UD5. FUNCIONES

1. INTRODUCCIÓN A LAS FUNCIONES

En la programación clásica, las funciones fueron la base de la **programación modular**. La idea de programar aplicaciones de forma modular se basa en el paradigma "divide y vencerás" que, aplicado al desarrollo de aplicaciones, implica en no abordar el problema complejo, sino en descomponerlo en problemas más pequeños, más fáciles de solucionar uno a uno.

La programación modular permite la posibilidad de que un programa se divida en un conjunto de módulos cada uno de los cuales se programaba de manera independiente. Cada módulo se encarga de una determinada tarea o función de modo que, a partir de una serie de datos de entrada (**parámetros**)

produce unos resultados o bien, realiza una acción determinada. La aplicación completa se convierte en una serie de módulos que se interconectan consiguiendo solventar el problema completo.

En el caso de JavaScript, se dispone de funciones y objetos para implementar este paradigma.

Una 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 podremos ir mejorando la función para que todas las aplicaciones que la utilicen, vean reflejadas al instante estas mejoras.

Por otro lado, JavaScript es un lenguaje asíncrono 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.

En realidad, ya hemos utilizado funciones en las unidades precedentes. Por ejemplo, **parseInt** es una función con la que hemos convertido textos en números enteros.

2. CREACIÓN DE FUNCIONES

2.1. ELEMENTOS DE UNA FUNCIÓN

Normalmente las funciones requieren de los siguientes elementos:

- Un **identificador** (nombre). Que cumple las mismas reglas que los identificadores variables. Como convención formal, pero no obligatoria, las funciones se identifican con nombres en minúsculas. Hay que señalar que en JavaScripts, las funciones pueden incluso ser anónimas.
- Uno o más **parámetros** 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 **return**. 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 conocen como **procedimientos**.
- Las **instrucciones** de la función, que son las sentencias que se ejecutan cuando se invoca la función. Es el código en sí de la función. Este código se ejecuta cuando invocamos a la función desde cualquier parte del código.

2.2. DECLARAR E INVOCAR FUNCIONES

Las funciones deben ser declaradas antes de que se puedan usar. Aunque hay más maneras, como veremos más adelante, pero esta es la más clásica y se la conoce en JavaScript como **notación declarativa**:

```
function nombre([listaParámetros]){
        ...cuerpo de la función...
```

```
}
```
Ejemplo de declaración usando la notación declarativa:

```
function saludo(){
        console.log("Hola");
}
```
Esta función no tiene parámetro ni retorna ningún valor. Pero sí realiza una acción, escribe Hola por la consola. Para utilizar esta función debemos invocarla de esta forma:

```
saludo(); //Escribe Hola
```
La forma de invocar una función es indicar su nombre y después, entre paréntesis, indicar los parámetros de la función. Aunque la función, como ocurre con el anterior ejemplo, no requiera de parámetros, debe utilizar los paréntesis.

Podemos mejorar esta función si permitimos que use parámetros. En lugar de escribir ***Hola***, vamos a conseguir que escriba el texto que la enviemos. La declaración de la función será:

```
function saludo(mensaje){
        console.log(mensaje);
}
```
Ahora podemos invocarla:

```
saludo("Hasta la vista"); //Escribe Hasta la vista por consola
```
Las funciones pueden devolver un resultado.

```
function triple(n){
        return 3*n;
}
```
Esta función recibirá un número al que llamamos n, y retorna el resultado de multiplicar por 3 ese número. Un ejemplo de uso de esta función sería:

```
let x=6, y=4, z= "Hola";
console.log(triple(9)); //Escribe 27
console.log(triple(x)); //Escribe 18
console.log(triple(x+y)); //Escribe 30
console.log(triple(x)+triple(y)); //Escribe 30
console.log(triple(triple(9)); //Escribe 81
console.log(triple(z)); //Escribe NaN
```
Podemos observar cómo al invocar la función, se admite cualquier expresión para indicar sus parámetros.

Esta otra función sirve para contar a cantidad de números pares que hay en un array.

```
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
```

Hay que tener en cuenta que las funciones se pueden invocar en cualquier parte, por ejemplo, al indicar los valores de un array.

```
let array=[triple(1),triple(2),triple(3)];
console.log(array);
```

El resultado será:

```
▸ (3) [3, 6, 9]
```

Las expresiones, siempre que sean válidas, aunque complejas, pueden simplificar mucho nuestro trabajo.

```
console.log(pares([triple(1),triple(2),triple(3)])); //Escribe 1
```

Solo el triple de 2 es par, por eso el resultado de esa expresión es el número 1.

- ❑ **ACTIVIDAD 1 - PRIMERA FUNCIÓN**

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 este código se puede hacer, 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 ejemplo:

```
const trip=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 3. 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 **trip**. Aunque **trip** 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.

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

Es decir, la variable es una referencia a la función. En JavaScript tenemos, como en muchos lenguajes, referencias a arrays y a otros objetos, y también, algo que ocurre en menos lenguajes, referencias a funciones.

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 2 variables hagan referencia a la misma función. Si añadimos este código al anterior:

```
let x=trip;
console.log(x(8)); //Escribe 24
```

Ambas variables (x y trip) utilizan la misma función.

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**

Que resulta equivalente a :

```
function(x){
        return 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 1 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 2 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;
}
```

También son necesarias las llaves cuando no hay un 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 ocuparían 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:

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

No parece un código que ahorre mucho tiempo respecto a la forma clásica de escribir funciones anónimas:

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

De ahí que lo habitual es que los programadores solo usen funciones flecha para definir funciones sencillas.

- ❑ ACTIVIDAD 1bis - CREAR LA FUNCIÓN DE LA ACTIVIDAD 1 COMO FLECHA

3. DETALLES SOBRE VARIABLES Y PARÁMETROS

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** o **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 otro código:

```
function f(){
        if(true){
                const a=9;
                let b=9;
                var c=9;
        }
        console.log(a);   //error
        console.log(b);  //error
        console.log(c);  //¡¡Esta sería correcta!!
        }
f();
```

Podemos utilizar la variable **c** fuera de la estructura **if**, aunque se declaró en ella. Sin embargo, las 2 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 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**.

3.2. PASO POR VALOR Y PASO POR REFERENCIA

```
var x=19;
```

```
function f(x){
        x++;
}
f(x);
console.log(x); //Escribe 19
```

Cuando 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. Sería más lógico el siguiente código:

```
var x=19;
function f(){
x=20;
}
f();
console.log(x); //Escribe 20
```

En este caso, no hay ambigüedad porque la función no declara ningún parámetro o variable interna y la **x** que modifica es una variable global.

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 crearse de la forma más independiente posible respecto al código que las rodea.

Veamos otro código:

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

Tras lo visto anteriormente, nos puede sorprender que se escriba un 9 y no un 1. Es decir, la función, ha modificado el valor del array original. Vamos a revisar las instrucciones paso a paso:

- 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 la función **g** con 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.

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 se pasan por valor y los objetos se pasan por referencia.

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.log(texto);
}
saludo();
saludo("Buenos días");
```

```
Hola
Buenos días
›
```

Las funciones pueden utilizar tantos parámetros por defecto como se desee.

- ❑ ACTIVIDADES 2 y 3 TABLA y NÚMEROS PRIMOS

3.4. NÚMERO VARIABLE DE PARÁMETROS

Vamos a ver un ejemplo:

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

El resultado es:

```
15
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 queramos, lo que permite crear funciones con un número variable de parámetros. El problema es cómo recoger esos valores. Para eso resulta útil

el operador de propagación (...). Este operador, utilizado en los parámetros de una función, permite almacenar una serie indefinida de parámetros en un array. Veamos 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);
```

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

En resumen, lo que hace el operador de propagación en este contexto es convertir la 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.

```
function media(...numeros){
        let acu=0;
        for(let n of numeros){
                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));
```

```
15
20
25
30
>
```

Lo que no podemos hacer es enviar un array a esta función, porque al escribir :

```
console.log(media([10,20,30,40,50])); //Escribe NaN
```

Habría que hacer otra función si queremos pasar los números de esa forma o bien modificar el código de la función para que acepte esta posibilidad.

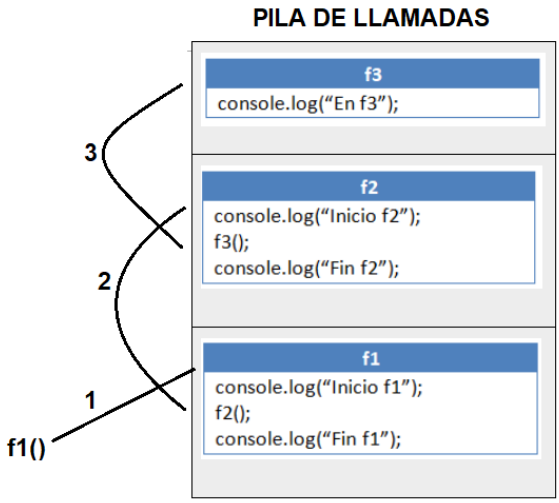
4. USO AVANZADO DE FUNCIONES

4.1. LA PILA DE FUNCIONES

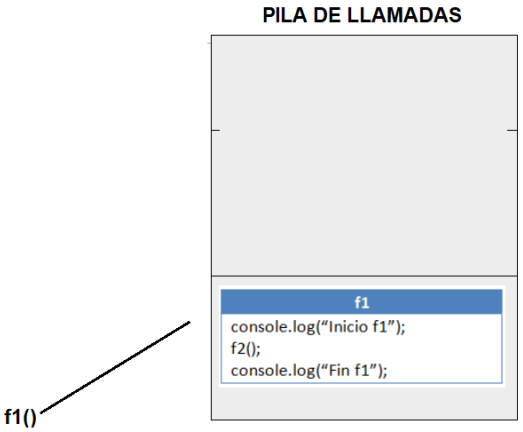
Cuando se invoca a una función en una expresión, esta debe esperar a que la función finalice para poder completar la expresión. Ejemplo:

```
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();
```

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

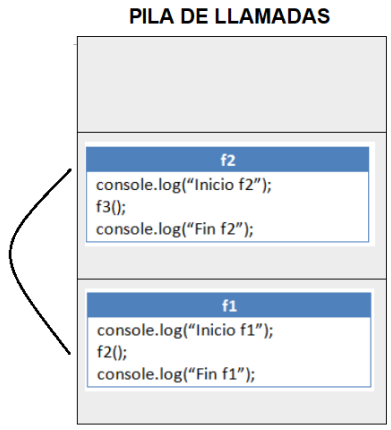


La primera función f1() no finaliza hasta que las otras han finalizado. Es decir, las funciones utilizan lo que se conoce como pila de llamadas que permite que, el intérprete JavaScript sepa qué funciones se deben de resolver antes. Las funciones se apilan en la pila de llamadas. La última función invocada 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.



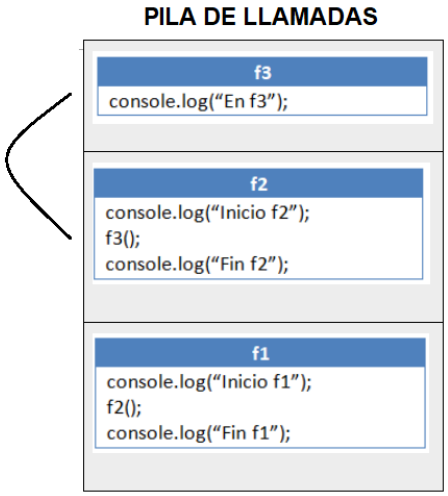
[1] La función f1() se coloca en la pila de llamadas.

[2] Se interpreta el código de f1. Se ejecuta la escritura del texto "Inicio f1"



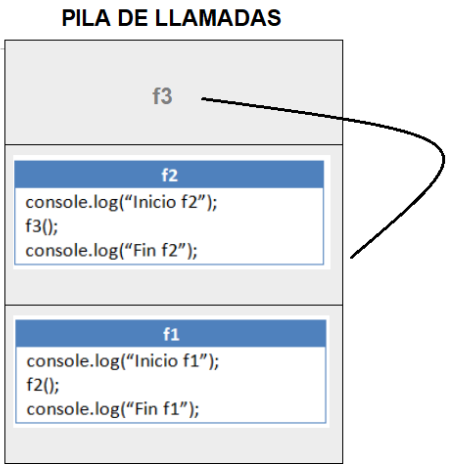
[3] Se invoca a f2(). Este código se pone en la cima de la pila. Se queda f1 a a la espera de que se resuelva f2.

[4] Se escribe "Inicio f2".



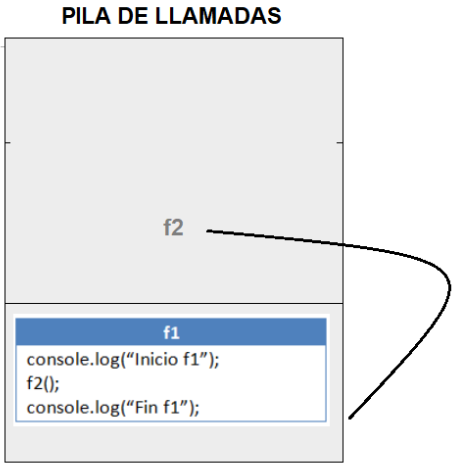
[5] Se invoca a f3(). Su código pasa a ocupar la cima de la pila. F2 se queda esperando que se resuelva el código de f3.

[6] Se escribe En f3.

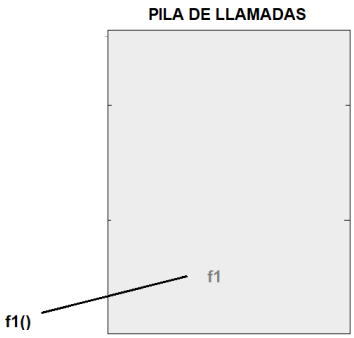


[7] La función f3 finaliza, devuelve el control a f2 y f3 se retira de la pila.

[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.



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

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 llamarían 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 este tope el código anterior provocaría efectos más devastadores.

```
Despedida
Saludo
Despedida
Saludo
Despedida
Saludo
Despedida
▸Uncaught RangeError: Maximum call stack size exceeded
    at saludo (html4.htm:13:3)
    at despedida (html4.htm:23:3)
    at saludo (html4.htm:15:3)
    at despedida (html4.htm:23:3)
    at saludo (html4.htm:15:3)
    at despedida (html4.htm:23:3)
    at saludo (html4.htm:15:3)
    at despedida (html4.htm:23:3)
    at saludo (html4.htm:15:3)
    at despedida (html4.htm:23:3)
```

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 en 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 llegar a esta última llamada ya que es la que cierra el bucle de llamadas y tras ella se irán resolviendo las anteriores hasta liberara la pila y conseguir el resultado final.

La recursividad se entiende mejor con ejemplos. El ejemplo clásico es el cálculo del factorial de un número. Así, el factorial de 5 (5!) es el resultado de 5.4.3.2.1. Pero podemos entender también que 5! Es lo mismo que 5*4! (5 por el factorial de 4) y eso permite la siguiente solución:

```
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.

factorial(5)	120
5*factorial(4)	5*24
4*factorial(3)	4*6
3*factorial(2)	3*2
2*factorial(1)	2*1
1	1
Ciclo de llamadas recursivas	Ciclo de devolución de valores

- ❑ **ACTIVIDAD 6 Y 8 – RECURSIVIDAD y TRIBONACCI**

4.2.1. ¿RECURSIVIDAD O ITERACIÓN?

Hay otra versión de la función factorial resuelta mediante el bucle **while** (solución iterativa) en lugar de utilizar la recursividad. Sería esta:

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

Entonces, ¿cuál es mejor?

Ambas implican ejecutar sentencias de forma repetitiva hasta llegar a una determinada condición que cierra el ciclo de operaciones. 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 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?. La recursividad se debería utilizar solamente si:

- No encontramos solución iterativa al 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, pudiendo generarse soluciones iterativas mucho más lentas que una solución recursiva.

4.3. FUNCIONES CALLBACK

Si hay una característica de JavaScript que distingue 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 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, función){
        función(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 2 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 de 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.

Otro ejemplo sería:

```
escribe(“Hola”, alert);
```

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

Veamos otro ejemplo:

```
function escribir(x, accion){
        console.log(accion(x));
}
function doble(y){
        return 2*y;
}
escribir (12, doble); //Escribe 24
```

- Al definir la función **doble** le damos la capacidad de devolver el parámetro que le enviamos multiplicado por 2.
- La función **escribir** recibe 2 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 le pasaremos el parámetro **x**.
- La invocación escribir(12,doble) acabará produciendo en la función escribir 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 2. 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 siguiente veremos algunas utilidades ya creadas que requieren usar funciones callback, y que nos va a dar funcionalidades muy avanzadas sobre las estructuras de datos explicadas en la unidad anterior.

- ❑ ACTIVIDAD 5 – MAP

4.4. USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS

4.4.1. ORDENACIÓN AVANZADA DE ARRAYS

Cuando explicamos el método **sort** para ordenar arrays, 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);
```

```
▸ (6) ['Nutria', 'boa', 'marsopa', 'oso', 'Águila', 'Ñu']
```

Pero la función **sort** tiene la posibilidad de indicar un parámetro que es una función callback. Esa función debe recibir 2 parámetros que sirven para explicar el criterio de ordenación. Po 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
- Devuelve 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 2 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 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:

```
(6) ['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 **ñ** entre la **n** 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, disponemos del método **localCompare** 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);
```

Ahora la ordenación es perfecta:

```
(6) ['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 que 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.

- ❑ **ACTIVIDAD 7 y 8 PALÍNDROMOS Y ANAGRAMAS**

https://www.w3schools.com/js/js_callback.asp

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 2 parámetros: uno irá almacenando los valores de cada elemento del array 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(34,"Alicante").set(28,"Valencia").set(41,"Castellón");
provincias.forEach(function(clave,valor){
        console.log(`Clave: ${clave}, Valor: ${valor}`)
```

```
});
```

```
Clave: Álava, Valor: 1
Clave: Alicante, Valor: 34
Clave: Valencia, Valor: 28
Clave: Castellón, Valor: 41
```

- ❑ ACTIVIDAD 4 COMENTAR CÓDIGO

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(notas);
```

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

```
▸ (14) [5, 6, empty × 3, 9, empty × 2, 8, empty, 9, empty, 7, 8]
```

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

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 2 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 la 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 0 para que el parámetro **acu** empiece valiendo cero (si no usamos ese parámetro, el parámetro **acu** empieza valiendo 1).

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

[1] const array=[1,2,3,4,5]

Se crea un array con los valores 1,2,3,4 y 5.

[2] let suma=array.reduce((acu,valor)=>acu+valor,0);

Invocamos el método reduce, pasamos como primer argumento la función (acu, valor)=>acu+valor

El segundo parámetro es un 0

[3] El mecanismo de trabajo de la función callback es este:

[3.1] En la primera llamada el parámetro valor coge el valor 1 (primer elemento del array). El parámetro acu valdrá 0 (que es el valor inicial indicado). La función devuelve 0+1: es decir,1.

[3.2] Se avanza al siguiente elemento, el parámetro acu vale 1 (resultado de la llamada anterior), valor vale 2 (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 3 (valor del tercer elemento del array), el parámetro acu vale 3 (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, esto es: 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+3+4+5), valor de la suma de todos los elementos del array.

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 modificará):

```
(5) [9, 6, 7, 8, 10]
```