

5.4 USO AVANZADO DE FUNCIONES

5.4.1 LA PILA DE FUNCIONES

Cuando se invoca a una función en una expresión, esta debe esperar a que la función finalice para poder completar la expresión. Ejemplo:

```
function f1(){
    console.log("Inicio f1");
    f2();
    console.log("Fin f1");
}

function f2(){
    console.log("Inicio f2");
    f3();
    console.log("Fin f2");
}

function f3(){
    console.log("En f3");
}

f1();
```

El resultado es:

```
Inicio f1
Inicio f2
En f3
Fin f2
Fin f1
```

PILA DE LLAMADAS

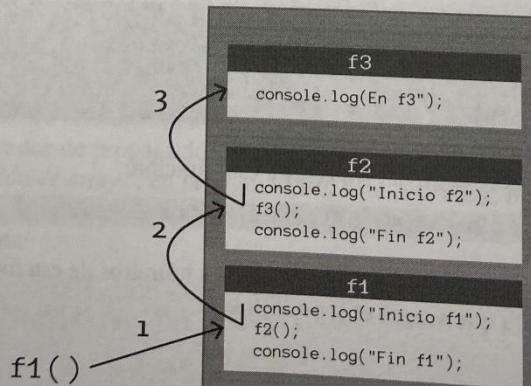


Figura 5.1: Ejemplo de pila de llamadas

Se observa claramente que la primera función (**f1**) no finaliza hasta que las otras llamadas han finalizado. Es decir, las funciones utilizan lo que se conoce como **pila de llamadas** que permite que, el intérprete JavaScript sepa qué funciones se deben de resolver antes.

En la Figura 5.1, podemos observar como se apilan las funciones en la pila de llamadas. La última función invocada queda en la cima. A medida que se resuelva el código de las últimas funciones se irán retirando de la pila a la vez que se devuelve el flujo a la función anterior.

PILA DE LLAMADAS

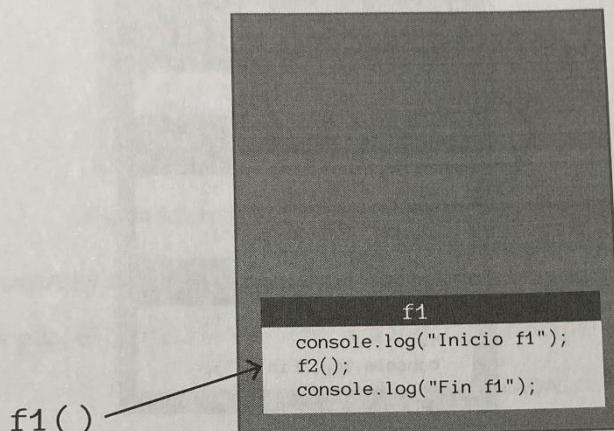


Figura 5.2: La función f1 se coloca en la pila de llamadas

En el código anterior el proceso sería el siguiente:

- [1] Se invoca a **f1**. El código de esta función se coloca en la pila de llamadas (Figura 5.2).

PILA DE LLAMADAS

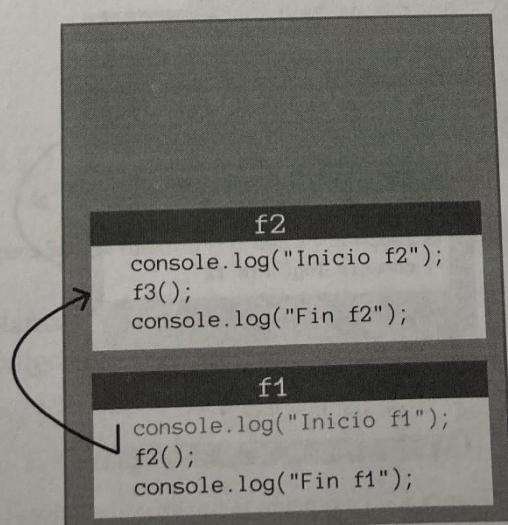


Figura 5.3: La función f2 se coloca en la pila de llamadas

- Inicio f1**
- [2] Se interpreta el código de f1. Se ejecuta la escritura del texto **Inicio f1**.
- [3] Se invoca a f2. Este código se pone en la cima de la pila (Figura 5.3) Se queda f1 a la espera de que se resuelva f2.
- [4] Se escribe **Inicio f2**.
- [5] Se invoca a f3. Su código pasa a ocupar la cima de la pila. f2 se queda esperando que se resuelva el código de f3.

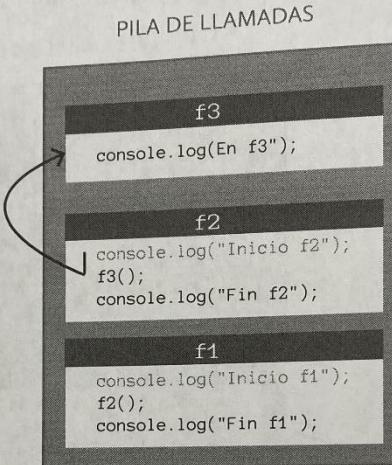


Figura 5.4: La función f3 se coloca en la pila de llamadas

- [6] Se escribe **En f3**.
- [7] La función f3 finaliza, devuelve el control a f2 y f3 se retira de la pila.

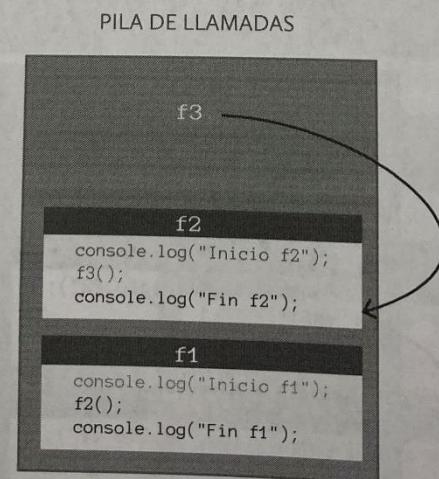


Figura 5.5: La función f3 finaliza y devuelve el control

- [8] f2 recupera el control y escribe el mensaje **Fin f2**.

[9] Como f2 ha finalizado, devuelve el control a f1 y se retira de la pila.

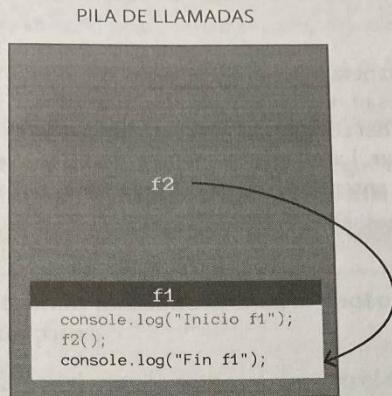


Figura 5.6: La función f2 finaliza y devuelve el control

[10] f1 recupera el control y escribe **Fin f2**.

[11] f1 se retira de la pila, el control se devuelve a la función principal

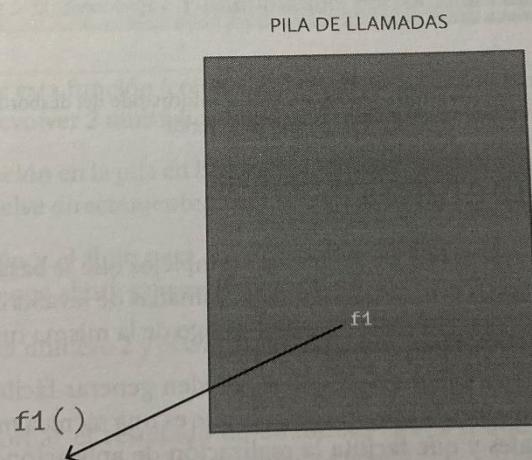


Figura 5.7: La función f1 finaliza y devuelve el control

Ante una mala gestión de llamadas se puede llegar a provocar el famoso **desbordamiento de pila**. Veamos un ejemplo de código que lo produciría:

```
function saludo(){
    console.log("Saludo");
    despedida();
}
function despedida(){
    console.log("Despedida");
```

```

        saludo();
    }
saludo();

```

Lo que ocurrirá es que de manera indefinida se llamarán una función a la otra en una especie de bucle infinito. Pero todos los motores de JavaScript (tanto node.js como los navegadores, por ejemplo) cortarán la ejecución del código tras una serie de llamadas debido a que se detecta que la pila de llamadas se va a llenar. La pila tiene un tamaño máximo y eso protege de problemas mayores, sin ese tope el código anterior provocaría efectos más devastadores.

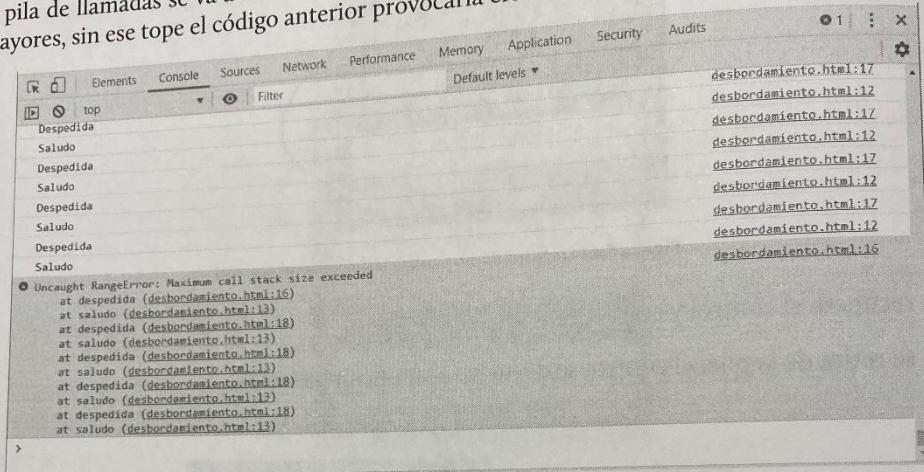


Figura 5.8: Ventana de depuración de Google Chrome informando del desbordamiento de pila que provoca el código anterior

5.4.2 RECURSIVIDAD

Hay una técnica de resolución de problemas complejos que se basa en la capacidad que tienen las funciones de invocarse a sí mismas. La pila de llamadas de JavaScript admite esta posibilidad y lo que ocurrirá es que aparecerá varias veces el código de la misma función en la pila.

Aunque esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas y propiciar un desbordamiento de pila, lo cierto es que es una técnica muy interesante que permite soluciones muy originales y que facilita la realización de aplicaciones sencillas para solucionar problemas muy complejos.

La idea es que cada invocación a la función resuelva parte del problema y se llame a sí misma para resolver la parte que queda del problema, y así sucesivamente. En cada llamada el problema debe ser cada vez más sencillo hasta llegar a una llamada, en la que la función devuelve un único valor. Es fundamental preparar bien esa última llamada ya que es la que cierra el bucle de llamadas y tras ella se irán resolviendo las anteriores hasta liberar la pila y conseguir el resultado final.

La recursividad se entiende mejor con ejemplos. En este sentido hay un ejemplo clásico: se trata del factorial. El factorial de un número entero se calcula multiplicando todos los números enteros anteriores. Así el factorial de 5 (que se denota matemáticamente como $5!$) es el resultado

de 5·4·3·2·1. Pero podemos entender también que $5!$ es lo mismo que $5·4!$ (cinco por el factorial de cuatro) y eso permite una solución muy creativa al problema:

```
function factorial(n){  
    if(n<=1) return 1;  
    else return n*factorial(n-1);  
}
```

La última instrucción `return n*factorial(n-1)` es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al uno, que devuelve uno.

Así si invocamos a esta función mediante el código: `factorial(4)`, la ejecución del programa generaría los siguientes pasos:

- [1] Se llama a la función factorial usando como parámetro el número 4 que será copiado en el parámetro `n`.
- [2] Se comprueba si `n` es mayor que uno. Como lo es, entonces se devuelve 4 multiplicado por el resultado de la llamada `factorial(3)`. Hemos, por lo tanto, de resolver el factorial de 3.
- [3] La invocación anterior añade un nuevo código a la pila en el que el parámetro `n` vale 3, por lo que esta llamada devolverá 3 multiplicado por el resultado de la invocación a `factorial(2)`.
- [4] Se añade el código de esta función (con `n` valiendo 2) en la pila de llamadas. Y se ejecuta el código que intenta devolver 2 multiplicado por el resultado de la `factorial(1)`.
- [5] Habrá una nueva función en la pila en la que el parámetro `n` vale 1. En este caso el código de esta invocación devuelve directamente 1.
- [6] Se retirará esa función y el flujo pasa a la función del paso 4, la cual ya puede devolver: `2*factorial(1)` ya que ahora sabemos el resultado.
- [7] La función devuelve el número 2 y el control se le queda la función del paso 3. Esta función se retira de la pila.
- [8] Ahora podremos retornar el resultado de `3*factorial(2)`, que será 6. Se devuelve el resultado a la función inicial y la actual se quita de la pila.
- [9] Ahora ya podremos devolver el resultado de `4*factorial(3)`, será 24. Esta función también se retira de la pila.

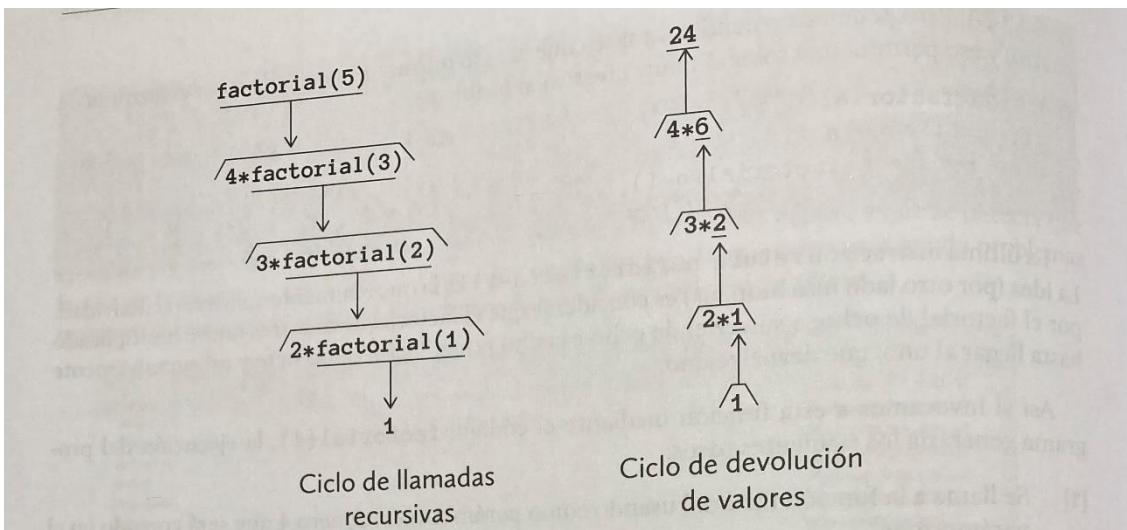


Figura 5.9: Reproducción de las llamadas recursivas en el ejemplo del factorial

ACTIVIDAD 5.3: RECURSIVIDAD

- La Práctica 5.6, permite practicar con la recursividad mediante la función de Fibonacci. Es una buena práctica para aplicar la recursividad y compararla con las soluciones iterativas.

5.4.2.1 ¿RECURSIVIDAD O ITERACIÓN?

Hay otra versión de la función factorial resuelta mediante un bucle `for` (solución iterativa) en lugar de utilizar la recursividad. Se trataría de esta:

```
function factorial(n){
    let res=1;
    while(n>1){
        res*=n;
        n--;
    }
    return res;
}
```

La cuestión es ¿cuál es mejor?

Ambas implican ejecutar sentencias de forma repetitiva hasta llegar a una determinada condición que cierra el ciclo de repeticiones. En el caso de la solución iterativa es un contador el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta conseguir una invocación a la función que devuelva un valor sencillo.

En términos de rendimiento es más costosa la recursividad, ya que implica realizar muchas llamadas a funciones en cada una de las cuales se genera una copia del código de la misma, lo que sobrecarga

No hacer la Actividad 5.3/5.6

la memoria del ordenador y tiene una forma de ejecución más lenta. Es decir, es más rápida y menos voluminosa la solución iterativa.

¿De qué sirve la recursividad, entonces? Hay una realidad: si poseemos la solución iterativa a un problema, no deberíamos utilizar la recursividad. La recursividad se debería utilizar solamente si:

- No encontramos la solución iterativa a un problema.
- El código es mucho más claro en su versión recursiva y no implica mucha diferencia a nivel de rendimiento sobre la solución iterativa.

En todo caso, al final todo depende de la habilidad del programador ante un mismo problema, hay soluciones iterativas mucho más lentas que una solución recursiva, pero esto normalmente se debe a una mala habilidad del programador.

5.4.3 FUNCIONES CALLBACK

Si hay una característica de JavaScript que distingue mucho a este lenguaje de otros, es el manejo de las llamadas funciones **callback**. Han propiciado una forma de trabajar muy especial y facilitado el entendimiento, como veremos más adelante, de que JavaScript es un lenguaje asíncrono y basado en eventos.

La idea en realidad es muy simple: si las funciones se pueden asignar a variables, también se pueden asignar a parámetros de las funciones. ¿Qué permite esta posibilidad? Conseguir que las funciones ejecuten otras funciones a través de los parámetros, es decir: las funciones pueden recibir datos y acciones a realizar. Veamos un ejemplo:

```
function escribe(dato,funcion){  
    funcion(dato);  
}  
escribe("Hola",console.log);
```

Si ejecutamos el código, veremos por consola que se escribe la palabra **Hola**. El código puede ser muy difícil de entender inicialmente pero es muy interesante. La función **escribe** recibe dos parámetros: el primero es el texto a escribir. El segundo es el nombre de la función que se encargará de realizar la escritura. Hemos pasado como segundo parámetro la expresión **console.log** por lo que la expresión **función(dato,console.log)** es totalmente equivalente (en este caso) a **console.log(dato)**.

No parece muy útil este código, pero si es fácil entender su versatilidad, ya que también podemos lanzar esta invocación:

```
escribe("Hola",console.error);
```

console.error es un método que permite escribir un error por consola. Normalmente la diferencia es que el texto sale coloreado en rojo. Pero lo interesante es que la función cambia su forma de escribir debido a la función que usamos.

Un último ejemplo sería:

```
escribe("Hola",alert);
```

Evidentemente, esta función no es muy útil, pero veremos más adelante las enormes posibilidades que dan este tipo de funciones.

Veamos este otro ejemplo:

```
function escribir(x,accion){
    console.log(accion(x));
}
function doble(y){
    return 2*y;
}
escribir(12,doble);
```

El resultado que se escribe es 24. Vayamos por partes:

- Al definir la función **doble** la damos la capacidad de devolver el parámetro que la enviamos multiplicado por dos.
- La función **escribir** recibe dos parámetros: **x** (un número) y una función. Con esos parámetros invoca a **console.log** haciendo que muestre el resultado de la función que indiquemos a la que pasaremos el parámetro **x**.
- La invocación de **escribir(12,doble)** acabará produciendo en la función **escribe** el código **console.log(doble(12))**

Es muy habitual usar funciones callback usando funciones anónimas. Si observamos el siguiente código:

```
escribir(12,function(y){
    return 2*y;
});
```

Si suponemos que la función **escribir** es la misma que en el código anterior, esta llamada a la función escribir provoca el mismo resultado: 24. El segundo parámetro no es el nombre de una función, es una función anónima cuya definición es devolver el parámetro que reciba multiplicado por dos. Ese código se asociará al parámetro **accion**.

Es más, incluso podríamos usar funciones flecha:

```
escribir(12,y=>2*y);
```

Inicialmente cuesta mucho crear funciones propias que usen funciones callback como parámetros. Pero lo útil es que hay muchos métodos de objetos básicos de JavaScript que requieren utilizar funciones callback. El uso de estos métodos facilita su aprendizaje. Por ello, en el apartado siguiente veremos algunas utilidades ya creadas que requieren usar funciones callback, y

que nos van a dar funcionalidades muy avanzadas sobre las estructuras de datos explicadas en la unidad anterior.

5.4.4 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS

5.4.4.1 ORDENACIÓN AVANZADA DE ARRAYS

Cuando explicamos el método `sort` para ordenar arrays (véase 4.4.8 "Modificar el orden de los elementos de un array", en la página 139) explicamos el problema de que, por defecto, esta función ordena el texto aplicando estrictamente el orden de la tabla Unicode. Y así, este código:

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

Produce este resultado, que en absoluto es el deseable:

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

Pero la función `sort` tiene la posibilidad de indicar un parámetro que es una función callback. Esta función debe recibir dos parámetros que sirven para explicar el criterio de ordenación. Por lo que debemos programar el código de esa función de modo que comparando, en la forma deseada, los parámetros:

- La función devuelva un número negativo si el primer parámetro es menor que el segundo
- Devuelva cero si son iguales
- Devuelva un número positivo si su segundo parámetro es mayor que el primero.

Un ejemplo de función personal para ordenar de modo que aparezcan primero los textos más cortos, sería esta:

```
function ordenPersonal(a,b){  
    return a.length-b.length;  
}
```

La función devuelve, dando por hecho que ha recibido dos parámetros de tipo string, un número negativo si el primer parámetro es más corto que el segundo, cero si los tamaños son iguales y un número positivo si el primer parámetro es más largo que el segundo. Si usamos esa función como función anónima (y en forma de flecha) que enviamos a `sort` el código sería:

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

Consigue el resultado:

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

Pero volvamos al problema de ordenar textos en la forma deseada respetando la ordenación en idioma castellano. Es decir, dejando la *eñe* entre la *ene* y la *o*, olvidar la diferencia entre mayúsculas y minúsculas y el resto de problemas que aporta el orden estricto de la tabla Unicode. Para ello, afortunadamente, disponemos del método **localeCompare** de los strings (véase (4.1.2.1 "Método localeCompare", en la página 121). Por lo cual el problema se soluciona de esta forma:

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

Ahora la ordenación es perfecta:

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

Un detalle importante es que **localeCompare** sin indicar un segundo parámetro que indica el código del país, podría ordenar mal (podría aparecer el *Ñu* antes de la *Nutria*) porque usa la configuración nacional local del usuario. Por eso, es más acertado incluir el código del idioma sobre el que deseamos ordenar:

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

La capacidad de enviar una función callback para ordenar permite realizar ordenaciones absolutamente personales en un array.

5.4.4.2 MÉTODO FOREACH

JavaScript nos ofrece una forma muy sofisticada de recorrer arrays, mapas y conjuntos. Se realiza mediante un método de los arrays que se llama **forEach** y que se incorporó al estándar **ES2015**. La sintaxis es la siguiente:

```
nombreArray.forEach(function(elemento, índice){
    instrucciones que se repiten por cada elemento del array
});
```

forEach requiere indicar una función que necesita dos parámetros: uno irá almacenando los valores de cada elemento del array y el segundo irá almacenando los índices. No es imprescindible usar ambos, el parámetro que almacena el índice es opcional.

Esa función permite establecer la acción que se realizará con cada elemento del array. Al igual que ocurría con el bucle **for...in** el método **forEach** no tiene en cuenta los elementos indefinidos. Así, el código que permite mostrar un array de notas es:

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

Es una forma de recorrer arrays muy elegante, no necesitamos obtener los valores del array mediante el índice (`notas[i]`), el parámetro `nota` se encarga de ir recogiendo directamente cada valor del array.

En el caso de los conjuntos, el funcionamiento es semejante, pero a la función callback que recibe como parámetro `forEach` no se le indica más parámetro que la variable que recogerá cada elemento del conjunto:

```
let conjunto=new Set();
conjunto.add("Paul").add("Ringo").add("George").add("John");
conjunto.forEach(function(valor){
    console.log(valor);
});
```

Escribirá los elementos del conjunto:

```
Paul
Ringo
George
John
```

Finalmente decir que, en el caso de los mapas, el método `forEach` acepta una función donde se acepta un parámetro para almacenar cada elemento del mapa y otro para almacenar las claves.

```
const provincias=new Map();
provincias.set(1,"Álava").set(28,"Madrid").set(34,"Palencia")
    .set(41,"Sevilla");
provincias.forEach(function(valor,clave){
    console.log(`Clave: ${clave}, Valor: ${valor}`);
});
```

Se escribirá por pantalla lo siguiente:

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

5.4.4.3 MÉTODO MAP

Es otro método de recorrido de arrays que permite recorrer cada elemento y, a través de una función callback que se pasa como único parámetro, establecer el cálculo que se realiza con cada elemento.

El método map no modifica el array, sino que devuelve otro con los mismos elementos y al que se le habrá aplicado la acción que se pasa como parámetro

Si, por ejemplo, deseamos doblar el valor de cada elemento de un array, el código sería:

```
const notas=[5,6,,,9,,,8,,9,,7,8];
```

```
const doble=notas.map(x=>2*x);
console.log(doble);
```

El nuevo array doble contiene los mismos valores que el array de notas, pero con los valores doblados. El código anterior escribe:

```
[ 10,
  12,
  <3 empty items>,
  18,
  <2 empty items>,
  16,
  <1 empty item>,
  18,
  <1 empty item>,
  14,
  16 ]
```

Solo los arrays disponen de método **map**.

5.4.4.4 MÉTODO REDUCE

Se trata de un método que requiere de una función callback que está pensada para recorrer cada elemento del array. Sin embargo, a diferencia de los métodos **map** y **forEach**, la idea es devolver un valor, resultado de hacer un cálculo con cada elemento del array.

El método en sí tiene un segundo parámetro (el primero es la función callback) que sirve para indicar el valor inicial que tendrá la variable que sirve para acumular el resultado final. La función callback recibe dos parámetros: el primero es el acumulador en el que se va colocando el resultado deseado y el segundo sirve para recoger el valor del elemento del array que se va recorriendo en cada momento.

Así, podemos sumar todos los elementos de un array y devolver el resultado de esta forma:

```
const array=[1,2,3,4,5];
let suma=array.reduce((acu,valor)=>acu+valor,0);
console.log(suma);
```

Hemos usado una función flecha como función callback para el primer parámetro del método **reduce**. Esta función usa los parámetros **acu** para ir almacenando el total de las sumas y **valor** que es el que va recogiendo cada valor del array. En el segundo parámetro indicamos un cero para que el parámetro **acu** empiece valiendo cero (si no usamos ese parámetro, el parámetro **acu** empieza valiendo uno).

Vamos a ver, paso a paso, como se interpreta este código

[1] **const array=[1,2,3,4,5]**
Se crea el array con los valores 1, 2, 3, 4 y 5

- [2] `let suma=array.reduce((acu,valor)=>acu+valor,0)`
 Invocamos al método `reduce`, pasamos como primer argumento la función:
`(acu,valor)=>acu+valor`
 el segundo parámetro es un cero
- [3] El mecanismo de trabajo de la función callback es este:
- [3.1] En la primera llamada el parámetro **valor** coge el valor uno (primer elemento del array). El parámetro **acu** valdrá cero (que es el valor inicial que hemos indicado). La función devuelve $0+1$: es decir, uno.
 - [3.2] Se avanza al siguiente elemento, el parámetro **acu** vale uno (resultado de la llamada anterior), **valor** vale dos (valor del segundo elemento del array). La función devuelve $1+2$, es decir: 3
 - [3.3] Se avanza al tercer elemento. El parámetro **valor** vale tres (valor del tercer elemento del array), el parámetro **acu** también vale tres (resultado de la llamada anterior). Se retorna $3+3$, es decir: 6
 - [3.4] Avanzamos al cuarto elemento con **acu** valiendo 6 y **valor** valiendo 4. El resultado de esta llamada es $6+4$, por lo tanto: 10.
 - [3.5] El quinto elemento vale 5, el acumulador vale 10. El resultado de esta llamada, que es la última, es $10+5$.
 - [3.6] El resultado de la función `reduce` será 15 ($1+2+3+4+5$), valor de la suma de todos los elementos del array.

5.4.4.5 MÉTODO FILTER

El método `reduce` es muy potente, pero a nivel práctico no se usa demasiado. El método `filter`, sin embargo, es muy utilizado. Este método utiliza una función callback que recibe un único parámetro. Gracias a ese parámetro se recoge cada valor del array. La función retorna una condición que debe cumplir cada elemento. Este método obtiene un nuevo array que tendrá como elementos, aquellos que cumplan la condición de la función callback:

```
const array=[4,9,2,6,5,7,8,1,10,3];
const arrayFiltrado=array.filter(x=>x>5)
console.log(arrayFiltrado);
```

El array llamado **arrayFiltrado** quedará de esta forma (el array original no se modifica):

```
[ 9, 6, 7, 8, 10 ]
```

Se quedan en este array los elementos que tengan un valor mayor que cinco.