JavaScript para almacenar datos son espectaculares.

3. RECORRER ARRAYS

3.1. USO DEL BUCLE FOR PARA RECORRER ARRAYS

Una de las tareas fundamentales, en la manipulación de *arrays*, es recorrer cada elemento de un array para poderlo examinar. Esto es fácil de hacer mediante el bucle *for*.

```
const notas=[5,6,7,4,9,8,9,9,7,8];
for(let i=0;i<notas.length;i++){
   console.log(´La nota ${i} es ${notas[i]}’);
}
```

En el bucle se utiliza el método *length* que nos permite saber el tamaño de un array. El contador *i* recorre todos los índices del array notas. El resultado del código es:

```
La nota 0 es 5
La nota 1 es 6
La nota 2 es 7
La nota 3 es 4
La nota 4 es 9
La nota 5 es 8
La nota 6 es 9
La nota 7 es 9
La nota 8 es 7
La nota 9 es 8
```

El problema es cuando hay elementos indefinidos en el array:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let i=0;i<notas.length;i++){
   console.log(´La nota ${i} es ${notas[i]}’);
}
```

Ahora el array tiene muchos elementos indefinidos. Si se deja así, el código mostrará lo siguiente:

```
La nota 0 es 5
La nota 1 es 6
La nota 2 es undefined
La nota 3 es undefined
La nota 4 es undefined
La nota 5 es 9
La nota 6 es undefined
La nota 7 es undefined
La nota 8 es 8
La nota 9 es undefined
La nota 10 es 9
La nota 11 es undefined
La nota 12 es 7
La nota 13 es 8
```

Lo lógico es evitar los elementos indefinidos. Por lo que el código debería modificarse:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let i=0;i<notas.length;i++){
        if(notas[i]!=undefined)
                console.log(´La nota ${i} es ${notas[i]}’);
}
```

El resultado sería el siguiente:

```
La nota 0 es 5
La nota 1 es 6
La nota 5 es 9
La nota 8 es 8
La nota 10 es 9
La nota 12 es 7
La nota 13 es 8
> |
```

3.2. BUCLE FOR...IN

JavaScript posee un bucle mucho más cómodo para recorrer *arrays*. Se trata de un bucle *for* especial llamado **for..in**. La ventaja es que no necesita tanto código, basta con indicar el nombre del contador y el array que se recorre. Su sintaxis general es la siguiente:

```
for(let índice in nombreArray){
    instrucciones
}
```

El índice es el nombre de una variable que se declara en el propio *for*, y que irá tomando todos los valores del índice del array que recorre. Esa variable se salta los elementos indefinidos, solo recorre los definidos. Por lo que el bucle *for..in* que nos permite recorrer las notas, saltándonos las indefinidas, es:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let  i in notas){
        console.log(`La nota ${i} es ${notas[i]}`);
}
```

Se puede observar lo limpio que queda el código de recorrido con este bucle.

3.3. BUCLE FOR..OF

El estándar ES2015 incorporó un nuevo bucle llamado **for..of**. Pensado para simplificar aún más el recorrido de *arrays*, es un bucle similar al anterior, pero en el que la variable que se crea en el bucle va almacenando los diferentes valores del array (en lugar de los índices como hacía el for..in).

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let nota of notas){
        console.log(nota);
}
```

El resultado es:

```
5
6
3 undefined
9
2 undefined
8
undefined
9
undefined
7
8
>
```

Hay una diferencia: for..of no se salta los elementos indefinidos. Nuevamente, necesitamos usar la instrucción **if** para poder saltarlos.

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let nota of notas){
        if(nota!=undefined)
                console.log(nota);
}
```

Es un bucle muy simple de utilizar y se ha convertido en el bucle habitual de recorrido.

- ❑ **ACTIVIDAD 1** --- FOR..IN Y FOR..OF CON STRINGS ---

4. MÉTODOS DE LOS ARRAYS

Los *arrays* son objetos. Eso, entre otras cosas, significa que cada array tiene propiedades y métodos ya incorporados pensados para facilitarnos el trabajo. Para acceder a esos métodos basta con colocar un punto tras el nombre del array y después escribir el nombre del método o propiedad.

4.1. TAMAÑO DEL ARRAY

La propiedad **length** nos permite obtener el tamaño de un array.

```
const dias=["Lunes",”Martes”,”Miércoles”,”Jueves”,”Viernes”,”Sábado”,”Domingo”];
console.log(dias.length); //Escribe 7
```

Gracias a este método, es fácil añadir un elemento al final de un array:

```
nota[nota.length]=18; //Añadimos el número 18 al final del array nota.
```

4.2. SABER SI UN ELEMENTO ES UN ARRAY

JavaScript aporta un operador llamado **instanceof**. Este método, en realidad, sirve para poder comprobar si un objeto pertenece a una determinada clase. Los arrays pertenecen a la clase **Array** (con A mayúscula).

```
let a=[1,2,3,4,5,6,7,8,9];
let b=”Hola”;
console.log(a instanceof Array); // Escribe true
console.log(b instanceof Array); // Escribe false
```

4.3. MÉTODOS PARA AÑADIR ELEMENTOS

El método **push** permite añadir un elemento al final de un array. Requiere indicar qué valor le damos a ese elemento:

```
const cantantes=["Quevedo"];
cantantes.push("Alejo");
```

Ahora el array cantantes tiene 2 elementos.

El método contrario es **pop**. Lo que hace es retirar del array el último elemento, podemos asignar dicho elemento a una variable.

```
const cantantes2=["Peter ", "Paul ", " Mary "];
let x =cantantes2.pop();
console.log(`El array ha quedado así: ${cantantes2}`);
console.log(`La variable x vale ${x}`);
```

El método pop no tiene argumento, pero requiere paréntesis (como todos los métodos). Tras el uso de pop, se quita el valor Mary y se asigna a una variable llamada x. Por eso, lo que se escribe en pantalla es:

```
El array ha quedado así: Peter,Paul
La variable x vale Mary
>
```

Hay 2 métodos más que permiten añadir y quitar elementos a un array. La diferencia es que lo hacen por delante, por el primer elemento del array. El método que quita el primer elemento del array se llama **shift**.

```
const cantantes3 =["Crosby","Stills","Nash"];
let x =cantantes3.shift();
console.log(`El array ha quedado así: ${cantantes3}`);
console.log(`La variable x vale ${x}`);
```

```
El array ha quedado así: Stills,Nash
La variable x vale Crosby
>
```

El método contrario es **unshift**, al que hay que indicar como parámetro el valor a añadir por el lado izquierdo:

```
const cantantes4 =["Crosby","Stills","Nash"];
let x =cantantes4.unshift("Young");
console.log(`El array ha quedado así: ${cantantes4}`);
```

El texto Young pasa a ser el primer elemento, el resto se desplaza:

```
El array ha quedado así: Young,Crosby,Stills,Nash
```

4.4. MEZCLAR ARRAYS

El método **concat** permite añadir los elementos de un array al array que usa el método. El resultado de este método es un nuevo array, no se modifica ninguno de los *arrays* originales, que contiene los elementos unidos de ambos *arrays*. Por eso, el resultado de *concat* debe de ser asignado a un nuevo array. Ejemplo:

```
const planetas1 =["Mercurio","Venus","La Tierra", "Marte"];
const planetas2 =["Júpiter","Saturno","Urano", "Neptuno"];
const planetas =planetas1.concat(planetas2);
console.log(planetas);
```

```
▼ Array(8) i
    0: "Mercurio"
    1: "Venus"
    2: "La Tierra"
    3: "Marte"
    4: "Júpiter"
    5: "Saturno"
    6: "Urano"
    7: "Neptuno"
    length: 8
  ▶ [[Prototype]]: Array(0)
```

4.5. OBTENER Y AÑADIR SUBARRAYS

4.5.1. OBTENER SUBARRAYS

El método **slice**, permite copiar una serie de elementos de un array indicando el índice del primer elemento que deseamos copiar y el índice final. El resultado es un nuevo array.

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
const tresDias=dias.slice(3,6);
console.log(tresDias);
```

```
▼ (3) ['Jueves', 'Viernes', 'Sábado'] i
```

En la línea dias.slice(3,6) se está extrayendo los elementos con índice 3, 4 y 5 (el sexto no se incluye). Estos 3 elementos formarán un nuevo array (en el código se llama tresDias).

Si el segundo número no se indica, se extrae hasta el final. Es decir dias.slice(3) obtiene los valores [Jueves","Viernes","Sábado","Domingo"]

4.5.2. COPIAR ARRAY

El propio método **slice** nos permite hacer una copia de un array. Para ello basta que se use sin parámetros.

```
let array2=array1.slice(); // array2 es una copia de array1.
```

4.5.3. AÑADIR Y ELIMINAR ELEMENTOS

El método **splice** es muy interesante. Permite eliminar definitivamente los elementos de un array. Para ello, se indica como primer parámetro, desde dónde iniciamos la eliminación y cuántos elementos eliminamos. Este método modifica el array original y devuelve otro array que contiene los elementos eliminados.

```
const castillaLaMancha=["Guadalajara","Sevilla","Granada","Ciudad Real","Albacete"];
const borrados=castillaLaMancha.splice(1,2);
console.log(`Se han borrado los elementos: ${borrados}`);
console.log(`Ahora quedan: ${castillaLaMancha}`);
```

```
Se han borrado los elementos: Sevilla,Granada
Ahora quedan: Guadalajara,Ciudad Real,Albacete
```

Aun más interesante es el hecho de que, además de eliminar elementos, los podemos cambiar por la serie de elementos que indiquemos:

```
const castillaLaMancha=["Guadalajara","Sevilla","Granada","Ciudad Real","Albacete"];
const borrados=castillaLaMancha.splice(1,2,"Toledo","Cuenca");
console.log(castillaLaMancha);
```

Resulta:

```
(5) ['Guadalajara', 'Toledo', 'Cuenca', 'Ciudad Real', 'Albacete']
```

El número de elementos que se añaden no tiene por qué coincidir con los que se quitan.

4.6. CONVERTIR UN ARRAY EN STRING

El método **join** de los *arrays* permite convertir un array en un *string*. El *string* resultante contiene todos los elementos de un array:

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
console.log(dias.join(""));
console.log(dias.join("-"));
console.log(dias.join("<->"));
```

El resultado sería:

```
LunesMartesMiércolesJuevesViernesSábadoDomingo
Lunes-Martes-Miércoles-Jueves-Viernes-Sábado-Domingo
Lunes<->Martes<->Miércoles<->Jueves<->Viernes<->Sábado<->Domingo
```

4.7. BÚSQUEDA DE ELEMENTOS EN UN ARRAY

4.7.1. MÉTODO INDEXOF

Este método permite buscar el valor elemento de un array. Si lo encuentra devuelve su posición, y si no, devuelve el valor –1. Ejemplo:

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
console.log(dias.indexOf("Jueves")); //Escribe 3
```

Si hubiese elementos con el valor “Jueves” devolvería el primero empezando por la izquierda. También podemos indicar el inicio de la búsqueda, por ejemplo:

```
dias.indexOf(“Jueves”,4);
```

Busca la palabra **Jueves** a partir del elemento con índice 4 (en este caso no lo encontrará y devolverá -1)

4.7.2. MÉTODO LASTINDEXOF

Método idéntico al anterior, solo que empieza a buscar desde el último elemento:

```
const números=[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
console.log(números.lastIndexOf(3));
```

Este código devuelve **12**, que es la última posición en la que aparece el valor **3** en el array. Al igual que ocurría con **IndexOf**, podemos empezar a buscar desde una posición concreta, pero, en este caso, se busca hacia la izquierda desde esa posición. Por ejemplo:

```
const números=[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
console.log(números.lastIndexOf(3,6));
```

Escribe el número 5, porque empezando a buscar desde la posición 6 hacia la izquierda, el primer valor que tiene un 3 es el elemento índice 5.

4.7.3. MÉTODO INCLUDES

Apareció en el estándar ES2015. Es un método que permite buscar un elemento y devuelve **true** si ese valor está en el array y **false** si no está.

```
const números=[2,3,5,6,4,3,7,8,4,3,2,8,3,2];
console.log(números.includes("Jueves")); //Escribe true
```

4.8. MODIFICAR EL ORDEN DE LOS ELEMENTOS DE UN ARRAY

4.8.1. INVERTIR EL ORDEN

El método reverse es capaz de dar la vuelta a los elementos de un array:

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
dias.reverse();
console.log(dias);
['Domingo', 'Sábado', 'Viernes', 'Jueves', 'Miércoles', 'Martes', 'Lunes']
```

Se puede observar que reverse no devuelve un nuevo array, sino que modifica el array original.

4.8.2. MÉTODO SORT

Uno de los más interesantes métodos para los *arrays*. Permite ordenar los elementos de un array. Ejemplo:

```
const dias=["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"];
dias.sort();
console.log(dias);
['Domingo', 'Jueves', 'Lunes', 'Martes', 'Miércoles', 'Sábado', 'Viernes']
```

Es una función de fácil uso, pero tiene el problema habitual de no saber ordenar con las características nacionales y de considerar que las minúsculas son mayores que las mayúsculas.

Observemos este ejemplo:

```
const palabras=["Ñu","Águila","boa","oso","marsopa","Nutria"];
palabras.sort();
console.log(palabras);
```

```
['Nutria', 'boa', 'marsopa', 'oso', 'Águila', 'Ñu']
```

Aparece **Nutria** como primera palabra y el **Ñu** es el último (aunque hay un oso) y el **Águila** el penúltimo. La razón es que la letra **A** con tilde está muy por detrás de la **Z** en la tabla Unicode. Lo mismo ocurre con la **ñ** y además, está el problema, de las mayúsculas y las minúsculas.

La manera de solventar este problema es gracias a que el método **sort** admite indicar una función de tipo **callbak** que nos permita personalizar la ordenación. Esta posibilidad se describirá más adelante.

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 se añadir facilidades para manipular varios datos a la vez y recogerlos 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* únicamente.

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 valores de 2 variables.

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

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

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

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

La variable **x** valdrá uno, **y** valdrá 2 y **z** valdrá 3. Incluso es válido este código.

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

Ahora **x** vale 1 e **y** vale 3.

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 3 elementos que valdrán 3, 4 y 5 respectivamente.

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

5. ESTRUCTURAS DE TIPO SET

5.1. INTRODUCCIÓN

Los Sets (conjuntos) son estructuras 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.

5.2. DECLARACIÓN DE CONJUNTOS

Podemos declarar e iniciar un conjunto vacío de valores:

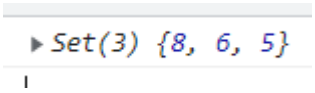
```
const lista=new Set();
```

Esta forma de 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:



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 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);
```

```
▶ Set(3) {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);
```

Pero el resultado, es sorprendente:

```
▶ 0: "C"
▶ 1: "o"
▶ 2: "n"
▶ 3: "j"
▶ 4: "u"
▶ 5: "t"
```

Cada elemento es una letra y, como siempre, no se repiten los mismos caracteres. Sí es posible añadir strings largos, pero mediante el método add:

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

```
▼ Set(1) {'Conjunto'}
```

5.3. MÉTODOS DE LOS CONJUNTOS

5.3.1. TAMAÑO DE LOS CONJUNTOS

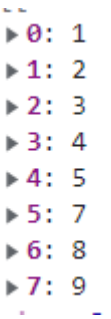
La propiedad **size** permite saber el tamaño de un conjunto:

```
const lista=new Set([2,4,6,8,10]);
console.log(lista.size); // Escribe 5
```

5.3.2. ELIMINAR VALORES

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

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



El método **clear** elimina todo el conjunto.

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

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
```

5.4. CONVERTIR CONJUNTOS EN ARRAYS

JavaScript posee un operador muy interesante conocido como operador de propagación (**spread**). Este operador usa 3 puntos seguidos (**...**) detrás de los cuales podemos indicar el conjunto a convertir.

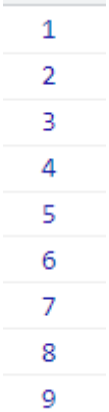
```
const lista=new Set([1,2,3,4,5,6,7,8,9]);
const array=[...lista];
console.log(array);
```

Escribe: `[1, 2, 3, 4, 5, 6, 7, 8, 9]` donde el conjunto se ha convertido en array.

5.5. RECORRER CONJUNTOS

Al igual que ocurre con los *array*s, 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.

6. MAPAS

6.1. ¿QUÉ SON LOS MAPAS?

Los mapas permiten, en JavaScritpt, 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 2 elementos con la misma clave, pero sí pueden repetir valor.

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 sets, realmente los mapas son objetos.

6.3. ASIGNAR VALORES A MAPAS

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," Valencia");
provincias.set(34, " Alicante");
provincias.set(42, " Castellón");
console.log(provincias);
```

El resultado es:

```
▸ Map(4) {1 => 'Álava', 28 => 'Valencia', 34 => 'Alicante', 42 => 'Castellón'}
```

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,"Valencia").set(34,"Alicante").set(42,"Castellón");
console.log(provincias);
```

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, "Júlia"],[2,"Marc"],[3,"Aina"]]);
console.log(personas);
```

```
▸ Map(3) {1 => 'Júlia', 2 => 'Marc', 3 => 'Aina'}
```

6.4. OPERACIONES SOBRE MAPAS

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 Alicante
```

6.4.2. BUSCAR UNA CLAVE DE 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,"Valencia").set(34,"Alicante").set(42,"Castellón");
console.log(provincias.has(34)); //Escribe true
console.log(provincias.has("Alicante"); //Escribe false
```

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,"Valencia").set(34,"Alicante").set(42,"Castellón");
provincias.delete(34); //Elimina el elemento con valor "Alicante"
console.log(provincias);
```

```
▶ Map(3) {1 => 'Álava', 28 => 'Valencia', 42 => 'Castellón'}
```

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 del tipo **for...of.** Sin embargo, los objetos iterables no son realmente *arrays* y, por lo tanto, se manipulan de forma distinta.

Los mapas permiten 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,"Valencia").set(34,"Alicante").set(42,"Castellón");
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);
}
```

```
1
28
34
42
----------
Álava
Valencia
Alicante
Castellón
>
```

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,"Valencia").set(34,"Alicante").set(42,"Castellón");
console.log(...provincias);
```

El resultado de este código permite ver el array que produce el operador (**...**):

```
▸ (2) [1, 'Álava'] ▸ (2) [28, 'Valencia'] ▸ (2) [34, 'Alicante'] ▸ (2) [42, 'Castellón']
```

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 2 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,"Valencia").set(34,"Alicante").set(42,"Castellón");
for(let elemento of provincias){
    console.log(elemento);
}
```

Usando la forma habitual de recorrido **for..of**, resulta que se nos proporciona la variable que recoge cada elemento, un array con 2 elementos: el primero es la clave y el segundo es el valor. Por ello, el código mostraría este resultado:

```
▸ (2) [1, 'Álava']
▸ (2) [28, 'Valencia']
▸ (2) [34, 'Alicante']
▸ (2) [42, 'Castellón']
```

También podemos desestructurar el array de los elementos para separar en 2 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: Valencia
Clave: 34, Valor: Alicante
Clave: 42, Valor: Castellón
```

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