

FUNCIONES

OBJETIVOS

- Valorar la importancia de las funciones en el desarrollo de aplicaciones
- Identificar los elementos y utilidad de los elementos de una función
- Reconocer la sintaxis de creación de funciones
- Crear funciones usando notación declarativa, notación anónima y notación de flecha
- Aplicar el uso de funciones en la creación de aplicaciones web
- Asimilar el funcionamiento de la pila de funciones
- Interpretar el aporte de la recursividad en la resolución de problemas mediante funciones
- Manejar funciones callback tanto para resolver problemas propios como para utilizar métodos de la librería estándar de JavaScript

CONTENIDOS

- 5.1 INTRODUCCIÓN A LAS FUNCIONES**
- 5.2 CREACIÓN DE FUNCIONES**
 - 5.2.1 ELEMENTOS DE UNA FUNCIÓN
 - 5.2.2 DECLARAR E INVOCAR FUNCIONES
 - 5.2.3 ASIGNAR FUNCIONES A VARIABLES
 - 5.2.4 FUNCIONES FLECHA
- 5.3 DETALLES SOBRE VARIABLES Y PARÁMETROS**
 - 5.3.1 ÁMBITO DE LAS VARIABLES
 - 5.3.2 PASO POR VALOR Y PASO POR REFERENCIA
 - 5.3.3 ARGUMENTOS CON VALORES POR DEFECTO
 - 5.3.4 NÚMERO VARIABLE DE PARÁMETROS
- 5.4 USO AVANZADO DE FUNCIONES**
 - 5.4.1 LA PILA DE FUNCIONES
 - 5.4.2 RECURSIVIDAD
 - 5.4.3 FUNCIONES CALLBACK
 - 5.4.4 USO DE MÉTODOS AVANZADOS PARA MANIPULAR ESTRUCTURAS DE DATOS
- 5.5 PRACTICAS RESUELTA**
- 5.6 PRÁCTICAS RECOMENDADAS**
- 5.7 RESUMEN DE LA UNIDAD**
- 5.8 TEST DE REPASO**

5.1 INTRODUCCIÓN A LAS FUNCIONES

En la programación clásica, las funciones fueron la base de la llamada **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 completo, 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 produce unos resultados. La aplicación completa se convierte en una serie de módulos que se interconectan consiguiendo solventar el problema completo.

Pues bien, en el caso de JavaScript se dispone de funciones y objetos para implementar este paradigma. Indudablemente las funciones son un mecanismo para programar de forma 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 una, 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 **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, `parselInt` es una función que hemos utilizado para poder convertir textos en números enteros.

5.2 CREACIÓN DE FUNCIONES

5.2.1 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, como veremos más adelante, 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 **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 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.

5.2.2 DECLARAR E INVOCAR FUNCIONES

Las funciones deben de ser declaradas antes de que se puedan usar. La sintaxis habitual para declarar una función, es:

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

Hay más maneras (como veremos más adelante) de declarar funciones, pero esta es la más clásica (es similar en casi todos los lenguajes de programación) y se la conoce en JavaScript como **notación declarativa**.

Ejemplo de declaración usando la notación declarativa:

```
function saludo(){
    consolé.log("Hola");
}
```

Esta función no tiene ni parámetros ni retorna ningún valor. Pero sí realiza una acción (aunque muy sencilla), escribe **Hola** por consola. Para utilizar esta función debemos invocarla de esta forma:

```
saludo(); //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, como ocurre con el anterior ejemplo, no requiera de parámetros, debe de 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 envíemos. La declaración de la función será:

```
function saludo(mensaje){
    consolé.log(mensaje); }
```

Ahora podremos invocar de esta forma:

```
saludo("Hasta la vista"); //Escribe Hasta la vista por consola
```

Como hemos comentado, 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 tres a ese número. Un ejemplo de uso de esta función:

```
let x=6, y=4, z="Hola";  
consolé.log(triple(9)); //Escribe 27  
consolé.log(triple(x)); //Escribe 18  
consolé.log(triple(x+y)); //Escribe 30  
consolé.log(triple(x)+triple(y)); //Escribe 30  
consolé.log(triple(triple(9))); //Escribe 81  
consolé.log(triple(z)); //Escriba NaN
```

Podemos observar cómo al invocar la función, se admite cualquier expresión para indicar sus parámetros. En el código anterior se muestran ejemplos de esta idea:

- En la primera invocación se usa la expresión: **triple(9)**. Dentro de la función, el parámetro *n* recoge este valor y le devuelve multiplicado por tres.
- En el caso de **triple(x)**, el parámetro *n* recoge el valor de la variable *x* que es 6, por eso la función retorna 18.
- La invocación: **triple(x+y)**, hace que primero se resuelva la expresión y el valor resultante (en este caso el resultado es **10**) se pase como parámetro a la función. Por eso devuelve **30** (triple de 10).
- Más llamativo es el código: **triple(x)+triple(y)**. En este caso se debe resolver primero el código **triple(x)** que devuelve 18. Después se resuelve la expresión **triple(y)** que, como la variable *y* vale 4, devuelve el valor 12. Se suman ambos valores (30).
- Incluso un parámetro puede ser el resultado de invocar primero a la función como en: **triple(triple(9))** Lo primero será resolver la llamada más interior (**triple(9)**) que, en este caso, retorna el valor 27. Este valor se pasa como parámetro a la función que lo devolverá multiplicado por tres, es decir 81.
- Lógicamente como la variable *z* es un string. La invocación: **triple(z)** intentará multiplicar por tres ese texto, produciendo el valor no numérico **NaN**.

La función **triple** del código anterior es tan trivial que, evidentemente, no es necesario crear una función así. Se ha usado un ejemplo tan sencillo para facilitar la explicación sobre cómo crear y utilizar funciones.

Esta otra función sirve para contar la cantidad de números pares que hay en un array. No es ya tan trivial, es una función más elaborada.

```
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:

```
consolé.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, por ejemplo, al indicar los valores de un array:

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

El resultado será:

```
[ 3, 6, 9 ]
```

Las expresiones, siempre que sean válidas, ahora pueden ser muy complejas, aunque, a la vez, simplifiquen mucho nuestro trabajo:

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

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

ACTIVIDAD 5.1: PRIMERA FUNCIÓN

- En este punto conviene realizar la Práctica 5.1 "**Función de pares e impares**", en la página 190, es muy sencilla y facilita entender en qué nos ayudan las funciones.

5.2.3 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.

Como hemos comentado anteriormente, 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 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 tres. La novedad es que esa función no tiene nombre. La función es anónima, pero el código de la misma es accesible porque se le estamos asignando a la variable *trip*. Aunque *trip* es una variable, no es una función (aunque nos parezca un pequeño matiz, es importante), puede acceder al código de la función de la misma forma que si hubiera puesto ese nombre a la función:

```
consolé.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 dos variables hagan referencia a la misma función. Si añadimos este código al anterior:

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

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

5.2.4 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;
consolé.log(triple(20));
```

La definición de la función anónima es la sorprendente expresión:

```
x=>3*x
```

Que resulta ser 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 un parámetro, se deben colocar entre paréntesis. Ejemplo:

```
const media=(x,y)=>(x+y)/2;
consolé.log(media(10,20)); //Escri be 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;
}
consolé.log(sumatorio(3)); //Escribe 6, resultado de 3+2+1
```

También son necesarias las llaves cuando no hay **return**:

```
const saludo = mensaje=>{
  consolé.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 = ()=>{
  consolé.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 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 5.2: CREAR UNA FUNCIÓN FLECHA

- Crea una función flecha que sirva para devolver verdadero si un número es par y falso si no lo es.
- Asigna esa función a una variable y prueba la función para ver que responde correctamente.

5.3 DETALLES SOBRE VARIABLES Y PARÁMETROS

5.3.1 ÁMBITO DE LAS VARIABLES

Ya hemos comentado anteriormente (3.3.2.1 *"Diferencia entre let y var"*, en la página 73) que las variables tienen una duración en el código dependiendo de cómo se han declarado. Sin embargo, es en este punto en el que podemos aclarar más los aspectos sobre el ámbito de uso de las variables. 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;
    consolé.log("soy la función f");
}
f();
consolé.log(a); //error
consolé.log(b); //error
consolé.log(c); //error
```

Aunque hemos invocado a la función mediante `f()` lo cierto es que las variables `a`, `b` y `c` no se pueden utilizar fuera de la función.

Veamos este 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);      //i ¡Esta sería correcta!!
}
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`.

5.3.2 PASO POR VALOR Y PASO POR REFERENCIA

Este título de apartado le será muy familiar a todos los programadores de lenguajes clásicos. Pero para los nuevos programadores vamos a iniciar este apartado con un ejemplo:

```
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.

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/*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/pasando el valor de *lax* 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.

Indiscutiblemente es un poco enrevesada la explicación. Pero la moraleja es muy sencilla, 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. Sería más lógico este código:

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

Aún caben más tentaciones:

```
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.

Para complicar más el asunto veamos este otro código:

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

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.

5.3.3 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");
```

Por pantalla aparece:

```
Hola
Buenos días
```

La primer invocación a la función es: **sa 1 udo()** 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.

5.3.4 NÚMERO VARIABLE DE PARÁMETROS

Para entender este apartado, vamos a ver este ejemplo:

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

El resultado de este código 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 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, del que ya hemos hablado en otros apartados (véase 4.4.9 **“Desestructuración de arrays”**, en la página 140). Este operador, utilizado en los parámetros de una función, permite almacenar una serie indefinida de parámetros en un array. Veamos una idea de cómo funciona:

```
function f(x,y,...mas){
    consolé.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
}
consolé.log(media(10,20));
consolé.log(media(10,20,30));
consolé.log(media(10,20,30,40));
consolé.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. En este caso, lo es a tenor de los resultados:

```
15
20
25
30
```

Lo que no podremos hacer es enviar un array a esta función:

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

Habrá 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.

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(){
    consolé.log("Inicio f1");
    f2();
    consolé.log("Fin f1");
}

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

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

f1();
```

El resultado es:

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

PILA DE LLAMADAS

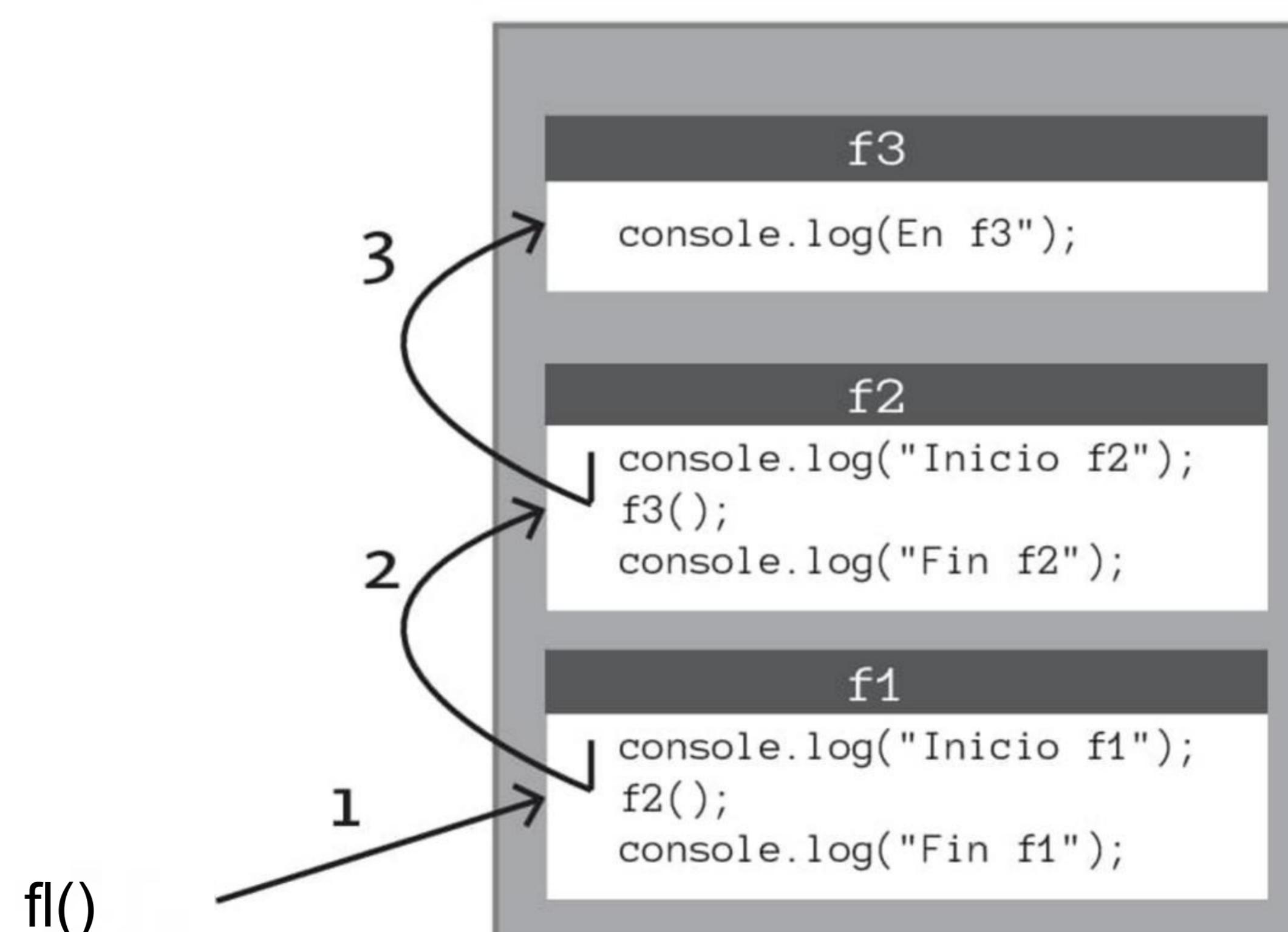


Figura 5.1: Ejemplo de pila de llamadas

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.

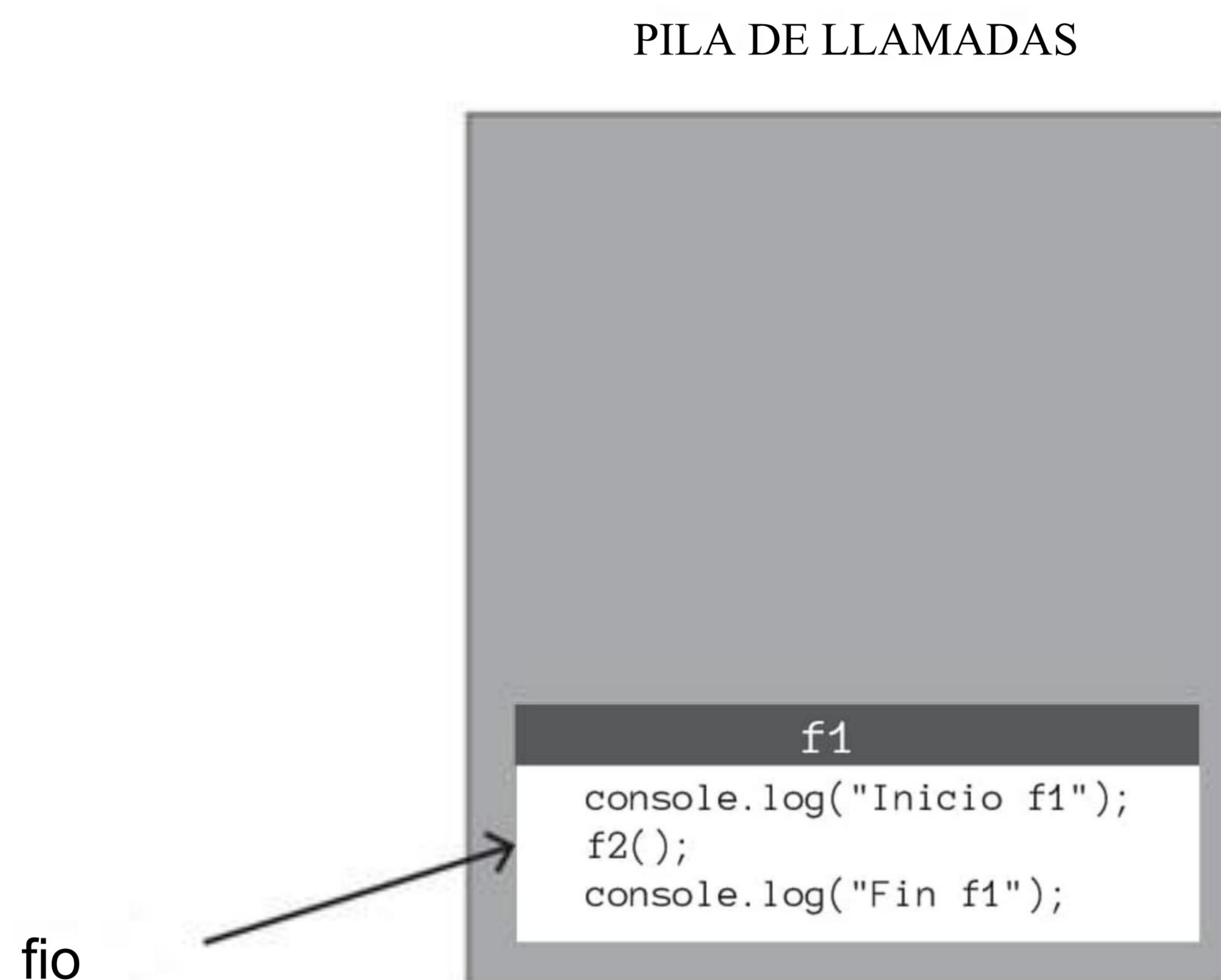


Figura 5.2: La función f1 se coloca en la pila de llamadas

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).

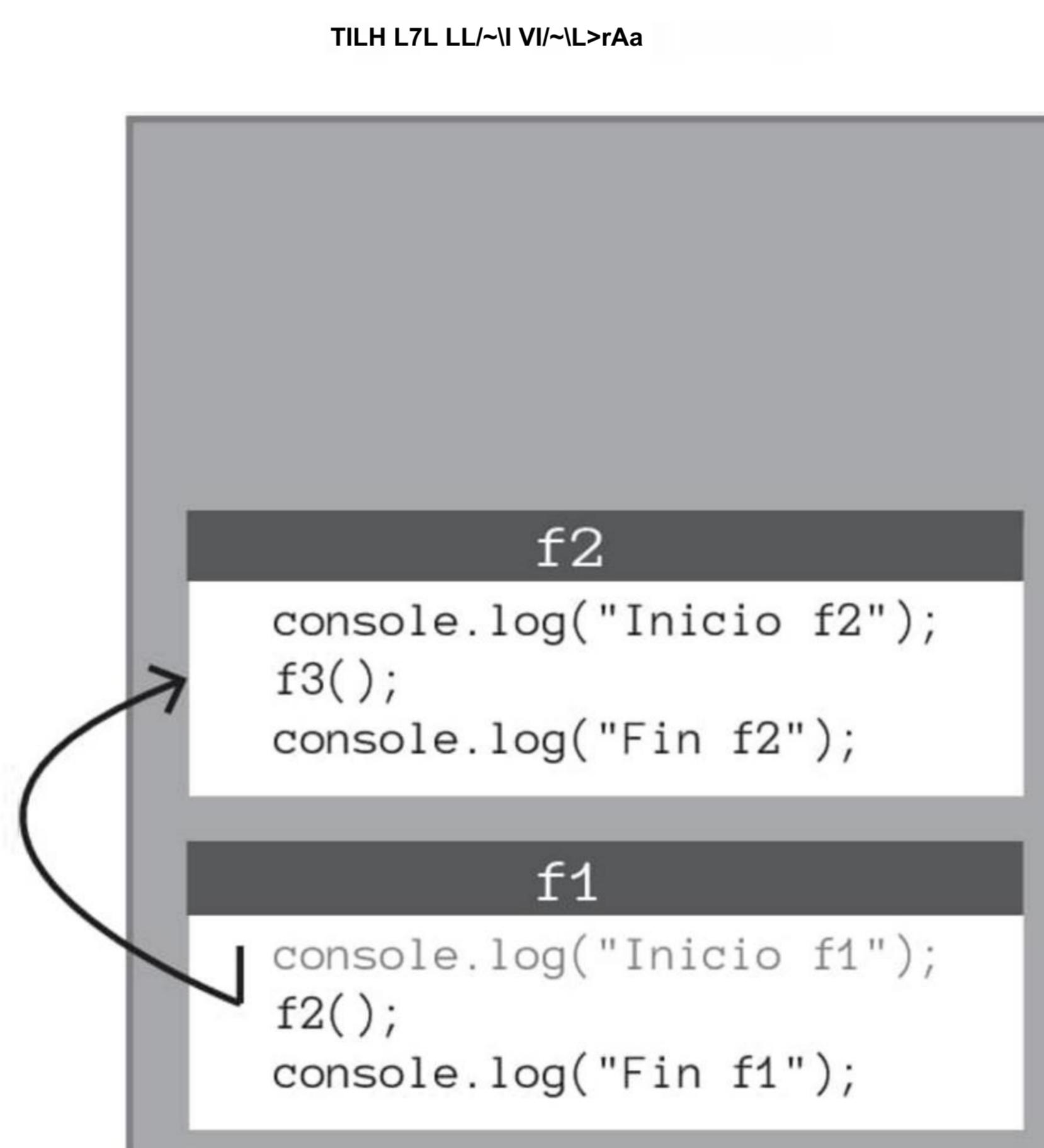


Figura 5.3: La función f2 se coloca en la pila de llamadas

- [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 (Figura 5.3) Se queda f1 a la espera de que se resuelva f2.
- [4] Se escribe ***Inicio J2***.
- [5] Se invoca a **f?**. Su código pasa a ocupar la cima de la pila. f2 se queda esperando que se resuelva el código de **f\$**.

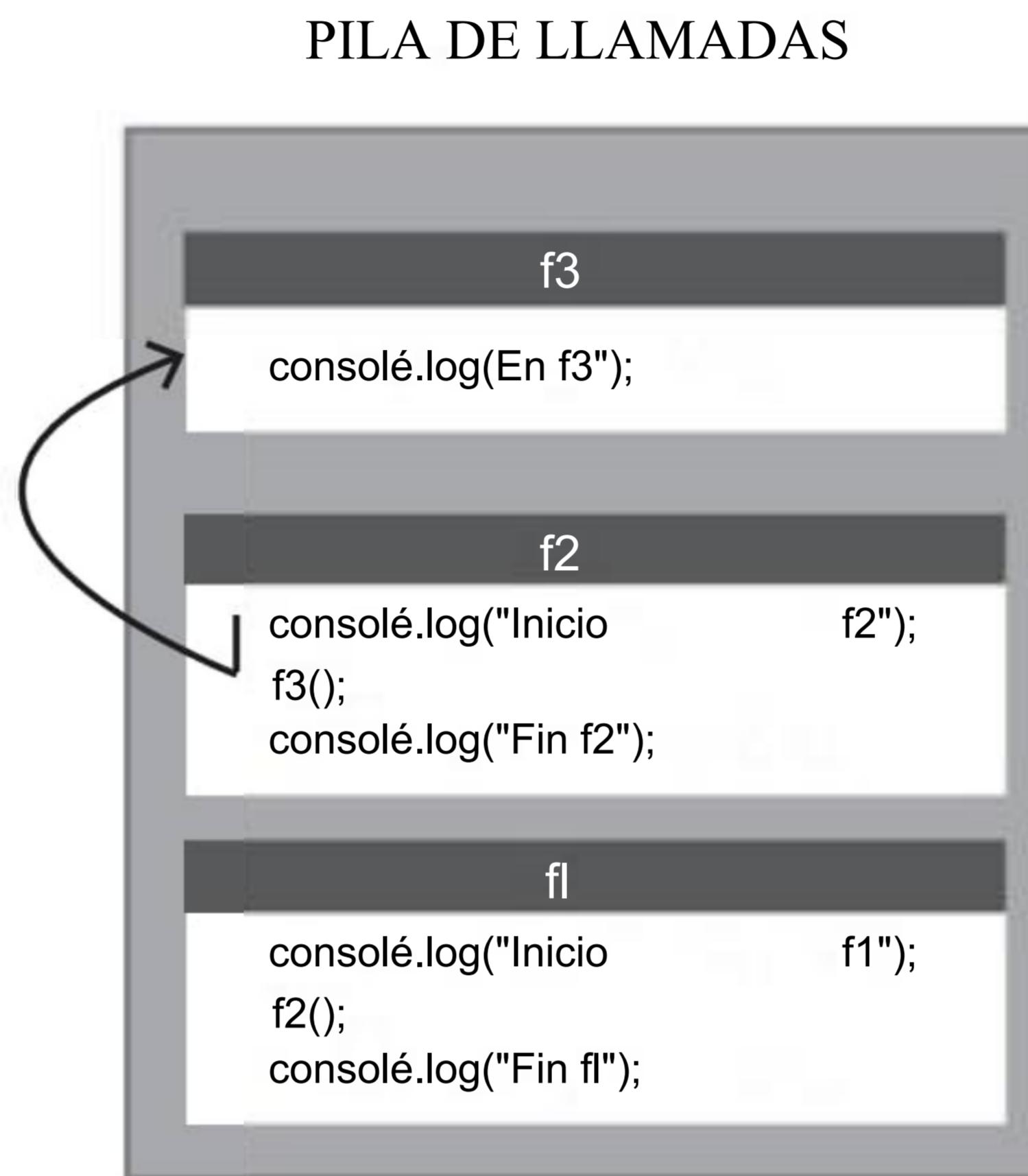


Figura 5.4: La función f3 se coloca en la pila de llamadas

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

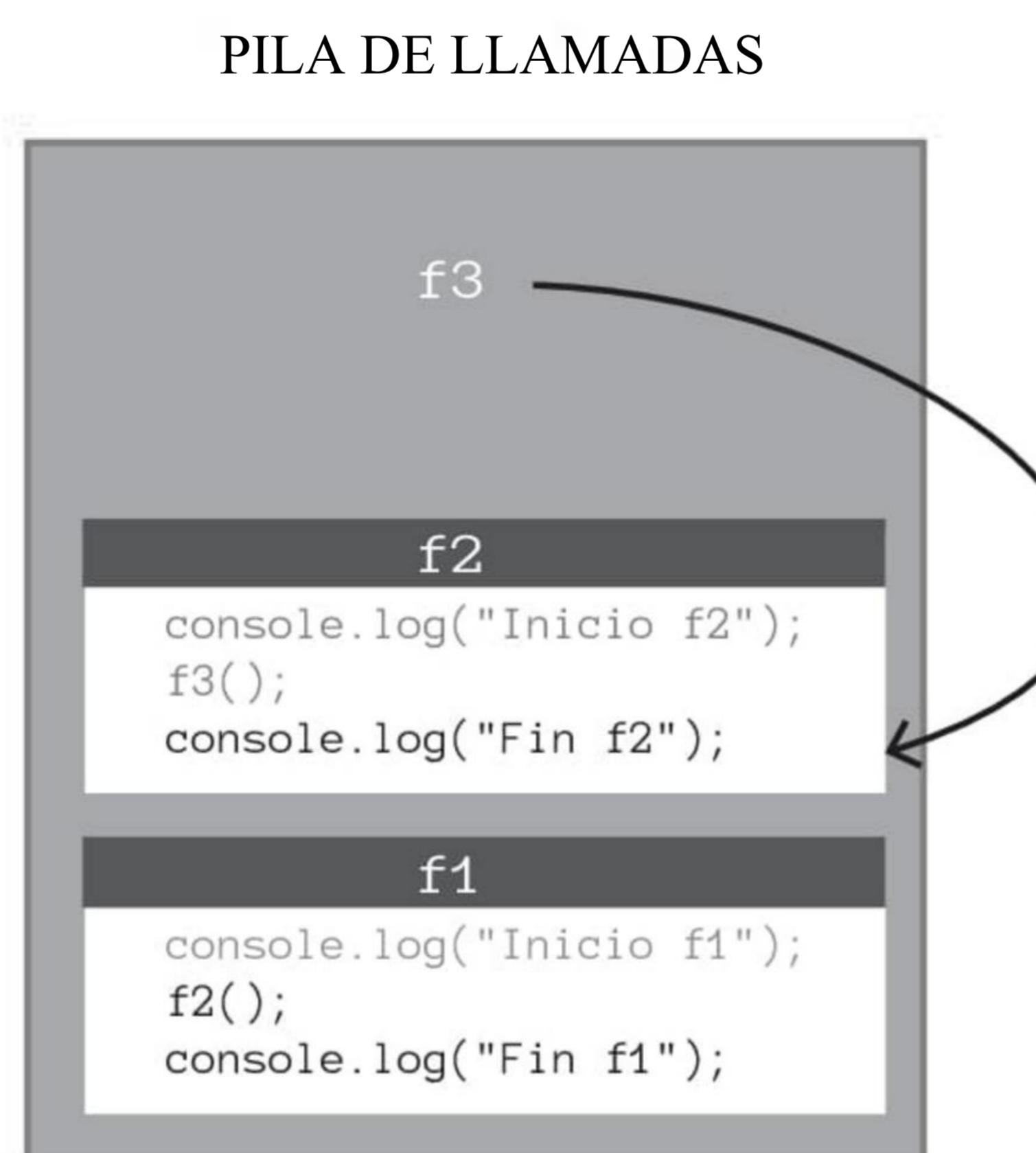


Figura 5.5: La función f3 finaliza y devuelve el control

- [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.

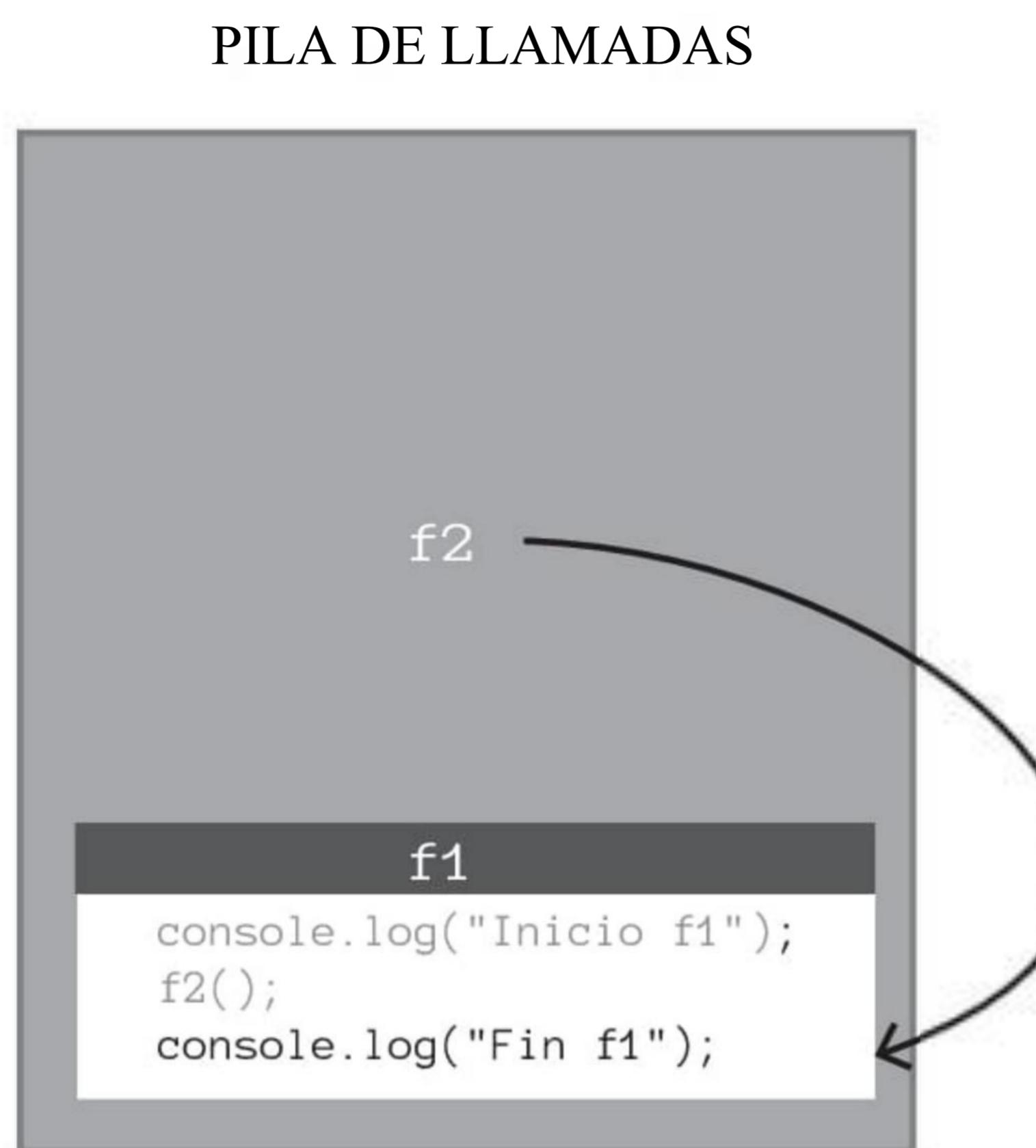


Figura 5.6: La función f2 finaliza y devuelve el control

- [10] f1 recupera el control y escribe **Fin f2**.
- [11] f1 se retira de la pila, el control se devuelve a la función principal

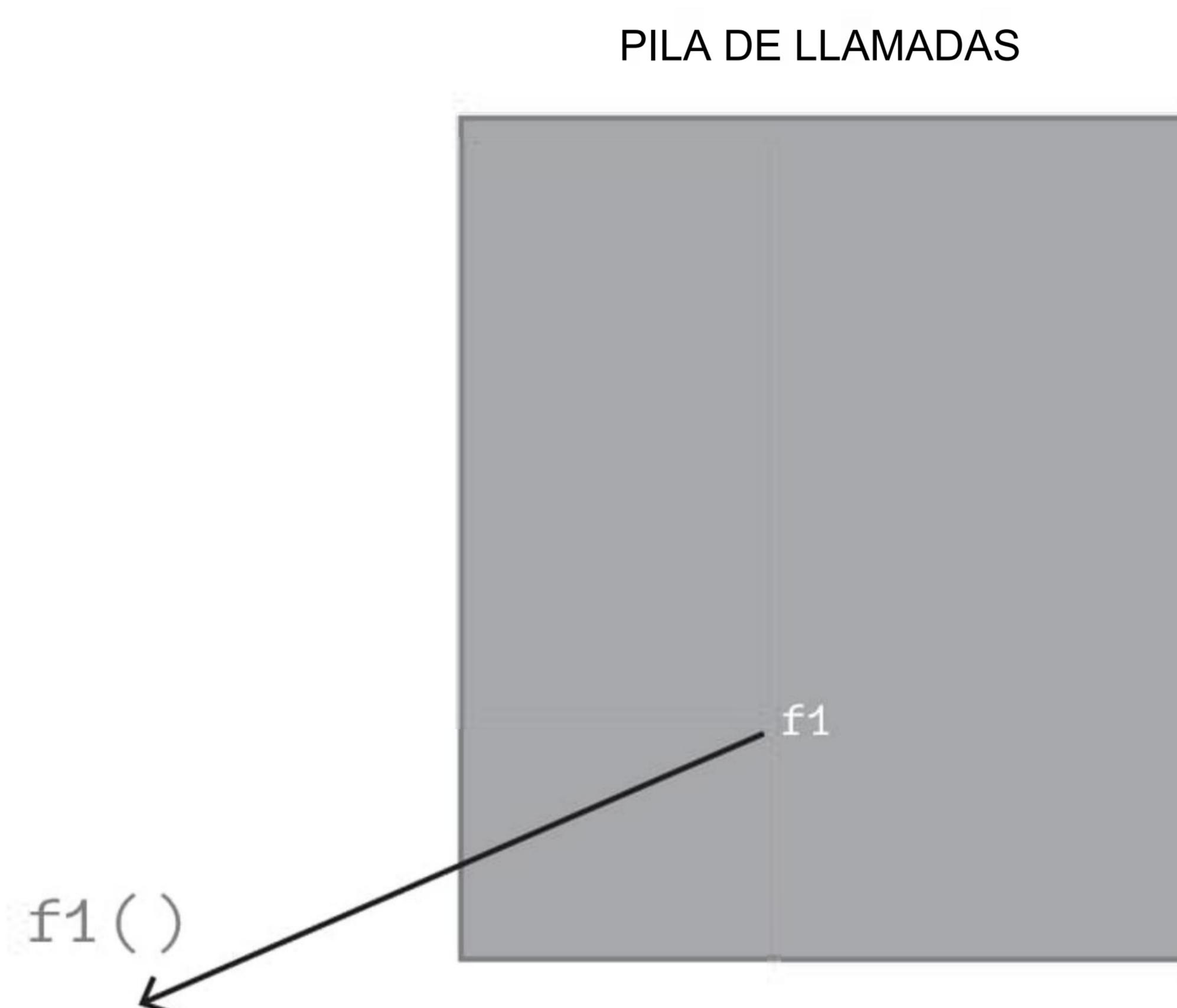


Figura 5.7: La función f1 finaliza y devuelve el control

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(){
    consolé.log("Saludo");
    despedida();
}

function despedida(){
    consolé.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.

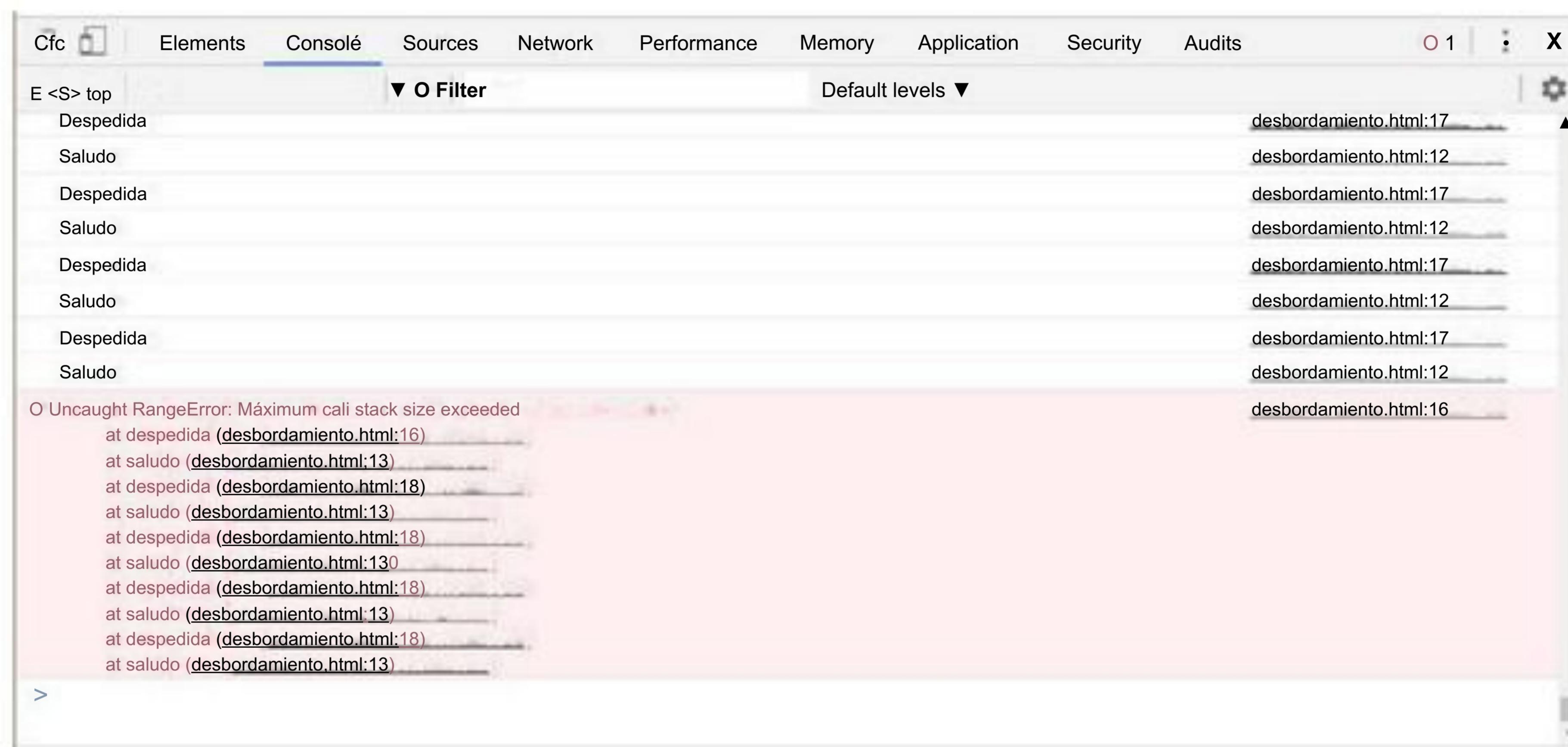


Figura 5.8: Ventana de depuración de Google Chrome informando del desbordamiento de pila que provoca el código anterior

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 $5 \cdot 4!$ (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.

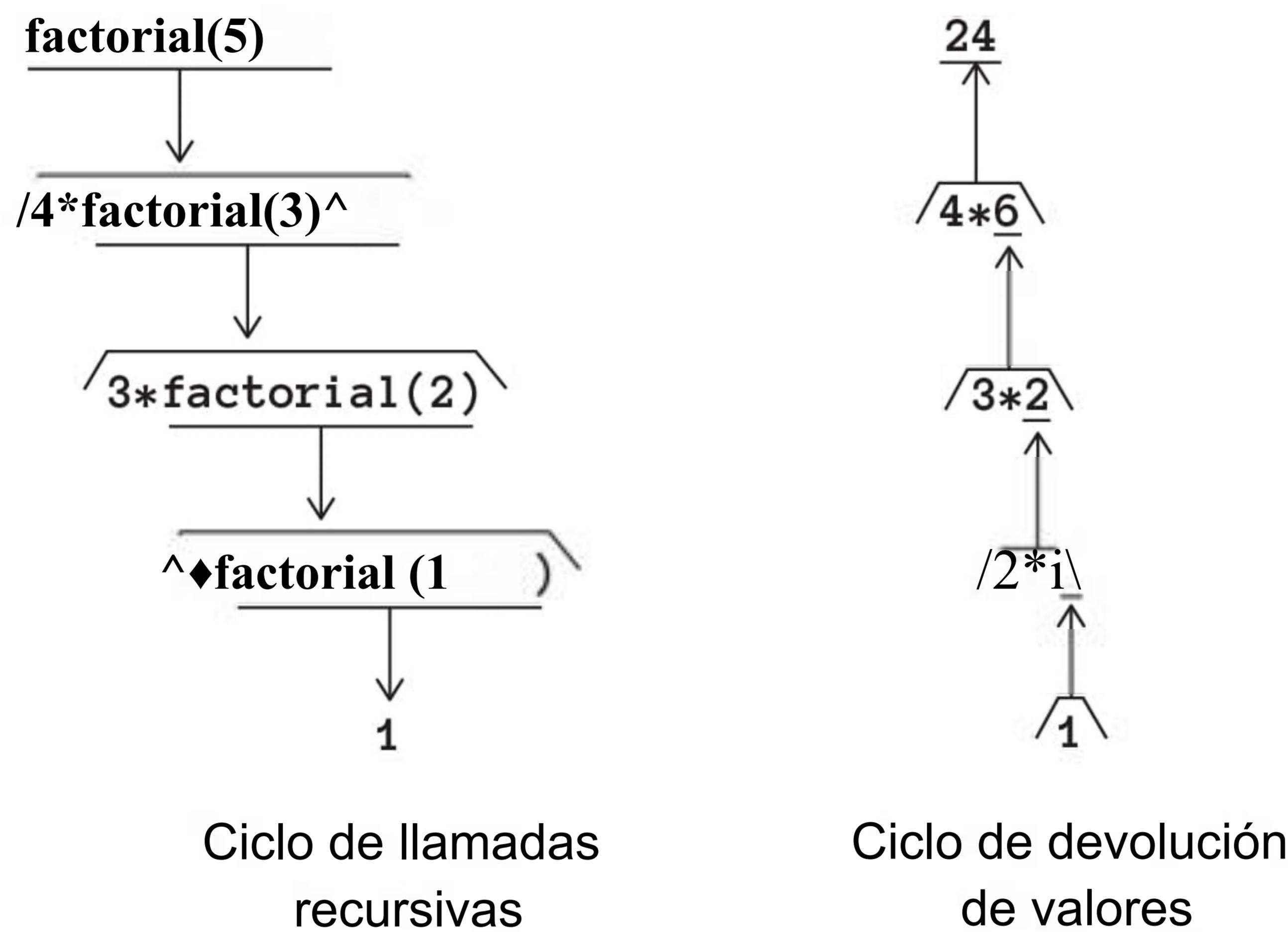


Figura 5.9: Reproducción de las llamadas recursividad en el ejemplo del factorial

ACTIVIDAD 5.3: RECUSIVIDAD

- 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 ¿RECUSIVIDAD 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 cual 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? 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,función){
    función(dato);
}
escribe("Hola",consolé.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 **consolé.log** por lo que la expresión función (dato, consolé .log) es totalmente equivalente (en este caso) a **consolé.log(dato)**.

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

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

consolé.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,acción){
    consolé.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 envíamos 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 **consolé.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 **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 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();
consolé.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);
consolé.log(palabras);
```

Consigue el resultado:

```
[ ‘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 (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));
consolé.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"));
consolé.log(palabras);
```

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

5.4.42 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,indice){
    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){
    consolé.log('La nota ${i} es ${nota}x');
});
```

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("Paúl").add("Ringo").add("George").add("John");
conjunto.forEach(function(valor){
    consolé.log(valor);
});
```

Escribirá los elementos del conjunto:

```
Paúl
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){
    consolé.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
```

5.4.43 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);
consolé.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*.

SAAA 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);
consolé.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.

5.5 PRACTICAS RESUELTAS

Práctica 5.1: Función de pares e impares

- En esta práctica crearemos una función muy sencilla que devuelve la palabra "par" si la enviamos un número par. En caso contrario retorna la palabra "impar".
 - Conseguir que se escriban en una página web 500 números aleatorios del 1 al 10.000 y que a su lado se diga si es par o impar gracias a la función anterior.
-

SOLUCIÓN: PRÁCTICA 5.1

El código de la función se debería escribir en un archivo JavaScript separado, eso permitirá utilizar la función en las aplicaciones que deseemos. El archivo podría tener este código (archivo **par.js**)

```
function par(x){  
    return (x%2==0) ? "par" : "impar";  
}
```

La página web principal debe de cargar el archivo anterior en el apartado de la cabecera (**head**), es lo habitual cuando se usan archivos con funciones. Eso permitiría usar el código JavaScript en cualquier parte del documento.

Como siempre el código JavaScript que realmente modifica la página web se coloca justo antes de cerrar la etiqueta **body**. En ese código es donde usaremos la función creada en el archivo anterior para dibujar tablas.

Práctica 5.2: Función que dibuja una tabla

- Crea una función en JavaScript que nos permita dibujar una tabla en una página web
- Como parámetros indicaremos el número de filas y de columnas con dos números. Por defecto la función tomará 10 filas y 4 columnas
- La tabla se crea con un borde negro de 1 píxel entre cada celda, pero un tercer parámetro permite indicar el color (por defecto será negro). El borde exterior medirá 3 píxeles y siempre será del mismo color que el borde de las celdas.
- La tabla ocupará toda la anchura de la página
- Usa la función para crear una tabla con borde negro de 10 filas y 4 columnas
- Úsala de nuevo para generar una tabla de 20 filas y 10 columnas, con borde negro
- Finalmente, consigue dibujar 10 tablas de 5 filas y 4 columnas que tengan borde verde.

SOLUCIÓN: PRÁCTICA 5.2

Nuevamente conviene tener el código de la función en un archivo separado (se llamará **tabla.js**)

```
function dibujaTabla(nFilas=10,nCols=4,color="black"){
    document.write(
        '<table style='border-collapse:collapse;' +
        'border:3px solid ${color};width:100%;'>' +
    );
    for(let i=1;i<=nFilas;i++){
        document.write("<tr>");
        for(let j=1;j<=nCols;j++){
            document.write(
                '<td style='border:1px solid ' +
                '${color}'>&nbsp;</td>' +
            );
        }
        document.write("</tr>");
    }
    document.write("</table>");
}
```

Este sería el código de la página principal (**index.html**):

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Función de dibujo de tablas</title>
    <script src="tabla.js"></script>
</head>
<body>
<script>
    dibujaTabla();
    dibujaTabla(20,10);
    for(let i=1;i<=10;i++){
        dibujaTabla(5,4,"green");
    }
</script>
</body>
</html>
```

Práctica 5.3: Función que permite saber si un número es primo

- Crea una función que permita saber si un número es primo o no.
- Un número es primo si no se puede dividir de forma entera por otro número, sin contar el uno ni el propio número, dando un resto de cero.
- Aprovecha la función creada para crear una página web que escriba los números primos del 1 al 1000.

SOLUCIÓN: PRÁCTICA 5.3

Hay muchas posibles soluciones y algunas mucho más eficientes que la que se presenta aquí. Pero es interesante este ejercicio para plantear cuestiones algorítmicas sobre resolución de problemas complejos.

Si no conoces algoritmos matemáticos ya resueltos sobre esta cuestión, un primer planteamiento sería ir dividiendo el número sobre el que deseamos saber si es primo desde el número dos hasta llegar al número anterior. Pero eso significa que si el número fuera, por ejemplo, el 1000 deberíamos comprobar 998 divisiones.

La verdad es fácil entender que si a la mitad de divisiones ninguna da un resto cero, no hace falta seguir dividiendo. Es más, llegando a la raíz cuadrada del número sin que ninguna división sea exacta, ya sabemos que el número es primo. Por ejemplo, para el 11, basta dividir entre 2 y 3. No tiene sentido ir más allá, si fuera divisible por el cuatro, lo sería también por un número menor que el cuatro, ya que el cuadro supera la raíz cuadrada y así con todos los demás números.

Independientemente de estos planteamientos tan matemáticos, esta solución vuelve a probar lo interesante de programar de forma modular, porque el colofón es sacar los números primos del 1 al 1000 y aunque aquí hay otros algoritmos (como la criba de Eratóstenes), lo cierto es que podemos aprovechar nuestra función para obtener los primos. No es la solución más eficiente, pero, sí es muy interesante como aplicación de la utilidad de las funciones.

Este sería el código de la función que devuelve **true** si su parámetro es primo y **false** si no lo es. Es muy interesante esta función por el uso continuo de **return**.

```
function esPrimo(n){  
    //No tiene sentido enviar como parámetro  
    //números negativos o cero, por si acaso devolvemos  
    //false en previsión de un bucle infinito  
    if(n<1) return false;  
    //El 1 le separamos, no hace falta dividir  
    if(n==1) return true;  
    //Para el resto de números empezamos a dividir entre 2  
    //y terminamos cuando la raíz cuadrada del contador supere al número  
    for(let i=2;i*i<=n;i++){  
        //si el número se puede dividir por el contador
```

```

//no tenemos un primo
if(n%i==0) return false;
}
//si hemos salido del bucle sin ejecutar el return
//tenemos un número primo
return true;
}

```

La finalización de ejercicio es el código de la página web que muestra los primos del 1 al 1000. En ella cargamos la función anterior (el archivo será **primo.js**) y después ejecutamos un bucle en el que vamos comprobando cada número. Para que sea más rápido el bucle, hemos sacado primero los números 1 y 2 (no los comprobamos) y el bucle (sabiendo que el resto de pares no son primos) empieza por el 3 y va contando de 2 en 2 (3, 5, 7, 9, 11, etc.). Así el bucle es el doble de rápido.

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" >
    <title>Números primos</title>
    <script src="primo.js"></script>
</head>
<body>
<script>
    //escribimos el 1 y 2,sabemos que son primos
    document.write("<p>1</p> <p>2</p>");
    for(let i=3;i<10000;i +=2){
        if(esPrimo(i))
            document.write('<p>$ {i}</p>');
    }
</script>
</body>
</html>

```

Práctica 5.4: Ordenar palabras en orden inverso

- Crea una aplicación que pida al usuario palabras continuamente hasta que se acepte el cuadro sin texto o se cancele.
- Se eliminarán las palabras repetidas y además se ordenarán en español, pero en orden inverso (de la Z a la A).

SOLUCIÓN: PRÁCTICA 5.4

Este sería el código JavaScript a colocar en la aplicación web:

```
const setPalabras=new Set();
var arrayPalabras=[];
//hacemos una primera lectura adelantada
var palabra=prompt("Escriba una palabra (o nada si desea acabar");
while(palabra !=null && palabra != ""){
    setPalabras.add(palabra);
    palabra=prompt("Escriba una palabra (o nada si desea acabar");
}
//convertimos en array
arrayPalabras=[...setPalabras];
//ordenamos por lo contrario a localCompare
//por eso ponemos un signo "menos" por delante
arrayPalabras.sort((a,b)=>(-a.localeCompare(b,"es")));
//recorremos el array usando forEach
//(podríamos) usar cualquier otro bucle
arrayPalabras.forEach(function(pal){
    document.write('<p>$ {pal}</p>');
});
```

Práctica 5.5: Crear mapa con repeticiones de arrays

- Crea una función que reciba un array de palabras.
- La función devolverá un mapa que contenga como clave cada palabra y el valor es el número de veces que esa palabra aparece en el array.
- Haremos una página web que lea palabras hasta que el usuario cancele o deje el cuadro vacío y mostraremos las repeticiones de las palabras.

SOLUCIÓN: PRÁCTICA 5.5

El código de la función lo grabaremos en el archivo repeticiones.js

```
function mapaRepeticiones(array){
    if(array instanceof Array == false){
        //no es un array
        return null;
    }
    else{
        let mapa=new Map();
        //en este caso llamamos clave a cada valor del array
        //porque serán las claves del mapa
        for(let clave of array){
            //comprobamos si el valor está en el mapa
```

```

if(mapa.get(clave)Hunde fined){
    //si lo está incrementamos el contador
    mapa.set(clave,mapa.get(clave)+l);
}
else{
    //si no, la añadimos con contador a 1
    mapa.set(clave,1);
}
return mapa;
}
}

```

El código javaScript que se encarga de pedir al usuario palabras e indicar las veces que se repite cada una, es el archivo **accion.js**. Usa la función definida en el archivo anterior.

```

const arrayPalabras= [];
let mapa;
//hacemos una primera lectura adelantada
var palabra=prompt("Escriba una palabra (o nada si desea acabar)");
while(palabra Hnull && palabra!=""){
    arrayPalabras.push(palabra);
    palabra=prompt("Escriba una palabra (o nada si desea acabar");
}
//ordenamos el array
arrayPalabras.sort((a,b)=>(a.localeCompare(b,"es")));
//convertimos en mapa
mapa=mapaRepeticiones(arrayPalabras);
//recorremos el mapa y mostramos las repeticiones
for([palabra,cont] of mapa){
    document.write( <p>$!palabra¡, ${cont repeticiones} );
}

```

Finalmente la página carga en la cabecera el código de la función para asegurar su disponibilidad en el resto de archivos. En el cuerpo carga la acción en sí de la página.

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Repeticiones de palabras</title>
    <script src="repeticiones.Js"> </script>
</head>
<body>
    <script src=accion.js></script>
</body>
</html>

```

Práctica 5.6: Función recursiva Fibonacci

- La función de Fibonacci es un clásico de la programación recursiva. Se trata de una función relacionada con una sucesión de elementos donde los dos primeros son el cero y el uno, y el resto son la suma de sus dos anteriores.
- Es decir, la sucesión es 0,1,1, 2, 3, 5, 8,13, 21, 34, 55, 89, etc.
- A la función le mandaríamos el número del que queremos saber el valor Fibonacci y nos devolvería dicho valor. Así si pasamos el número 10, devolvería 55.
- Es mucho más fácil la solución recursiva, pero sería muy interesante hacer también la solución no recursiva.

SOLUCIÓN: PRÁCTICA 5.6

La idea recursiva para del hecho de que la función de Fibonacci de un determinado número es la suma de los dos anteriores. Si no tenemos cuidado este genera una pila de llamadas infinita, pero las llamadas finalizan en el número uno y en el cero, ya que de estos enviamos los valores uno y cero respectivamente. Este sería el código de la función:

```
function fibonacci(x){
    if(x==0) return 0;
    else if(x==1) return 1;
    else return fibonacci(x-1)+fibonacci(x-2);
}
```

Como se puede apreciar el código es muy sencillo. Para la solución sin recursividad debemos ir recorriendo los elementos y acumulando las sumas, es algo más difícil de programar, pero es más eficiente:

```
function fibonacci 2(x){
    //acumuladores de los dos elementos anteriores
    let acul=1;
    let acu2=0;
    //acumula el resultado final
    let fibo=1;

    if (x==0) return 0;
    else if (x==1) return 1;
    else{
        for(let i=2;i <=x;i++){
            //se actualizan los valores para la siguiente suma
            [acul,acu2]=[acu2,fibo];
            //resultado
            fibo=acul+acu2;
        }
        return fibo;
    }
}
```

5.6 PRÁCTICAS RECOMENDADAS

Práctica 5.7: Función para detectar palíndromos

- Crea una función para resolver lo que se pedía en la Práctica 4.7 "*Palíndromos*", en la página 157.
 - La función recibe un texto y devolverá verdadero si es un palíndromo y falso si no lo es.
 - Hay que tener en cuenta que para que se consideren bien los palíndromos, se ignoran los signos de puntuación (espacios, interrogaciones, comas, puntos, etc.) también se ignoran tildes y diéresis (se considera igual el carácter *á* que el carácter *a*) y no se distingue entre mayúsculas y minúsculas.
-

Práctica 5.8: Función para detectar anagramas

- Crea una función a la que se le pasen una serie de textos (mínimo dos) y detecte si los mismos son anagramas o no.
 - Un anagrama es una palabra que resulta de trasponer las letras de otra: por ejemplo *ESTANCO* y *ACENTOS*.
 - La función devuelve verdadero si todas las palabras pasadas son anagramas de las mismas letras.
-

Práctica 5.9: Función Tribonacci

- Basada a la sucesión de Fibonacci, tenemos una sucesión conocida como **Tribonacci**. En ella cada elemento es la suma de los tres anteriores.
 - La sucesión es 1,1, 2,4, 7,13,24,44, 81,149,274, etc.
 - Resuelve la función de forma recursiva y de forma iterativa.
-

Práctica 5.10: Crear función "Filtro"

- El método *filter* de los arrays visto en este tema, permite indicar una función **callback**, para aplicar un filtro a los elementos de un array.
- Es buena práctica para aprender a implementar funciones **callback** tratar de crear nuestras propias funciones

- Crea una función llamada **filtro** que reciba un array y una función callback. La función callback se entenderá que tendrá un solo parámetro y que devuelve verdadero o falso dependiendo del criterio que establezcamos.
 - Nuestra función **filtro** retornará un nuevo array con los elementos que cumplan los criterios establecidos en la función callback.
 - Pruébala con varios arrays y funciones de filtro que establezcas a medida.
-
-

Práctica 5.11: Mapa buscaminas

- Crear una aplicación web que muestre un mapa del popular juego buscaminas en el que aparezcan las minas dibujadas y también que se indique en las casillas sin minas, las minas que hay alrededor.
- Hacerlo de forma modular de manera que dividamos la aplicación en una serie de funciones. Concretamente recomendamos:
 - Una función a la que le pasemos el tablero buscaminas (sería un array de dos dimensiones) y coloque en él de manera aleatoria las minas. Esta podría dividirse en dos, siendo una más sencilla la que recibe el array del tablero, una posición en él y devuelve las minas alrededor de esa posición. La función principal simplemente invoca a esa segunda recorriendo cada casilla
 - Otra función que recorra el tablero marcando en cada casilla las minas que hay alrededor. Finalmente una tercera función que dibuje el tablero en una página web
- Se pedirá al usuario el tamaño del tablero y las minas a colocar.
- Ejemplo de resultado final (tablero de 9x9 y 16 minas):

1	MINA	1		1	MINA	2	1	MINA
	1	2	1	2	2	MINA	3	2
		1	MINA	2	2	2	MINA	1
1	1	2	2	MINA	2	2	1	1
1	MINA	2	3	4	MINA	1	1	1
1	2	3	MINA	MINA	3	3	MINA	1
	1	MINA	3	4	MINA	4	2	2
1	2	1	1	2	MINA	3	MINA	1
1	MINA	1		1	1	2	1	1

5.7 RESUMEN DE LA UNIDAD

Las funciones facilitan la modularidad a la hora de programar aplicaciones. Cada función será capaz de resolver una tarea sencilla a partir de una serie de datos que la función requiere para poder ejecutar su tarea.

Las funciones JavaScript se declaran con la palabra clave **function** a la que sigue el nombre de la función, los parámetros de la misma (que son los datos que requiere para poder resolver la tarea) y el cuerpo de la función (que es el código que la función ejecute cuando se la invoque).

En el cuerpo de la función puede aparecer la palabra **return** que es la que permite que la función devuelva un valor concreto. No es obligatorio que aparezca, hay funciones que no retornan un valor concreto.

Hay funciones anónimas cuyo cuerpo se puede asignar a variables o parámetros de funciones, que pasan a ser referencias a la propia función.

Las funciones flecha permiten definir funciones de forma más cómoda sin tener que utilizar la palabra **function** ni, en muchas ocasiones, tener que indicar explícitamente la palabra **return**. Son cómodas para funciones simples, pero no tanto para funciones complejas.

Cuando una función se invoca indicando tipos simples de datos (números, booleanos, strings, etc.) en los parámetros, estos reciben una copia de esos valores. Pero si enviamos arrays u otros objetos, entonces los parámetros obtendrán una referencia al objeto original. En este último caso, si se modifica el objeto a través del parámetro, se estará modificando el objeto original y no una copia.

JavaScript admite parámetros de funciones con valores predeterminados y, también definir parámetros con el operador de propagación para un número indeterminado de parámetros.

Cada vez que invocamos a funciones, estas se colocan en una pila que tiene un tamaño finito y que podríamos colapsar en caso de excesivas llamadas a funciones sin resolver.

La recursividad es una técnica que permite que una función se invoque a sí misma. Con habilidad permite resolver de forma sencilla, problemas complejos.

Las funciones **callback**, son funciones que recogen los parámetros de otras funciones. Así, las propias funciones reciben datos y, además, acciones que pueden realizar.

El uso práctico de funciones callback queda de manifiesto en tareas como ordenar arrays (**sort**), recorrer elementos de objetos iterables (**forEach**), hacer cálculos con todos los elementos de un array (**map**) o filtrar los elementos de un array en base a una condición (**filter**).

5.8 TEST DE REPASO

1.- ¿Qué modelo de programación de aplicaciones está relacionado con el uso de funciones?

- a) Programación estructurada
- b) Programación modular
- c) Programación orientada a objetos
- d) Programación orientada a eventos

2.- ¿Qué muestra este código por pantalla?

```
function f1(...a){
    consolé.log(a)
}
f1(1,2,3,4);
```

- a) Set { 1, 2, 3, 4 }
- b) Map { 1=>1,2=>2,3=>3,4=>4 }
- c) 1, 2, 3, 4
- d) [1, 2, 3, 4]

3.- ¿Qué muestra este código?

```
function f2(a,b,c=10){
    consolé.log('a=${a}, b=${b},'+
               'c=${c}');
}
f2(1)
```

- a) a=1
- b) a=1, b=, c=10
- c) a=1, b=null, c=10
- d) a=1, b=NaN, c=10
- e) a=1, b=undefined, c=10
- f) Ocurre un error, no se muestra nada

4.- ¿Qué es una función flecha?

- a) Una función anónima declarada con una sintaxis más sencilla
- b) Una función que aplica técnicas de recursividad
- c) Una función donde uno de sus parámetros es una referencia a otra función
- d) Una función que no utiliza la palabra clave **return**.

5.- ¿Qué escribe este código?

```
function f3(n){
    if(n<=1) consolé.log(n);
    else {
        consolé.log(n);
        consolé.log(f3(parseInt(n/2)));
    }
}
f3(19);
```

- a) 19 19 19 19 19 1
- b) 19 19 19 19 1
- c) 19 9 4 2 1
- d) 19 10 5 3 1

6.- ¿Qué muestra por pantalla este código?

```
var x=19;
function f4(){
    consolé.log(x);
}
f4();
consolé.log(x);
```

- a) undefined 19
- b) undefined undefined
- c) 19 19
- d) Ocurre un error

7.- ¿Qué muestra por pantalla este código?

```
let x=19;
function f5(){
    consolé.log(x);
}
f5();
consolé.log(x);
```

- a) undefined
19
- b) undefined
undefined
- c) 19
19
- d) Ocurre un error

8.-

9.- ¿Qué muestra por pantalla este código?

```
function f6(){
    let x=19;
    consolé.log(x);
}
f6();
consolé.log(x);
```

- a) undeñned
19
- b) undeñned
undeñned
- c) 19
19
- d) Ocurre un error

10.- ¿Qué muestra por pantalla este código?

```
function f6(){
    var x=19;
    consolé.log(x);
}
f6();
consolé.log(x);
```

- a) undeñned
19
- b) undeñned
undeñned
- c) 19
19
- d) Ocurre un error

11.- ¿Qué muestra por pantalla este código?

```
function f7(a,b){
    return b(a);
}
consolé.log(f7(9,x=>x+2));
```

- a) Ocurre un error
- b) function
- c) object
- d) 11

12.- ¿Qué muestra por pantalla este código?

```
function f8(s){
    s="no";
    return s;
}
let t="si";
f8(t)
consolé.log(t);
```

- a) si
- b) no
- c) undeñned
- d) Ocurre un error

13.- ¿Qué muestra por pantalla este código?

```
function f9(a){
    a=["no"];
    return a;
}
let b=["si"];
f9(b);
consolé.log(b);
```

- a) ['si']
- b) ['no']
- c) undeñned
- d) Ocurre un error

14.- ¿Qué muestra por pantalla este código?

```
function f10(a){
    a[0]="no";
    return a;
}
let b=["si"];
f10(b);
consolé.log(b);
```

- a) ['si']
- b) ['no']
- c) undeñned
- d) Ocurre un error