



Estructuras de datos

Objetivos

- Conocer la importancia de las estructuras de datos.
- Analizar el funcionamiento de arrays, conjuntos y mapas.
- Exponer las operaciones de manipulación sobre estructuras de datos.
- Enumerar las operaciones básicas principales.
- Aprender a escribir código más eficiente.
- Introducir la importancia del aprendizaje de funciones.

Contenidos

- 3.1. Arrays
- 3.2. Conjuntos
- 3.3. Mapas

Introducción

Los tipos de datos simples que se dieron a conocer en la unidad anterior representan la arcilla con la que se construyen los programas. Pero si hay que construir un edificio se necesitan ladrillos, vigas y otras estructuras más robustas que cumplan funciones de mayor responsabilidad.

En esta unidad se estudian los ladrillos del edificio, estructuras de datos organizados en colecciones que pueden manipularse como si fueran una única unidad, de la misma manera que la arcilla da forma a cada ladrillo, que a su vez se combina con otros para crear estructuras más complejas.

Al final de esta unidad se sabrá lo suficiente como para comenzar a escribir programas más útiles y se tendrá la preparación suficiente para abordar el estudio de conceptos más complejos.

3.1. Arrays

Un **array** es una estructura de datos (concretamente, un objeto) ordenada que permite almacenar múltiples valores bajo un mismo identificador. Su nomenclatura es muy variada, pudiéndose encontrar referencias a los arrays como arreglos, vectores, listas, tablas o relaciones; aunque en muchos casos son denominaciones que coinciden con otros conceptos del mundo de la programación y a veces conducen a malentendidos. Por ello, esta obra se refiere a ellos por su denominación anglosajona sin traducir, *arrays*.

En JavaScript, a diferencia de otros lenguajes de programación, pueden almacenarse elementos de distinto tipo dentro del mismo array. Además, su tamaño es elástico, es decir, no hay por qué limitarse a la cantidad de elementos que tiene en un momento dado: pueden añadirse y eliminarse elementos siempre que sea necesario, y el array se hará más grande o pequeño de forma dinámica.

3.1.1. Tipos de arrays

Lo primero que debe saberse de los arrays es que tienen un nombre, un identificador que se les asigna (igual que a las variables), contienen una serie de valores y cada valor tiene asociado un índice.

En la Figura 3.1 se muestra un array unidimensional llamado **distancias**, que tiene seis elementos de tipo numérico. Para acceder a cada elemento se hace referencia al índice de la posición que ocupa en el array.

También se pueden tener **arrays bidimensionales**, al estilo de cualquier estructura tabular, y hacer referencia a los elementos indicando en qué fila y columna se sitúan (Figura 3.2).



Figura 3.1. Conceptos sobre la morfología de un array.

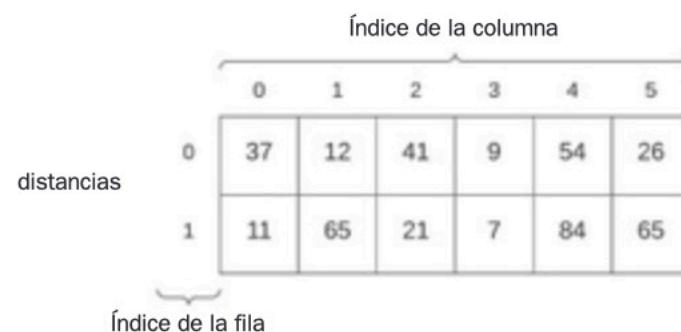


Figura 3.2. Indicadores de posición de un array bidimensional de 2x6.

En este caso se tiene una estructura de datos en forma de **array bidimensional** que almacena seis distancias en cada una de sus dos filas. De manera que, para hacer referencia al elemento cuyo valor es 7, debe indicarse que se desea la distancia que está almacenada en la posición determinada por la fila 1 y la columna 3.

Para utilizar un **array tridimensional** lo mejor es imaginarse un cubo de Rubik en el que se almacena un valor para cada combinación X, Y, Z, tal y como se hace para representar una figura en el espacio.

Pero ¿qué ocurriría si se quisiera utilizar un array de más de tres dimensiones? ¿Es esto posible? Sí, es perfectamente posible. Se pueden tener arrays con tantas dimensiones como se desee; la única limitación es la memoria disponible del equipo donde se trabaje. ¿Cómo puede imaginarse esa estructura de datos? Muy sencillo, partiendo de un array unidimensional y almacenando dentro de cada posición otro array unidimensional, y dentro de cada una de las posiciones de este segundo array otro array unidimensional, y así indefinidamente hasta completar todas las dimensiones necesarias.

La Figura 3.3 representa un **array multidimensional** (cada nivel de anidamiento supone una dimensión). La primera dimensión tiene seis elementos. Cada uno de esos seis elementos contiene a su vez un array de dos elementos, y cada uno de esos dos elementos contiene a su vez un array de tres elementos. Se puede seguir ahondando en la estructura y añadir más dimensiones hasta donde sea necesario. Acceder al elemento cuyo valor es 5 sería tan sencillo como indicarle a JavaScript el índice de cada dimensión, desde la más externa a la más interna, es decir, (2,1,2) en este caso.

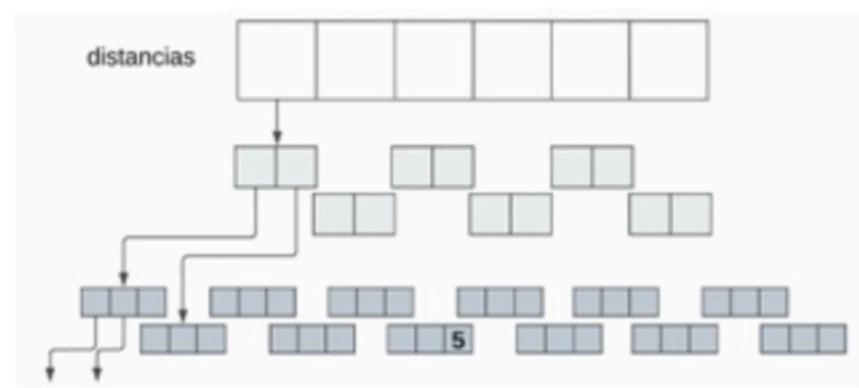


Figura 3.3. Diagrama conceptual de un array multidimensional.

Actividad propuesta 3.1

Arrays multidimensionales sobre el papel

Realiza un diagrama donde se pueda ver la estructura de un array de cuatro dimensiones, de 5x4x3x2 elementos numéricos de tipo entero con valores cualesquiera.

3.1.2. Creación y acceso a los elementos

Para crear un array puede utilizarse una de estas tres variantes:

```
let array1 = new Array();
let array2 = Array();
let array3 = [];
```

En los tres se crea un array vacío, sin elementos. Tomando la primera variante como primer objetivo, hay que entender qué ocurre en función de lo que se escriba en el interior de los paréntesis:

```
let array1 = new Array(2);
let array2 = new Array(3,4);
let array3 = new Array("Luis");
```

En la primera línea se crea un array con dos elementos, sin especificar el valor de los elementos que contiene. En la segunda línea se crea un array con dos elementos tomando el primer elemento el valor 3 y el segundo elemento el valor 4. La tercera línea crea un array con un solo elemento cuyo valor es "Luis". Exactamente igual ocurriría si se obvia la palabra reservada **new** en los tres casos. Pero este debate se posterga a cuando se estudie la programación orientada a objetos y los objetos predefinidos del lenguaje, más adelante en este libro.

En las siguientes líneas de ejemplo se ve lo que ocurre cuando se crean con la tercera variante, usando corchetes:

```
let array1 = [2];
let array2 = [1,3];
let array3 = ["Luis"];
```

En los tres casos se crean arrays unidimensionales con tantos elementos como valores aparezcan separados por comas. Es decir, en el primer caso se tiene un array con un solo elemento cuyo valor es 2. En el segundo caso se generan dos elementos con valores 1 y 3, respectivamente. Y en el tercer caso se tiene un único elemento cuyo valor es "Luis".

Para la explicación que viene a continuación se parte de la suposición de haber creado el array **edades** de cualquiera de estas tres formas (son equivalentes y significan lo mismo):

```
let edades = new Array (18,21,34,12,92);
let edades = Array (18,21,34,12,92);
let edades = [18,21,34,12,92];
```

El contenido de un array se puede ver de este modo:

```
console.log(edades);
```

En la consola del navegador se mostrará su estructura, contenido, longitud (número de elementos del array) y qué posición ocupa cada elemento (Figura 3.4).

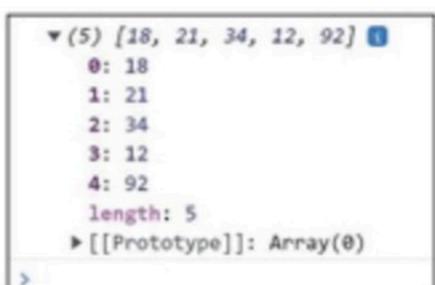


Figura 3.4. Información de un array ofrecida por la consola de Google Chrome.

En todos los casos es un array que contiene cinco elementos de tipo número entero. Acceder al primer elemento y cambiarle el valor se puede hacer directamente:

```
edades[0] = 111;
```

La modificación del valor del tercer elemento puede hacerse de este modo:

```
edades[2] = 998;
```

Así, el array **edades**, en este punto del programa, almacenaría lo que se muestra en la Figura 3.5.

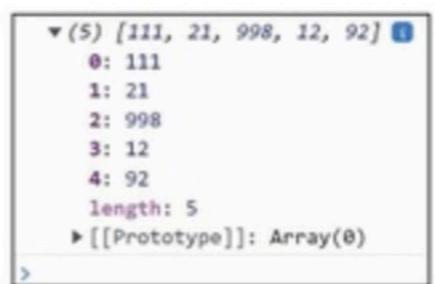


Figura 3.5. Contenido del array usando la consola de Google Chrome.

Importante

A los elementos de un *array* se accede contando las posiciones desde el 0. Si un *array* tiene cinco elementos, se accede a ellos utilizando los índices 0, 1, 2, 3 y 4.

Cuando se dice «acceder al tercer elemento», se está accediendo al elemento cuyo índice es 2.

Es muy importante interiorizar esto porque suele ser una fuente interminable de errores, incluso entre los programadores más experimentados.



Para mostrar un elemento se hace exactamente lo mismo: se piensa en la posición que ocupa dentro del *array*, se localiza su índice y se trata la expresión como si fuera una variable más. De esta forma se mostraría en pantalla el contenido del cuarto elemento:

```
console.log(edades[3]);
```

Durante la creación de arrays el uso de las comas, que separan los elementos del *array*, es muy importante. Podría darse el caso, por ejemplo, de querer almacenar en un *array* tres elementos, pero se desconoce el segundo de ellos:

```
let procesadores = ["Intel", "AMD"];
console.log(procesadores);
console.log(procesadores[1]);
```

Esta expresión no generaría ningún error, sino que almacenaría un valor ***undefined*** en la segunda posición (Figura 3.6).

```
* (3) ['Intel', vacío, 'AMD']
  0: "Intel"
  1: undefined
  2: "AMD"
  length: 3
  ▶ [[Prototype]]: Array(0)
undefined
```

Figura 3.6. Aspecto de elementos ***undefined*** (vacío) en la consola de Google Chrome.

Pero esto solo ocurre si la coma no es el último elemento. Si así fuera, la última coma se obviaría y no contaría como un nuevo elemento sin definir (Figura 3.7):

```
let procesadores = ["Intel", "AMD",];
```

En este ejemplo no se tendrían cinco elementos, sino cuatro, estando el primero y el tercero sin definir (Figura 3.7).

```
* (4) [vacío, 'Intel', vacío, 'AMD']
  0: vacío
  1: "Intel"
  2: vacío
  3: "AMD"
  length: 4
  ▶ [[Prototype]]: Array(0)
```

Figura 3.7. Contenido del array mostrado en la consola de Google Chrome.

De nuevo, hay que recordar que el primer y el tercer elementos son aquellos que ocupan las posiciones 0 y 2, respectivamente.

Actividad resuelta 3.1**Creación y manipulación de un array simple**

Construye un array que almacene el nombre de cinco localidades y muestra en pantalla aquellas que ocupen posiciones impares.

Solución

```
let localidades = ["Sanlúcar", "Chipiona", "Rota", "Barbate", "Tarifa"];
let i=0;
while (i<localidades.length) {
    if (i % 2) {
        console.log(localidades[i]);
    }
    i++;
}
```

Hasta ahora, todo lo estudiado sobre creación y acceso a los elementos de un *array* se ha circunscrito al caso de *arrays* unidimensionales. Es el momento de ver cómo se crea un *array* bidimensional con dos filas y tres columnas:

```
let tablaNotas = [[,,], [,,]];
```

Como puede verse, cada nivel de anidamiento de corchetes indica una dimensión del *array*, y el número de elementos separados por comas hace referencia al número de elementos de esa dimensión. En este caso hay una primera dimensión con dos elementos, y cada uno de ellos definen una segunda dimensión con tres elementos cada una.

Para almacenar valores en ese *array* hay que almacenar un valor en cada posición del *array*, es decir:

```
tablaNotas[0][0] = 1; // Fila 0 - Columna 0
tablaNotas[0][1] = 2; // Fila 0 - Columna 1
tablaNotas[0][2] = 3; // Fila 0 - Columna 2
tablaNotas[1][0] = 4; // Fila 1 - Columna 0
tablaNotas[1][1] = 5; // Fila 1 - Columna 1
tablaNotas[1][2] = 6; // Fila 1 - Columna 2
```

Sin embargo, esta forma de crear *arrays* multidimensionales es bastante engorrosa y nada práctica, primero porque el uso de las comas puede conducir a errores y segundo porque es prácticamente imposible automatizar la creación.

Para ello, es más útil utilizar otra de las variantes que ya se han visto, usando la palabra reservada ***new*** y creando el *array* con sucesivas llamadas a esa instrucción. Así se obtiene un *array* equivalente al anterior:

```
let tablaNotas = new Array(2);
tablaNotas[0] = new Array(3);
tablaNotas[1] = new Array(3);
```

A partir de aquí, se rellenan los valores de la misma forma que se ha visto anteriormente.

Pero, ¿qué ocurre si en vez de tener un array bidimensional de 2x3, como en el caso anterior, fuera necesario un array de cuatro dimensiones de 10x5x20x10 elementos? Crear e inicializar ese array tal y como se hizo anteriormente sería un trabajo muy tedioso, que consumiría mucho tiempo y cuya legibilidad y mantenimiento serían nulos.

Por eso se necesita alguna estrategia más eficiente que ayude a realizar este tipo de tareas. Es el momento de aprender a recorrer arrays.

Actividad propuesta 3.2

Colores RGB

Te han encargado desarrollar una aplicación web que permita modificar el coloreado de sus elementos (5 colores distintos) y para ello te han encargado crear alias de colores RGB para facilitarle al usuario la elección de los colores. Tu necesidad, por tanto, es crear una estructura de datos que almacene esta información:

- Naranja: #F39C12.
- Lima: #C0F312.
- Turquesa: #12F3E5.
- Rosa: #F312AF.
- Rojo: #F31212.

Crea la estructura de datos necesaria, rellénala con los datos aportados y muéstralos en pantalla.

Actividad propuesta 3.3

Números pares

Crea un array de 100 elementos yrellénalo con números aleatorios. Luego muestra en pantalla una lista con todos los números pares que contiene.

Nota: para generar números aleatorios desde 0 hasta MAX, puedes utilizar la expresión `Math.floor(Math.random()*MAX);`

Una desventaja de este bucle `for` está en que saca todos los elementos del array aunque haya posiciones en los que los elementos no tienen valor, ensuciando la salida. De manera que de tener este otro array la salida sería la que se muestra en la Figura 3.9:

```
let precios = [69.99,,12.49,,35.20,,99.90];
```

El precio 0 es: 69.99
El precio 1 es: undefined
El precio 2 es: undefined
El precio 3 es: 12.49
El precio 4 es: undefined
El precio 5 es: 35.2
El precio 6 es: undefined
El precio 7 es: undefined
El precio 8 es: 99.9

Figura 3.9. Salida del programa por la consola de Google Chrome.

Sin embargo, existe una opción más limpia y a la vez simplificada de recorrer un array, el llamado bucle `for..in`. Basta con indicar la variable que se usará para iterar sobre las posiciones del array y el nombre del propio array:

```
for (let i in precios) {
    console.log(`El precio ${i} es: ${precios[i]}`);
}
```

Como se puede ver, no es necesario inicializar el contador, ni controlar el tamaño del array, ni tampoco realizar el incremento del contador. Su funcionamiento es automático, y además no tiene en cuenta los elementos vacíos (Figura 3.10).

El precio 0 es: 69.99
El precio 1 es: 12.49
El precio 2 es: 35.2
El precio 3 es: 99.9

Figura 3.8. Salida del programa por la consola de Google Chrome.

El precio 0 es: 69.99
El precio 3 es: 12.49
El precio 5 es: 35.2
El precio 8 es: 99.9

Figura 3.10. Los elementos vacíos no se muestran en la salida.

Existe otra variante para recorrer un *array* que simplifica todavía más el proceso. Es el denominado bucle **for..of**. En este caso ni siquiera se utiliza una variable para iterar por cada posición, sino que se realiza automáticamente y de forma transparente para el usuario. Además, en cada iteración devuelve el valor de cada elemento del *array*:

```
let precios = [69.99,,12.49,,35.20,,99.90];
for (let precio of precios) {
    console.log(precio);
}
```

Su desventaja es que se desconocen los índices de las posiciones, y que en la salida aparecen también los elementos vacíos (Figura 3.11).

69.99
undefined
12.49
undefined
35.2
undefined
99.9

Figura 3.11. Los elementos repetidos se agrupan en burbujas.

¿Qué bucle **for** se debe usar entonces? Como siempre, la respuesta depende de las necesidades de cada momento. Con los ejemplos anteriores se ha visto a qué tiene acceso y qué muestra cada alternativa. Ante cada caso que se presente hay que pararse a pensar en qué se necesita y la elección de la variante óptima aparecerá por sí sola.

Todavía existe un bucle **for** adicional que estudiar, el bucle **forEach**, pero su análisis queda postergado hasta dominar el manejo de las funciones.

Aún queda una tarea que abordar con respecto a los recorridos: recorrer arrays de cualquier dimensión. La estrategia para estos casos es muy sencilla. Se trata de anidar un bucle **for** dentro de otro. ¿Cuántos niveles de anidamiento? Tantos como dimensiones tenga el *array*.

El siguiente *array* bidimensional de 4x3 servirá de ejemplo:

```
let matriz = [[5,1,2],[6,4,3],[9,3,8],[4,1,7]];
```

Sobre el papel, su estructura es la que se muestra en la Figura 3.12.

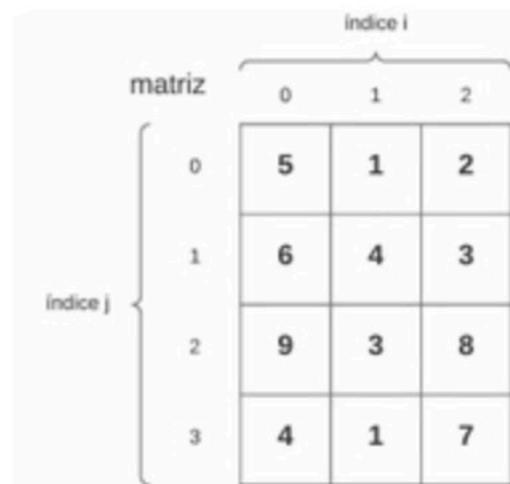


Figura 3.12. Representación conceptual de un array bidimensional en forma de tabla.

Y de este modo se efectuaría el recorrido por todos sus elementos para mostrarlos en pantalla:

```
for (let i=0; i<matriz.length; i++) {
    for (let j=0; j<matriz[i].length; j++) {
        console.log('Fila ${i} - Columna ${j}: ${matriz[i][j]}');
    }
}
```

Como se había adelantado, estamos ante un *array* de dos dimensiones, por tanto, en la solución deben aparecer dos bucles **for** anidados. Se utiliza un contador para recorrer los índices de cada dimensión (*i* para las filas y *j* para las columnas), teniendo la precaución de modificar la expresión que controla el número máximo de elementos del *array* (*matriz[i].length* para el caso de las filas, y *matriz[j].length* para el caso de las columnas).

Fila 0 - Columna 0: 5
Fila 0 - Columna 1: 1
Fila 0 - Columna 2: 2
Fila 1 - Columna 0: 6
Fila 1 - Columna 1: 4
Fila 1 - Columna 2: 3
Fila 2 - Columna 0: 9
Fila 2 - Columna 1: 3
Fila 2 - Columna 2: 8
Fila 3 - Columna 0: 4
Fila 3 - Columna 1: 1
Fila 3 - Columna 2: 7

Figura 3.13. Resultado de recorrer un array bidimensional.

Actividad resuelta 3.2

Diagonal de una matriz

Crea una matriz de 3x3, rellénala con números enteros y muestra en pantalla aquellos elementos que forman parte de la diagonal principal.

Solución

Para abordar la solución a este problema debes idear una estrategia que te permita detectar el criterio que cumplen los índices que forman la diagonal principal. Si dibujas sobre un papel la matriz rápidamente te darás cuenta de que en esas posiciones los índices son iguales, es decir, *i* y *j* coinciden:

```
let matriz = [[1,2,3],[4,5,6],[7,8,9]];
for (let i=0; i<3; i++)
    for (let j=0; j<3; j++)
        if (i==j)
            console.log('Elemento ${i}${j}: ${matriz[i][j]}');
```

Actividad propuesta 3.4

Liga local de fútbol

Quieres crear una estructura de datos que almacene la tabla de clasificación de una liga local de fútbol con estos datos:

- La liga la disputan 10 equipos.
- Para cada equipo necesitas estos datos:
 - Nombre.
 - Puntos.
 - Partidos jugados, ganados, empatados y perdidos.
 - Goles a favor y goles en contra.

Crea la estructura de datos que consideres más útil, rellénala con datos coherentes y muestra en pantalla toda la información de la clasificación del equipo ganador.

Con un array de, por ejemplo, cinco dimensiones, la estrategia de recorrido sería exactamente la misma:

```
for (let i=0; i<array5d.length; i++)  
  for (let j=0; j<array5d[i].length; j++)  
    for (let k=0; k<array5d[i][j].length; k++)  
      for (let l=0; l<array5d[i][j][k].length; l++)  
        for (let m=0; m<array5d[i][j][k][l].length; m++) {  
          console.log(array5d[i][j][k][l][m]);  
        }
```

3.1.4. Manipulación y operaciones con arrays

Con lo que ya se sabe de arrays se tiene la capacidad de resolver muchas tareas y desafíos del día a día. Sin embargo, supondría escribir miles de líneas de código para resolver tareas muy comunes cuyas soluciones ya se conocen y están completamente optimizadas.

Recuerda

Siempre que te plantees resolver una pequeña tarea sobre una estructura de datos, vale la pena investigar previamente si existe ya una solución óptima que la resuelve. Es muy probable que estés gastando tiempo y recursos *hardware* de forma innecesaria tratando de reinventar la rueda.

Estas son algunas de esas herramientas que proporciona el propio lenguaje y que ahorran una enorme cantidad de tiempo en cada proyecto.

Asignación de arrays

Una primera operación que puede realizarse con los arrays es una simple asignación. O no tan simple, porque su comportamiento no es el esperado.

Siempre que se realiza una asignación entre dos variables, lo que ocurre es que el contenido de la variable de la derecha «se copia» en la variable de la izquierda. Desde ese momento, se tienen dos variables independientes cuyos valores dependen únicamente de las operaciones en las que intervenga cada una de ellas, ¿no es cierto?

```
let unaVariable = 12;  
let otraVariable = unaVariable;  
otraVariable = otraVariable*2;  
console.log(unaVariable); //devuelve 12  
console.log(otraVariable); //devuelve 24
```

Pero ¿qué ocurriría de hacer esto mismo con unos arrays?

```
let sinIVA = [20.45,39.95,6.69];  
let conIVA = sinIVA;  
conIVA[0] = 110.25;  
console.log(sinIVA);  
console.log(conIVA);
```

Solo se ha modificado el array **conIVA**; por tanto, el array **sinIVA** debería permanecer inalterado. En cambio, la salida muestra lo que se aprecia en la Figura 3.14.

```
> (3) [110.25, 39.95, 6.69]
> (3) [110.25, 39.95, 6.69]
```

Figura 3.14. Resultado inesperado tras la ejecución de las instrucciones en las que se produce una asignación entre arrays.

¡Se han modificado los dos arrays! Lo que ha ocurrido es la propia esencia de los arrays, el concepto más importante que es necesario entender. El identificador de un array no es donde se almacena la estructura de datos, sino que es una referencia que apunta a la estructura de datos. De esta manera, cuando se asignan identificadores, lo que ocurre es que dos identificadores apuntan a la misma estructura de datos, por lo que cualquier modificación que se haga utilizando cualquiera de los dos identificadores estará afectando a la misma estructura de datos.



Figura 3.15. Dos identificadores apuntando a la misma estructura de datos.

Adición de elementos a un array

Existen varias formas de hacer esto. Si se sabe cuántos elementos tiene un array se le puede indicar directamente que guarde un nuevo elemento al final:

```
let elementos = ["a",7,true];  
elementos[3] = 23.45;  
// Resultado ["a",7,true,23.45]
```

Si se desconoce el número de elementos que tiene, puede utilizarse **push**, que directamente añade un elemento al final (y además devuelve el número de elementos del array tras la inserción).

```
let elementos = ["a",7,true];
elementos.push("xyz");
// Resultado ["a",7,true,"xyz"]
```

Por último, para añadir un elemento al principio se utiliza **unshift**:

```
let elementos = ["a",7,true];
elementos.unshift("el primero");
// Resultado ["el primero","a",7,true,"xyz"]
```

También es interesante señalar que tanto **unshift** como **push** permiten añadir en una misma instrucción varios valores, tantos como interese:

```
let elementos = ["a",7,true];
elementos.unshift("primero","segundo");
elementos.push("penúltimo","último");
// Resultado ["primero","segundo","a",7,true,"penúltimo","último"]
```

Actividad resuelta 3.3

Insertar elementos en un array

Crea un array vacío de cinco posiciones llamado **vector** e inserta el cuadrado de cinco números enteros aleatorios entre 0 y 10. Luego muestra en pantalla el contenido del array.

Solución

```
let vector = new Array(5);
for (let i=0; i<5; i++)
    vector[i] = Math.floor(Math.random()*10)**2;
for (let i=0; i<5; i++)
    console.log(vector[i]);
```

Eliminación de elementos de un array

Una primera aproximación para eliminar elementos consiste en utilizar los homólogos a **unshift** y **push**, que son **shift** (elimina el primer elemento) y **pop** (elimina el último elemento). En ambos casos se devuelve el elemento eliminado y, en caso de no haber elementos, se devuelve **undefined**.

```
let elementos = ["a",7,true];
elementos.shift();
elementos.pop();
// Resultado [7]
```

Al modificar la propiedad **length** se eliminan todos aquellos elementos que se quedan fuera de la nueva longitud del array:

```
let elementos = ["a",7,true,90.54,"Lucía",12];
elementos.length = 2;
// Resultado ["a",7]
```

También se puede eliminar la cantidad de elementos indicada desde una posición determinada usando **splice**. Además, devuelve un array con los elementos eliminados. Por ejemplo, la siguiente instrucción elimina los dos primeros elementos que encuentra desde la posición 3. Es decir, elimina los elementos que ocupan las posiciones 3 y 4.

```
let elementos = ["a",7,true,90.54,"Lucía",12];
let eliminados = elementos.splice(3,2);
// elementos -> ["a",7,true,12]
// eliminados -> [90.54,"Lucía"]
```

Actividad propuesta 3.5

Operaciones sobre arrays

Una aplicación de análisis de datos en la que estás trabajando necesita crear una estructura que almacene 10 múltiplos de 5 aleatorios de 0 a 100. Además, ni el primer elemento puede ser menor de 50, ni el último mayor de 50.

Se pide que resuelvas el problema siguiendo estos pasos:

- Crea un array de 10 elementos.
- Inicializa el array con múltiplos aleatorios de 5.
- Si el primer elemento es menor de 50, debes eliminarlo e insertar otro múltiplo de 5 aleatorio. Esta operación debes realizarla hasta que el primer elemento sea mayor o igual que 50.
- Si el último elemento es mayor de 50, debes eliminarlo e insertar otro múltiplo de 5 aleatorio. Esta operación debes realizarla hasta que el último elemento sea menor o igual que 50.
- Muestra el contenido del array en pantalla y comprueba que se cumplen todos los requisitos.

Nota: para generar números aleatorios desde 0 hasta MAX, puedes utilizar la expresión **Math.floor(Math.random()*MAX)**;

Concatenación de arrays

concat permite extender un array añadiéndole al final el contenido de otro array. Ninguno de los dos arrays se modifica, sino que se crea uno nuevo con el contenido de ambos. El array cuyos elementos aparecerán en primer lugar será aquel que haga uso de **concat**:

```
let original = ["a",7,true,90.54,"Lucía",12];
let nuevo = [90,80,70];
let extendido = original.concat(nuevo);
// original -> ['a', 7, true, 90.54, 'Lucía', 12]
// nuevo -> [90, 80, 70]
// extendido -> ['a', 7, true, 90.54, 'Lucía', 12, 90, 80, 70]
```

Copia de arrays

Los arrays se pueden copiar completos o traer una parte de ellos usando **slice**. Si no se indican parámetros, se copia todo el contenido del array en otro array. Si lo que se

quiero hacer es copiar solo una parte, hay que indicar dos parámetros: el primero será el índice de la posición desde la que se desea copiar (incluido el elemento que ocupa esa posición), y el segundo el índice de la posición hasta la que se quiere copiar (no incluido el elemento que ocupa esa posición).

```
let original = ["a",7,true,90.54,"Lucía",12];
let completo = original.slice();
let parcial = original.slice(2,4);
// original -> ['a', 7, true, 90.54, 'Lucía', 12]
// completo -> ['a', 7, true, 90.54, 'Lucía', 12]
// parcial -> [true, 90.54]
```

Búsqueda de elementos en un array

Existen varias opciones para realizar esta tarea en función de lo que se necesite exactamente.

La primera de ellas es `indexOf`, que devuelve el índice de la posición que ocupa el primer elemento que ha encontrado o `-1` si no lo ha encontrado. También permite indicar desde qué posición se desea buscar.

```
let pajar = ["a",7,true,50.54,7,"Marcos"];
let aguja = 7;
let resultado = pajar.indexOf(aguja);
// resultado -> 1

aguja = 8;
resultado = pajar.indexOf(aguja);
// resultado -> -1

aguja = 7;
resultado = pajar.indexOf(aguja,2);
// resultado -> 4
```

El método `lastIndexOf` realiza exactamente las mismas tareas que `indexOf`, pero trabajando desde el extremo derecho del `array`, no desde el extremo izquierdo. Si no se necesita conocer la posición del elemento, sino simplemente si el elemento existe en el `array`, puede utilizarse `includes`. Devuelve `true` si ha encontrado al menos una coincidencia o `false` en caso contrario.

```
let pajar = ["a",7,true,50.54,7,"Marcos"];
let aguja = 7;
let resultado = pajar.includes(aguja);
// resultado -> true
```

Actividad propuesta 3.6

Buscando la verdad

Crea una aplicación que almacene en un `array` diez elementos. Rellena cada posición del `array` con uno de los dos valores permitidos: `true` o `false`. Luego muestra en pantalla la posición que ocupa cada elemento `true`.

Nota: no puedes utilizar ningún bucle `for`.

Ordenación de arrays

La ordenación de `arrays` es otra de esas tareas comunes que no trae a cuenta programar a mano, ya que existe un método muy potente llamado `sort` que permite realizar cualquier ordenación que se necesite. Sin embargo, por el momento se estudiará su aplicación más simple, ya que su verdadera potencia radica en el uso de las funciones (que se verán más adelante).

```
let vector = [8,4,5,7,1];
vector.sort();
// vector -> [1, 4, 5, 7, 8]
```

Si en lugar de números los elementos del `array` almacenan cadenas de caracteres, ocurre algo inesperado que siempre hay que tener en cuenta:

```
let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort();
// ordenación esperada -> ['casa', 'Casado', 'ñam', prueba', 'zancos']
// ordenación obtenida -> ['Casado', 'casa', 'prueba', 'zancos', 'ñam']
```

La ordenación, efectivamente, se realiza por orden alfabético, pero teniendo en cuenta (y esto es muy importante) el lugar que ocupa cada carácter en la tabla Unicode. Por ese motivo **Casado** se ordena antes que **casa**, porque la **C** aparece antes que la **c**. Igualmente ocurre con la **ñ** (y con la **Ñ**), que aparece en la tabla Unicode mucho después que el resto de las letras, por no ser un carácter anglosajón.

Para saber más

En el enlace <https://unicode.org> o con el código QR se accede a la lista completa de la tabla Unicode. Es muy recomendable revisarla para entender muchos de los resultados inesperados que a veces se producen al trabajar con cadenas de caracteres y otros símbolos especiales.



Para solventar este contratiempo y modificar el comportamiento predeterminado de la ordenación, pueden establecerse criterios propios. Aunque, como se introdujo más arriba, es una tarea que se abordará cuando se dominen las funciones.

Otro método que suele resultar útil es `reverse`, que resuelve la necesidad de darle la vuelta a un `array`, invertirlo.

```
let vector = ["Casado","García","Martínez","López","Ballén"];
vector.sort();
// ordenación directa -> ['Ballén', 'Casado', 'García', 'López', 'Martínez']
vector.reverse();
// ordenación inversa -> ['Martínez', 'López', 'García', 'Casado', 'Ballén']
```

Existe una larga lista de utilidades más que ofrece el propio lenguaje para trabajar con `arrays`. Se puede acceder a la documentación oficial del lenguaje para descubrirlas. Además, en repositorios como GitHub o Stackoverflow, se pueden encontrar muchas otras que, en ocasiones, incluso mejoran el rendimiento.

3.2. Conjuntos

Los conjuntos o sets (como así se llaman en JavaScript) son estructuras de datos muy parecidas a los arrays pero con la particularidad de que no permiten valores duplicados. Además, esta prevención ante la duplicidad de elementos se realiza de forma automática sin necesidad de invertir esfuerzo en ello.

3.2.1. Creación de un conjunto

Para crear un conjunto se puede utilizar el operador **new** (en cuyo estudio se profundizará más adelante), ya que se trata de un objeto:

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

Para indicarle, desde su declaración, los elementos que lo componen inicialmente, es preciso pasarle un objeto de tipo iterable (*array, map, string, set*):

```
var conjunto1 = new Set([34,1,"Girasol",25.9]);
var conjunto2 = new Set("cadena");
```

```
▶ Set(4) {34, 1, 'Girasol', 25.9}
▶ Set(5) {'c', 'a', 'd', 'e', 'n'}
```

Figura 3.16. Eliminación automática de duplicados en los conjuntos.

En el ejemplo anterior se creó un conjunto a partir de un *array* y otro a partir de un *string*. En el segundo ejemplo puede comprobarse cómo automáticamente ha eliminado los elementos duplicados (la última «a»).

Actividad resuelta 3.4

Conjuntos a partir de arrays

Construye un conjunto a partir de un *array* que contiene los días de la semana. Utiliza alguna instrucción para comprobar la estructura del conjunto en la consola del navegador.

Solución

```
let dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];
let conjunto = new Set(dias);
console.info(conjunto);
```

3.2.2. Recorrido

El recorrido de un conjunto puede realizarse con el ya conocido **for..of**, tal y como se hacía en el caso de los *arrays*:

```
var conjunto = new Set(["primero","segundo","tercero","primero"]);
for (let elemento of conjunto) {
    console.log(elemento);
}
```

primero
segundo
tercero

Figura 3.17. Salida del bucle de recorrido de un conjunto en la consola.

3.2.3. Manipulación y operaciones con conjuntos

Existen muchos métodos asociados a la operativa de conjuntos dada su enorme utilidad, pero, por razones de espacio disponible, se estudiarán los que se utilizan con más frecuencia.

Adición de elementos

Para añadir elementos se utiliza el método **add**, indicándole el elemento que se desea añadir. Además, también permite que estas inserciones se encadenen:

```
var conjunto = new Set();
conjunto.add(7);
conjunto.add("Samuel").add(69).add("moteros");
// conjunto {7, 'Samuel', 69, 'moteros'}
```

Actividad propuesta 3.7

Creación de conjuntos

Crea un conjunto vacío y añádele todos los elementos de los *arrays* [12,12,12,14], [11,11,13,15], ["i","j","k","l"]. ¿Cuántos elementos contiene el conjunto? ¿Por qué? Haz un recorrido del conjunto, muestra en pantalla cada uno de sus elementos y razona por qué aparecen en ese orden.

Eliminación de elementos

La eliminación de un elemento concreto del conjunto puede realizarse con el método **delete**, que devuelve **true** o **false** para indicar el resultado de la operación.

```
conjunto.delete(69);
// conjunto {7, 'Samuel', 'moteros'}
```

Para eliminar todos los elementos de un conjunto no es necesario llamar a **delete** tantas veces como elementos haya. Puede usarse el método **clear**:

```
conjunto.clear();
// conjunto {}
```

Tamaño de un conjunto

Calcular el tamaño de un conjunto equivale a conocer el número de elementos que lo forman. La propiedad **size** se encarga de ello:

```
var conjunto = new Set().add(1).add(1).add(2).add(9);
console.info(conjunto.size);
// 3
```

Búsqueda de un elemento

Otra utilidad interesante es la que proporciona el método **has**: comprueba que un valor, o el resultado de una expresión, está en el conjunto. Devuelve un valor booleano para indicar si encontró el elemento o no.

```
var conjunto = new Set().add(1).add(1).add(2).add(9);
if (conjunto.has(9))
    console.log("Encontrado");
// Encontrado
```

Conversiones

Al inicio de la sección de conjuntos se vio que una de las formas de crear un conjunto era pasándole directamente un **array** con una lista de valores. La conversión en ese caso se hacía de forma automática.

Pero también puede realizarse la tarea opuesta, convertir un conjunto en un **array**. Para ello se utiliza un nuevo operador: el operador **de arrastre, de propagación o spread**. Se utiliza escribiendo unos puntos suspensivos (...) seguidos del nombre del conjunto que se desea convertir. Devuelve el **array** resultado de la operación:

```
var conjunto = new Set().add(1).add(1).add(2).add(9);
const vector = [...conjunto];
// vector [1, 2, 9]
```

Actividad resuelta 3.5

Eliminación de duplicados en los arrays

Utiliza los conceptos de conjuntos para eliminar los elementos duplicados de un **array**.

Solución

```
let arrayConDuplicados = [1,7,4,7,7,2];
let conjunto = new Set(arrayConDuplicados);
let arraySinDuplicados = [...conjunto];
console.info(arraySinDuplicados);
```

Unión

A veces resulta útil crear un conjunto a partir de dos o más **arrays** que contienen elementos de interés. Con el operador **spread** y la conversión automática de **arrays** en conjuntos eliminando los duplicados se puede ahorrar una buena cantidad de líneas de código:

```
let array1 = [10,20,30,40,50];
let array2 = [30,50,60,70,80];
let array3 = [60,70,80,90,100];
var conjunto = new Set([...array1,...array2,...array3]);
// conjunto {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
```

3.3. Mapas

Al estudiar los **arrays** se vio que los elementos se organizaban en posiciones numeradas automáticamente, desde el índice 0 hasta el máximo número de elementos menos uno. Después se vio que los conjuntos no permiten la manipulación de las posiciones, eliminan automáticamente los duplicados y ofrecen cierta operativa útil para su manipulación. En este caso, los mapas de alguna manera combinan las bondades de las dos estructuras anteriores, ya que permiten tener una colección de datos organizados en forma de pares «clave-valor», en las cuales las claves no se pueden repetir.

3.3.1. Creación de un mapa

Para crear un mapa, que también es un objeto, se vuelve a recurrir al operador **new**:

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

Al igual que en el caso de los conjuntos, se puede crear un mapa cuyos valores iniciales los aporte un **array**. Un pequeño listín telefónico servirá de ejemplo:

```
const telefonos = new Map([
    [615885225,"Elena"],
    [663998541,"Quirós"],
    [656232511,"Marta"],
    [696585537,"David"]
]);
```

```
▶ 0: {615885225 => "Elena"}
▶ 1: {663998541 => "Quirós"}
▶ 2: {656232511 => "Marta"}
▶ 3: {696585537 => "David"}
size: 4
```

Figura 3.18. Información sobre la estructura de un mapa en la consola de Google Chrome.

3.3.2. Recorrido de un mapa

Como en el caso de los conjuntos, el bucle más interesante para recorrer un mapa es el bucle `for..of`. Pero debe utilizarse con cuidado. Al realizar el recorrido ya conocido se obtiene lo que se muestra en la Figura 3.19:

```
const telefonos = new Map([
  [615885225,"Elena"],
  [663998541,"Quirós"],
  [656232511,"Marta"],
  [696585537,"David"]
]);
for (persona of telefonos)
  console.log(persona);
```

Se obtiene el par completo «clave-valor» de cada uno de los elementos que forman el mapa:



Figura 3.19. Salida del bucle que muestra todos los elementos del mapa.

Lo cual, no es demasiado útil para tratar los datos por separado. Para obtener en distintas variables las claves y los valores de cada elemento, puede realizarse la siguiente modificación al bucle anterior:

```
for (let [telefono,persona] of telefonos)
  console.log(`El teléfono de ${persona} es ${telefono}.');
```

De este modo, en las variables **teléfono** y **persona** se tienen todas las entradas del mapa (Figura 3.20).

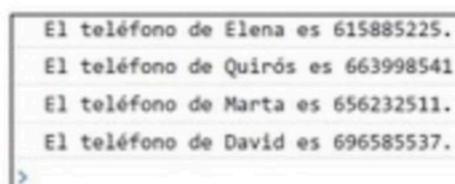


Figura 3.20. Salida del recorrido de un mapa tras utilizar un bucle `for..of` modificado con dos variables.

Además, si solo se tiene que trabajar con las claves o solo con los valores, también pueden usarse dos métodos que lo permiten: **keys** y **values**.

```
for (let telefono of telefonos.keys())
  console.log(telefono);
```

```
for (let persona of telefonos.values())
  console.log(persona);
```

615885225
663998541
656232511
696585537

Figura 3.21. Salida con las claves usando `keys()`.

Elena
Quirós
Marta
David

Figura 3.22. Salida con los valores usando `values()`.

3.3.3. Manipulación y operaciones con mapas

De la misma manera que con las dos estructuras de datos anteriores, a continuación se estudiarán algunas de las operaciones más útiles que permiten manipular el contenido de los mapas.

Adición de elementos

Para añadir elementos a un mapa se utiliza el método `set`, al que hay que indicarle tanto la clave como el valor del elemento. También pueden encadenarse las inserciones. Si se añaden varios valores con la misma clave, hay que tener en cuenta que solo permanecerá en el mapa aquel par que se haya insertado más tarde.

```
const telefonos = new Map();
telefonos.set(615885225,"Elena");
telefonos.set(615885225,"Elena").set(777777777,"Quirós");
telefonos.set(656232511,"Marta").set(777777777,"Sara");
telefonos.set(696585537,"David");
```

Se han añadido seis elementos al mapa; sin embargo, en la salida solo aparecen correctamente insertados cuatro (Figura 3.23).

> 0: {615885225 => "Elena"}
> 1: {777777777 => "Sara"}
> 2: {656232511 => "Marta"}
> 3: {696585537 => "David"}

size: 4

Figura 3.23. Resultado tras la inserción de elementos en el mapa.

La razón que lo justifica está en la propia naturaleza de los mapas: la clave correspondiente a **Elena** estaba duplicada, por lo que un elemento ha machacado al otro. Exactamente la misma lógica ha desencadenado la pérdida de **Quirós**, que al insertarse con la misma clave que **Sara** y ser esta más reciente es el elemento que ha perdurado.

■■■ Eliminación de elementos

Eliminar elementos es una tarea tan simple como llamar al método `delete` con la clave del elemento que se quiere eliminar:

```
telefonos.delete(777777777);
```

```
▶ 0: {615885225 => "Elena"}
▶ 1: {656232511 => "Marta"}
▶ 2: {696585537 => "David"}
size: 3
```

Figura 3.24. Contenido del mapa tras eliminar un elemento.

■■■ Búsqueda de elementos

Para buscar elementos se utiliza el método `has` con la clave del elemento que se desea encontrar. Si lo encuentra devolverá `true` y, en cualquier otro caso, `false`.

```
if (telefonos.has(666555222))
    console.log("¡Encontrado!");
else
    console.log("No está.");
// No está.
```

■■■ Lectura de valores

La utilidad que proporciona el método `get` es conocer el valor asociado a la clave especificada como parámetro. Se trata de una operación muy utilizada y cuya eficiencia es muy alta:

```
console.log(telefonos.get(656232511));
```

Actividad propuesta 3.8

Mapa de identidad

Crea un mapa vacío y añade los DNI de diez personas ficticias, usando el DNI como clave y el nombre como valor. A continuación, muestra en pantalla la lista de todos los DNI junto con el nombre de las personas. Modifica el nombre de la tercera persona y vuelve a mostrar todos los datos en pantalla para comprobar que la operación se ha realizado correctamente.

■■■ Conversiones

Al igual que en el caso de los conjuntos, también es posible obtener un `array` a partir del contenido de un mapa. Además, se utiliza el mismo operador que con los conjuntos, el operador `spread`, para obtener un `array` de `arrays`:

```
console.log([...telefonos]);
```

```
▼ (3) [Array(2), Array(2), Array(2)] ⓘ
▶ 0: (2) [615885225, 'Elena']
▶ 1: (2) [656232511, 'Marta']
▶ 2: (2) [696585537, 'David']
length: 3
```

Figura 3.25. Array de arrays resultado de aplicar la conversión con el operador de propagación.

Actividad propuesta 3.9

Ordenación de un mapa

Crea un mapa con cinco pares de elementos de manera que tanto las claves como los valores sean cadenas de caracteres. Luego idea algún mecanismo para ordenar alfabéticamente los valores del mapa. Muestra el resultado en pantalla.

Como se ha visto a lo largo de esta unidad, las estructuras de datos son colecciones de elementos que permiten crear, acceder y operar con los datos de una forma mucho más rápida, eficiente y segura que hacerlo de forma manual.

Antes de empezar a escribir los programas es necesario realizar una reflexión previa sobre qué estructura de datos es más adecuada para el fin que se persigue: ¿hay que acceder a las posiciones de cada elemento? ¿Puede haber valores duplicados? ¿Interesa personalizar los índices? ¿Se tiene acceso a propiedades y métodos que faciliten su manipulación? Las respuestas a preguntas como estas deben conducir a elegir la estructura de datos óptima en cada caso.

Finalmente, es importante destacar que las estructuras de datos estudiadas en esta unidad representan las más utilizadas por una generalidad de programadores, pero no son, ni mucho menos, todas las disponibles en el lenguaje. Existen muchas estructuras más, que se irán estudiando a lo largo de esta obra tan pronto como se adquieran los conocimientos necesarios para entenderlas en profundidad.

Para saber más

Para profundizar en el estudio de las tres estructuras de datos vistas en esta unidad puedes revisar estas referencias:

Arrays:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

Conjuntos:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Set

Mapas:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Arrays:



Conjuntos:



Mapas:

