CONTENIDOS

4.1 INTRODUCCIÓN A LAS FUNCIONES

JavaScript dispone de funciones y objetos para implementar el paradigma de la programación modular, pero en JavaScript las funciones adquieren una nueva dimensión gracias a su versatilidad y a su capacidad de ser usada de forma absolutamente dinámica.

Una **función** es código JavaScript que realiza una determinada tarea a partir de unos datos de entrada (***parámetros***). Dicha tarea consiste en devolver un determinado resultado a partir de los datos de entrada, o bien, realizar una acción determinada. Por ejemplo, podemos programar una función para que calcule el factorial de un número entero y devuelva dicho factorial. También, podemos crear una función más sofisticada como, por ejemplo, que se encargue de enviar los datos que la indiquemos a un servidor de base de datos.

Lo fundamental es que la misma función puede ser invocada una y otra vez. Incluso podemos usar las mismas funciones en diferentes aplicaciones. Es más legible un código que usa funciones y, además, facilita la detección de errores, ya que podemos ir mejorando y mejorando la función, para que todas las aplicaciones que la utilicen, vean reflejadas al instante esas mejoras.

Por otro lado, JavaScript es un ***lenguaje asincrono*** donde las instrucciones se ejecutan sin esperar a que la anterior termine. Eso dificulta ciertas tareas que requieren de un resultado anterior; es decir tareas que requieren un modo ***síncrono*** de trabajo. JavaScript utiliza las funciones para solucionar este, y otros problemas, gracias a que las funciones son un código que se puede asociar a cualquier variable o parámetro.

4.2 CREACIÓN DE FUNCIONES

4.2.1 ELEMENTOS DE UNA FUNCIÓN

Normalmente las funciones requieren de los siguientes elementos:

- Un **identificador** (nombre) de la función. Que cumple las mismas reglas que los identificadores de variables. Como convención formal (pero no obligatoria), las funciones se identifican con nombres en minúsculas. Hay que señalar que en JavaScript, las funciones pueden incluso ser anónimas.

- Uno o más **parámetros** que son variables locales a la función que sirven para almacenar los datos necesarios para que la función realice su labor. Puede haber funciones que no utilicen parámetros.

- Un **resultado** que es un valor (simple o complejo) que se devuelve a través de la instrucción ***retum***. Es posible que una función no devuelva un resultado, sino que, realice una determinada acción. Este tipo de funciones, que no usan la instrucción return, en otros lenguajes se las conoce como procedimientos.

    - Las **instrucciones** de la función, que son las sentencias que se ejecutan cuando se invoca a la función. Es el código en sí de la función. Este código

se ejecuta cuando invocamos a la función desde cualquier parte del código de la aplicación.

4.2.2 DECLARAR E INVOCAR FUNCIONES

Las funciones deben de ser declaradas antes de que se puedan usar. La sintaxis habitual para declarar una función de **forma clásica o notación declarativa**, es:

function *nombre* (*[lista_parametros]){*
instrucciones
}
Donde [lista_parametros] es opcional

Ejemplo:

function saludo(){
console.log("Hola");
}
Esta función no tiene ni parámetros ni retorna ningún valor. Pero sí realiza una acción (aunque muy sencilla), escribe **Hola** por consola.

La forma de invocar a una función es indicar su nombre y después, entre paréntesis, indicar los parámetros de la función. Aunque la función, no requiera de parámetros, debe de utilizar los paréntesis.

saludo(); // invoca a la función y escribe Hola

Ejemplo pasando un parámetro:

function saludo(mensaje){
console.log(mensaje);
}
Se invocaría por ejemplo:
saludo("Hola, ¿ Qué tal ?"); //Escribe Hola, ¿ Qué tal ?" por consola

La siguiente función multiplica por 3 el número introducido en el parámetro n y devuelve el resultado mediante la instrucción ***return***
function triple(n){
return 3*n;
}

Un ejemplo de función un poco más compleja:

```
function pares(array){
  let nPares=0;
  if(array instanceof Array){
    for(n of array){
       if(n%2===0){
         nPares++;
        }
     }
  }
   return nPares;
}
```

Un posible uso sería: ***console.log(pares([1,2,3,4,5,6,7,8,9]));*** //Escribe 4

El resultado indica que hay cuatro números pares en el array que se pasa a la función. Hay que tener en cuenta que las funciones se pueden invocar en cualquier parte.

4.2.3 ASIGNAR FUNCIONES A VARIABLES

Hay un tipo especial de función en JavaScript, se llama función anónima y nos lleva de pleno a la forma que tiene JavaScript de entender las funciones. Realmente las funciones son, simplemente, un código que se puede invocar una y otra vez. El acceso a ese código se puede hacer, como hemos visto, con el nombre de la función. Pero en JavaScript no es la única manera de acceder a dicho código. Cualquier mecanismo de javaScript que permita acceder a ese código es válido. Por ello, este código es correcto:

```
const triple=function(x){
return 3*x;
}
```

Si nos concentramos solo en el código tras la palabra function veremos un código que, recibiendo un parámetro que hemos llamado x, lo devuelve multiplicado por tres. La novedad es que esa función no tiene nombre. La **función es anónima**, pero el código de la misma es accesible porque se lo estamos asignando a la variable ***triple***. Aunque triple es una variable, no es una función, puede acceder al código de la función de la misma forma que si hubiera puesto ese nombre a la función ya que la variable es una referencia a la función:

```
console.log(triple(3)); //Escribe 9
```

El hecho de declarar trip como constante (const) tiene sentido si esa variable siempre se asocia con la función a la que se asigna en la declaración, si asignamos otra función ocurriría un error (porque la referencia cambia). Es posible incluso que dos variables hagan referencia a la misma función.

4.2.4 FUNCIONES FLECHA

Hay otra manera de declarar funciones que se ha convertido en muy popular debido a su facilidad de escritura. Solo sirve para funciones anónimas y consiste en que no aparece la palabra function y en que una flecha separa los parámetros del cuerpo de la función. Ejemplo:

```
const triple = x=>3*x;
console.log(triple(20));
```

La definición de la función anónima es la expresión: ***x=>3*x***

El símbolo de la flecha separa los argumentos del cuerpo de la función, en el que, además se sobrentiende la palabra return.

Evidentemente, es una notación para escribir más rápido. Si hay más de un parámetro, se deben colocar entre paréntesis. Ejemplo:

```
const media=(x,y)=>(x+y)/2;
console.log(media(10,20));
```
//Escribe 15

Esta función es un poco más compleja y requiere que los dos parámetros que utiliza la función estén entre paréntesis, si no la expresión fallaría.

Por otro lado, si el cuerpo de la función es más complejo, requiere ser incluido entre llaves:

```
const sumatorio = (n)=>{
  let acu=0;
  for(let i=n;i>0;i--){
    acu+=i;
  }
  return acu;
}
console.log(sumatorio(3));
```
//Escribe 6, resultado de 3+2+1

También son necesarias las llaves cuando no hay return:

```
const saludo = mensaje=>{
   console.log(mensaje);
 }
saludo("Hola");
```

Si en la función no hay parámetros, hay que colocar paréntesis vacíos en la posición que ocu parían los parámetros:

```
const hola = ()=>{
console.log("Hola");
}
hola();
```
//Escribe Hola

Pero hay que tener en cuenta que, ante funciones complejas, las ventajas de las funciones flecha se diluyen. Lo habitual es que se utilicen funciones flecha para definir funciones sencillas.

4.3 DETALLES SOBRE VARIABLES Y PARÁMETROS

4.3.1 ÁMBITO DE LAS VARIABLES

Las variables tienen una duración en el código dependiendo de cómo se han declarado. ***Las variables definidas en una función tanto con const, como con let, como con var no se pueden usar fuera de la función en la que se declaran***:

```
function f(){
  const a=9;
  let b=9;
  var c=9;
  console.log("soy la función f");
}
f();
console.log(a);    //error
console.log(b);   //error
console.log(c);   //error
```

Veamos este otro ejemplo:

```
function f(){
   if(true){
     const a=9;
     let b=9;
     var c=9;
   }
console.log(a);     //error
console.log(b);    //error
console.log(c);   // es correcta: salida 9
}
f();
```

Podemos utilizar la variable ***c*** fuera de la estructura ***if*** aunque se declaró en ella. Sin embargo, las dos líneas anteriores provocan un error ya que ***las variables definidas con const y let no pueden usarse fuera del bloque en el que fueron definidas***, en este caso no se pueden usar fuera del if.

En cuanto a los parámetros, tampoco pueden usarse fuera de la función en la que se definen:

```
function g(x){
  x=19;
}
g(8);
console.log(x);    //Error, x no se puede usar fuera de la función
```

Se pueden usar solamente en la función. Es decir, su ámbito es el mismo que el de las variables declaradas mediante la palabra var.

4.3.2 PASO POR VALOR Y PASO POR REFERENCIA

Si tenemos por ejemplo el siguiente código:

```
var x=19;
function f(x){
  x++;
}
f(x);
console.log(x);
```

Este código es un poco enrevesado, se ha forzado a que sea así para explicar un detalle muy importante. El código parece que anima a pensar que cuando se escribe en pantalla el valor de x aparece el número 20. Sin embargo, aparece el 19.

Si pasamos una variable a una función como parámetro, se recoge una copia de su valor. La variable original no se modifica. Por eso, lo lógico es que los parámetros no se llamen igual que las variables que se usan para pasar su valor.

Para entender las razones de este efecto veamos cómo se interpreta realmente este código:

- Inicialmente se crea una variable llamada x que como se declara con la palabra var y está declarada fuera de toda función, su ámbito es todo el código.
- La función **f** se declara como una función que tiene un parámetro que también se llama x. Pero esta ***x*** no es la x del punto anterior. Son variables diferentes, aunque tengan el mismo nombre. De hecho, por culpa de esa coincidencia, la función no puede acceder a la variable x declarada en la primera línea.
- Tras la llave de cierre de la función se invoca a la función **f** pasando el valor de la x que se declaró fuera de la función. Es decir, se pasa el número 19 a la función.
- Ese valor se copia al parámetro x de la función.
- Después se ejecuta el código de la función que modifica el valor del parámetro x ya que incrementa su valor. Valdrá 20. Pero la variable x original no se ha modificado.
- Cuando termina la función, el parámetro x se elimina. Perderemos su valor que era 20.
- Cuando ***console.log escribe el valor de x***, la variable a la que se refiere es la global, la que se declaró en la primera línea que sigue valiendo 19.

Si hacemos algo como lo siguiente, tampoco se vería actualizado el valor de x. Habría que retornar el valor de y para asignarlo de nuevo a x si queremos que esta se actualice

```
var x=19;
function f(y){
  y=20;
}
f(x);
consolé.log(x);
```

Podemos caer en la tentación de escribir algo así:

```
var x=19;
function f(){
  x=20;
}
f();
consolé.log(x);
```

Ahora sí aparece 20. Se ha modificado la variable x original dentro de la función. No hay ambigüedad porque la función no declara ningún parámetro o variable interna con ese mismo nombre. No obstante, no es recomendable usar variables globales dentro de las funciones. La modularidad que aportan las funciones se pierde si hacemos uso de esta técnica, ya que la función no sería transportable a otro archivo. Las funciones deben de crearse de la forma más independiente posible respecto al código que las rodea.

Observemos ahora el siguiente código:

```
var array=[1,2,3,4,5];
function g(a){
  a[0]=9;
}
g(array);
console.log(array[0]);
```

Tras lo explicando antes, ahora veremos, seguramente, con sorpresa que se escribe un 9 y no un 1. Es decir, la función ha modificado el valor del array original. Nuevamente, para saber por qué ocurre este hecho, veremos el procesamiento de este código paso a paso:

- En la primera línea se crea una variable global llamada array que es una referencia a un array que almacena los valores 1,2,3,4 y 5.
- Se declara una función llamada g que tiene un parámetro llamado a. El cuerpo de la función sirve para modificar el primer elemento de a, ya que da por hecho que ese parámetro es un array, y le otorga el valor 9.
- Se invoca la función g pasando el array original. El parámetro a recogerá una referencia a ese array. Esta vez no se recibe una copia, sino una referencia al array original. Los arrays no se copian cuando se asignan. Es decir, array y a son una referencia al mismo array.

- Dentro de la función se modifica el primer elemento de a para que valga 9. Eso es lo mismo que modificar el primer elemento de la variable array.
- La función termina, el parámetro a ya no estará disponible.
- Se escribe el primer elemento de la variable array y comprobaremos que dentro de la función se ha modificado realmente su valor.

Conclusión: Moraleja final: los tipos básicos (booleanos, números y strings) se pasan por valor, se envía una copia de su valor a los parámetros de las funciones, los tipos complejos: arrays, conjuntos, mapas,.... en definitiva cualquier objeto, pasan una referencia al objeto original; si en la función se modifica el parámetro relacionado, se modificará realmente el array original.
En definitiva, los datos simples (strings, números y valores booleanos) se pasan por valor, los objetos se pasan por referencia.

4.3.3 ARGUMENTOS CON VALORES POR DEFECTO

En JavaScript, los parámetros pueden tener un valor predeterminado. Eso convierte a dicho parámetro en opcional: es decir, podremos enviar o no valores para ese parámetro.

Ejemplo:

```
function saludo(texto="Hola"){
console.1og(texto);
}
saludo();
saludo("Buenos días");
```

Por pantalla aparece:

Hola
Buenos días

La primer invocación a la función es: saludo() lo que hace que el único parámetro de la función (texto) tome el valor “Hola” que es lo que aparece por pantalla. Las funciones pueden utilizar tantos parámetros por defecto como se desee.

4.3.4 NÚMERO VARIABLE DE PARÁMETROS.

Veamos este ejemplo:

```
function media(x,y){
  return (x+y)/2;
}
console.log(media(10,20));
console.log(media(10,20,30));
```

El resultado de este código es: las dos salidas escriben 15

La primera invocación (***media(10,20)***) hace que la función use el valor 10 para el parámetro x y 20 para el parámetro y. Pero en la segunda se pasa un tercer número. No hay error, pero ese tercer número simplemente es ignorado.

Lo interesante es que podemos pasar tantos valores como nos apetezca, lo que permite crear funciones con un número variable de parámetros. El problema es cómo recoger esos valores. Para eso nos sirve el operador de propagación "…". Este operador, utilizado en los parámetros de una función, permite almacenar una serie indefinida de parámetros en un array. Veamos un ejemplo de cómo funciona:

```
function f(x,y,...mas){
  console.log('x=${x} y=${y} mas=${mas}');
}
f(10,20);
f(10,20,30) ;
f(10,20,30,40);
```

Resultado:

```
x=10 y=20      mas=
x=10 y=20      mas=30
x=10 y=20      mas=30,40
```

En la primera línea se resuelve la invocación f(10,20) el parámetro mas se queda sin resolver (será vacío). En la segunda línea vemos el valor 30 asociado al parámetro mas. En la tercera línea observamos que son ambos números 30 y 40 los que se asocian este parámetro.

En resumen, ***lo que hace el operador de propagación en este contexto es convertir una lista de parámetros en un array***. Esto permite revisar nuestra función para el cálculo de la media, de modo que pueda utilizar cualquier número de argumentos. Deberemos resolverla pensando que lo que el usuario envía es un array de números (aunque no lo sea). El cálculo de la media sumará los elementos del array y los dividirá entre el número de elementos del array para calcular la media:

```
function media(... números){
  let acu=0;
  for(let n of números){
    acu+=n;
  }
return acu / numeros.length;
}
console.log(media(10,20));
console.log(media(10,20,30));
console.log(media(10,20,30,40));
console.log(media(10,20,30,40,50));
```

Como vemos, las invocaciones a la función usan los números que queramos. Si la función está bien resuelta, la media siempre es correcta.

4.4 USO AVANZADO DE FUNCIONES

4.4.1 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 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. 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).

Otro ejemplo:

```
function escribir(x,accion){
   console.log(accion(x));
}

function doble(y){
   return 2*y;
}
escribir(12,doble);  // a la salida escibe 24
```

- Al definir la **función doble** la damos la capacidad de devolver el parámetro de entrada 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;
 });
```

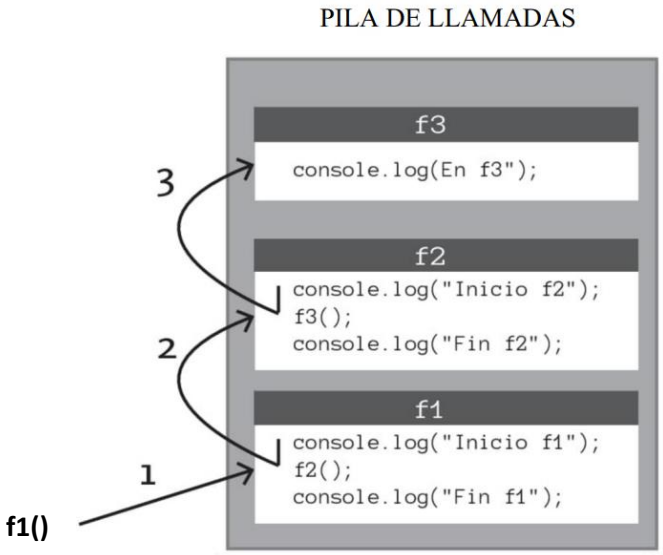
Si suponemos que la función escribir es la misma que en el código anterior, ésta llamada a la función escribir provoca el mismo resultado: 24. El ***segundo parámetro*** no es el nombre de una función, ***es una función anónima*** cuya definición es devolver el parámetro que reciba multiplicado por dos. Ese código se asociará al parámetro acción. 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 4.4.3 veremos algunas utilidades ya creadas que requieren usar funciones callback, y que nos van a dar funcionalidades muy avanzadas sobre las estructuras de datos Array, Set y Map.

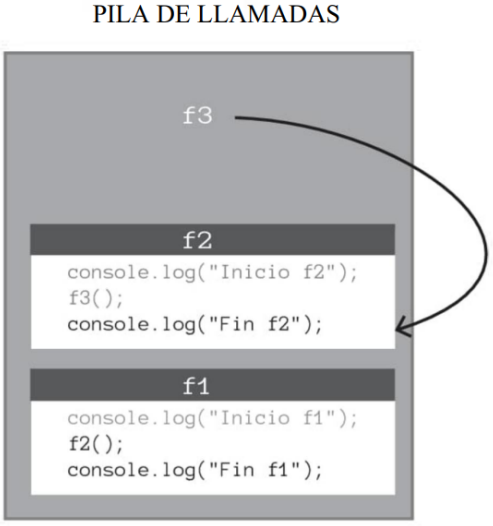
4.4.2 LA PILA DE FUNCIONES

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

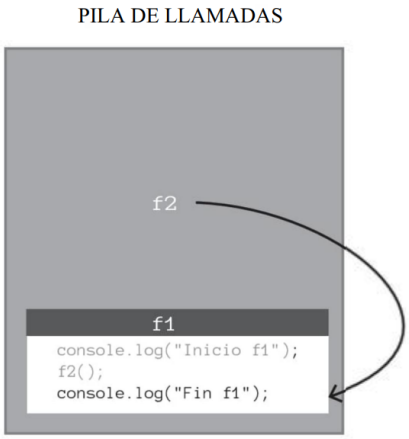
```
function f1(){
  console.log("Inicio fl");
  f2();
 console.log("Fin fl");
}

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

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

f1();
// el resultado tras invocar a f1() será:
Inicio fl
Inicio f2
En f3
Fin f2
Fin fl
```



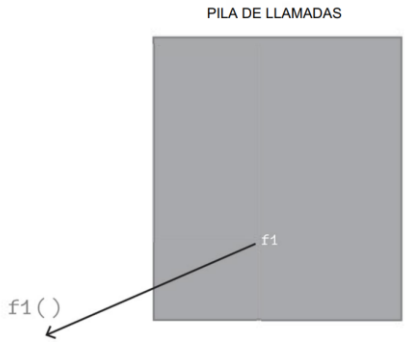
Cuando la función f3 finaliza, devuelve el control a f2 y f3 se retira de la pila



f2 recupera el control y escribe el mensaje Fin f2.
Como f2 ha finalizado, devuelve el control a f1 y se retira de la pila.



f1 recupera el control y escribe Fin f2.
f1 se retira de la pila, el control se devuelve a la función principal.



Ante una mala gestión de llamadas a las funciones se puede llegar a provocar lo que se conoce como ***desbordamiento de pila***.

4.4.3 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS

ORDENACIÓN AVANZADA DE ARRAYS

El método sort() para ordenar array, por defecto, 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 el resultado [‘Nutria’, ‘boa’, ‘marsopa’, ‘oso’, ‘Águila’, ‘Ñu’] que no es el deseado.

Pero la función o método sort tiene la posibilidad de ***indicar un parámetro que es una función callback***. Esa 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:

```
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); // salida [‘Ñu', 'boa', 'oso1, 'Águila', 'Nutria1, '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*. 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); // salida: ['Águila', 'boa', 'marsopa','Nutria', 'Ñu', 'oso']
```

Al método localeCompare se le puede pasar un segundo parámetro para indicar el código del idioma sobre el que queremos ordenar localeCompare(b,"es").

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

MÉTODO FOREACH

Este métodos nos ofrece una forma muy sofisticada de recorrer arrays, mapas y conjuntos. Su sintaxis es:

```
nombreArray.forEach(fucntion(elemento,índice){
        instrucciones que se repiten para 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}`);
});
```

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("Paúl").add("Ringo").add("George").add("John");
conjunto.forEach(function(valor){
  console.log(valor);  // escribe cada uno de los elementos del conjunto
});
```

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

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);   //devuelve un array nuevo
console.log(doble);
```

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 yforEach, 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).

MÉTODO FILTER

Este método utiliza una función callback que recibe un único parámetro. Gracias a ese parámetro se recoge cada valor del array. La fúnción 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. Ejemplo:

```
const array=[4,9,2,6,5,7,8,1,10,3];
const arrayFiltrado=array.filter(x=>x>5)   // devuelve un nuevo array
console.log(arrayFilirado);  // los elementos del arrayFiltrado son aquellos mayores de 5
```