4.2 ARRAYS

4.2.1 ¿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 todos estos nombres tienen connotaciones que hacen que se puedan confundir con otras estructuras de datos. Por ello, es más popular el nombre sin traducir al castellano: array.

Los arrays aparecieron en la ciencia de la programación de aplicaciones para solucionar un problema habitual al programar. Para entender este problema, supongamos que deseamos almacenar 25 notas de alumnos para ser luego manipuladas dentro del código JavaScript. Sin arrays necesitamos 25 variables distintas. Manejar esos 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 un dato individual dentro del array, hay que indicar su posición. La posición es un número entero conocido 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 son totalmente dinámicos, su tamaño se puede modificar después de la declaración a voluntad.

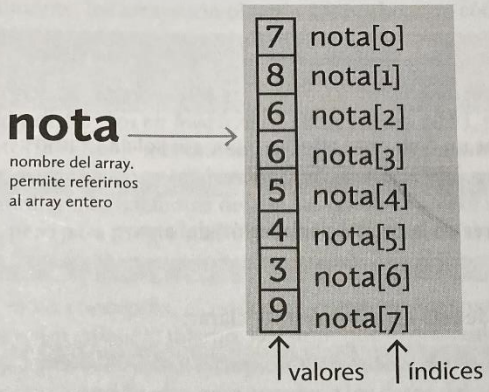

Figura 4.1: Ejemplo de array que almacena notas

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.

4.2.2 CREACIÓN DE ARRAYS

4.2.2.1 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 asignación, significan array vacío. Un array que se llama simplemente ***a*** (que, por cierto, es un nombre muy poco recomendable) 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 cero, 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 diez 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 (explicaremos los fundamentales más adelante) para mostrar el valor del array en la forma que deseemos.

4.2.2.2 USO DE CONST Y LET CON ARRAYS

Otro detalle a comentar es que es muy habitual declarar arrays con la palabra clave **const** en lugar de con **let**. En el tema anterior hemos explicado (3.3.2.1 *"Diferencia entre let y var"*, en la página 73) la diferencia entre ambos términos. Pero con los arrays hay matices a tener en cuenta. Por ejemplo:

```
const datos=[4.5,6.78,7.12,9.123];
```

Hemos declarado un array de cuatro 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 con profundidad más adelante (Unidad 6 *"Programación de Objetos en JavaScript"*, en la página 203), pero ya 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. A los programadores en C++ les puede ser familiar que hablemos de que las variables objeto son referencias a la información que contienen, pero para programadores que comienzan a trabajar con objetos, esta idea puede resultar muy confusa.

Pero, sin entrar en estos conceptos, sí podemos entender que cuando se crea un objeto la variable que lo contiene es un enlace al mismo. Digamos que lo que realmente contiene esa variable es cómo llegar al objeto. 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 mejor 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 ese 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.

4.2.2.3 OPERACIÓN DE ASIGNACIÓN CON ARRAYS

Aún 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 dos 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.

4.2.2.4 VALORES INDEFINIDOS

Veamos este código:

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

En la definición del array: `let a=[“Saúl","Rocío"];` estamos creando el array llamado ***a*** y dando valor a los dos primeros elementos (***a[0]*** y ***a[1]***). Luego, damos valor al cuarto elemento del array (`a[3]="María"`). 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 dos comas seguidas en la definición (después del valor ***"María"***) hacen que el tercer elemento (que sería ***a[2]***) se ignore.

4.2.2.5 ELIMINAR ELEMENTOS DE UN ARRAY

Para borrar un elemento se utiliza la palabra **delete** tras la 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 (aparece el texto "***<1 empty ítem>***" en su lugar). Este elemento pasa a ser indefinido.

4.2.2.6 ARRAYS HETEROGÉNEOS

Lo hemos comentado antes, los arrays de JavaScript son heterogéneos. Admiten mezclar en el mismo array valores de diferente tipo:

```
const a=[3,4,"Hola",true,Math.random()];
```

El array contiene dos números enteros (el 3 y el 4), un valor booleano (**true**) y un número decimal (resultado de la expresión `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 esta instrucción es perfectamente posible:

```
console.log(b[3][1]);//Escribe 55
```

En el ejemplo anterior, ***b[3][1]*** hace referencia al segundo elemento del cuarto elemento de ***b***. En definitiva, las posibilidades de uso de arrays en JavaScript para almacenar datos son espectaculares.

4.3 RECORRER ARRAYS

4.3.1 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=1;i<notas.length;i++){
    console.log(`La nota ${i} es ${notas[i]}`);
}
```

Ahora el array de notas contiene muchos elementos indefinidos. Si se deja así, el código mostrará lo siguiente cuando se ejecute:

```
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 de esta forma:

```
const notas=[5,6,,,,9,,,8,,9,,7,8];
for(let i=1;i<notas.length;i++){
    if(notas[i]!=undefined){
       console.log(`La nota ${i} es ${notas[i]}`);
    }
}
```

El bloque **if** permite comprobar primero que el elemento está definido y, solo si es así se muestra. El resultado sería el siguiente:

```
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
```

4.3.2 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.

4.3.3 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 **for..in**). Ejemplo:

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

El resultado es:

```
5
6
undefined
undefined
undefined
9
undefined
undefined
8
undefined
9
undefined
7
8
```

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

```
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.