

```
254  
255 ■ function updatePhotoDescription() {  
256 ■■ if (descriptions.length > (page * 9) + (currentImage - 1)) {  
257 ■■■ document.getElementById('bigImageDesc').innerHTML = descriptions[page * 9 + currentImage - 1];  
258 ■ }  
259 }  
260  
261 ■ function updateAllImages() {  
262 ■■ var i = 1;  
263 ■■ while (i < 10) {  
264 ■■■ var elementId = 'foto' + i;  
265 ■■■ var elementIdBig = 'bigImage' + i;  
266 ■■■ if (page * 9 + i - 1 < photos.length) {  
267 ■■■■ document.getElementById(elementId).src = 'images/small/' + photos[page * 9 + i - 1];  
268 ■■■■ document.getElementById(elementIdBig).src = 'images/big/' + photos[page * 9 + i - 1];  
269 ■■■ } else {  
270 ■■■■ document.getElementById(elementId).src = '';
```

Funciones

Objetivos

- Descubrir qué ventajas proporciona el uso de funciones.
- Conocer la sintaxis de todas las alternativas posibles para escribir funciones.
- Entender cómo afecta el ámbito de las variables a los programas.
- Interiorizar el funcionamiento de los parámetros y sus tipos.
- Analizar todos los tipos de funciones y entender sus diferencias y propósitos.
- Introducir la estrategia recursiva para resolver problemas.
- Extender la funcionalidad de estructuras de datos usando funciones.

Contenidos

- 4.1. Ventajas
- 4.2. Anatomía de una función
- 4.3. Ámbito de variables y tipos de parámetros
- 4.4. Tipos de funciones
- 4.5. Recursividad
- 4.6. Más funcionalidad para las estructuras de datos

Introducción

Las funciones representan el corazón de la programación en JavaScript. Son de tal importancia que es prácticamente imposible encontrar un programa escrito en este lenguaje que no incluya un gran número de ellas. Su versatilidad, eficiencia y flexibilidad las convierten en una potente herramienta para desarrollar aplicaciones web complejas.

En esta unidad se verá a un nivel bastante alto qué puede conseguirse con ellas y tras su estudio se estará en disposición de adentrarse en el nivel más avanzado del desarrollo de aplicaciones web del lado cliente con JavaScript.

4.1. Ventajas

Las funciones vienen a resolver uno de los principales problemas de la programación estructurada. A la hora de desarrollar una aplicación se puede dar el caso de que entre la lista de tareas que resuelve hay una que se repite de manera constante a lo largo de todo el programa. Habría que reescribir el código que resuelve esa tarea muchas veces en distintos puntos del programa. Si en algún momento se detectara un error en esa solución, o por ejemplo se quisiera mejorar su eficiencia, habría que buscar todas las porciones del código donde aparece la solución y actualizarla. Lo cual es una tarea tediosa e improductiva que consume muchos recursos de forma innecesaria.

Las funciones resuelven este problema, de manera que puede crearse una función (asignarle un nombre) y dentro de ella resolver la tarea en cuestión. En todos los sitios del programa donde hubiera que incorporar la solución, simplemente se haría referencia a ese nombre. Si por cualquier motivo se tuviera que modificar esa pieza de código, se haría en un único punto del programa (donde se definió la función) y el cambio repercutiría a todo el programa.

Este concepto de encapsulamiento y reutilización es la base de la modularidad. Se trata de crear piezas de código (funciones) que resuelven problemas concretos, más pequeños que la solución global, y alcanzar esta última combinando varias de esas piezas. Al mismo tiempo, si en otro programa se presentara la necesidad de resolver la misma tarea u otra muy parecida, puede reutilizarse el código ya probado de la otra aplicación, lo que ahorra una cantidad enorme de tiempo de desarrollo. Dicho de otra manera, las funciones permiten reutilizar el código, mejorar la eficiencia, aumentar su legibilidad y reducir los costes de mantenimiento de los programas.

Actividad resuelta 4.1

Login modularizado

Piensa en el proceso de inicio de sesión de una aplicación web cualquiera y crea una descomposición del problema en problemas más pequeños que podrías implementar con funciones.

Solución

En primer lugar, los procesos de inicio de sesión (la aparición del formulario de *login*) suelen activarse tras pulsar un botón. Esa podría ser la primera función: *muestraLogin()*.

Normalmente se puede acceder indicando un nombre y una contraseña o utilizando los procesos de *login* de otros servicios externos como Google o Facebook. Se podría crear una función para cada uno de esos sistemas de identificación: *loginGoogle()*, *loginFacebook()*, *loginUserPass()*. Dentro de cada una de las dos primeras se implementaría el acceso a la API de ambos servicios. En la tercera tendríamos que implementar nuestro propio proceso de autenticación.

En esta última función también podríamos tener una función para cada tarea común que suelen involucrar estos procesos: *checkForm()* para comprobar que se han introducido correctamente los datos y *forgotPass()* para iniciar el proceso de recuperación de la contraseña.

Toda esta estrategia puede granularse aún más hasta llegar a una descomposición más pequeña, aunque en este punto y para entender la modularidad podemos darnos por satisfechos.

4.2. Anatomía de una función

Existen muchos tipos de funciones, pero todas ellas y salvo contadas excepciones (que se verán más adelante) comparten una serie de elementos que se indican en la Figura 4.1.

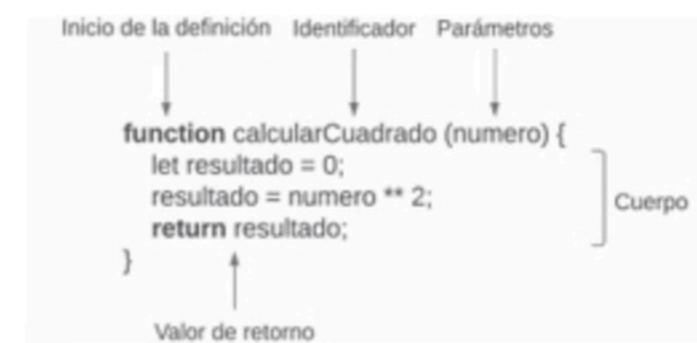


Figura 4.1. Diagrama con los elementos constitutivos de una función en notación declarativa.

El papel que juegan cada uno de estos elementos en la estructura de una función es el siguiente:

- Inicio de la estructura:** una función comienza con la palabra clave **function**. Más adelante se verá que no tiene por qué ser así, ya que existen formas mucho más simplificadas de establecer la estructura de una función. Pero, de momento, se trata de transitar por el camino sencillo.

- **Identificador:** es el nombre que el programador asigna a la función. Es el nombre al que se hace referencia para ejecutar la función. Una vez más, es necesario puntualizar que esto no siempre es así, puesto que existen funciones anónimas.
- **Parámetros:** a una función se le pueden pasar datos desde el exterior, con los que puede trabajar en su interior. Entre los paréntesis es donde se especifica si recibe parámetros, cuántos, sus identificadores y otros elementos que se verán en secciones posteriores.
- **Cuerpo:** es el bloque de instrucciones que se ejecutan cuando se llama a la función, su utilidad, su propósito.
- **Valor de retorno:** es el valor opcional que la función devuelve al exterior, al punto del programa desde el que se la invocó. En JavaScript, a diferencia de otros lenguajes, no es obligatorio su uso.

4.2.1. Definición

Para poder usar (llamar o invocar) una función es necesario que se haya definido previamente.

Nota técnica



A veces tienden a confundirse los conceptos de definición y declaración, llegándose a utilizar de manera indistinta. Sin embargo, para programadores que dominan varios lenguajes de programación, esta circunstancia puede generarles problemas de comprensión si se mezclan los conceptos.

Una **declaración** es simplemente un prototipo que indica el nombre de un procedimiento o función, su identificador y su lista de parámetros (como ocurre en C).

Una **definición**, por su parte, incluye todo lo anterior y además el cuerpo en el que se especifican las operaciones que realiza (como ocurre en JavaScript).

Hay muchas formas de definir una función (como se verá más adelante); precisamente esa es una característica que da idea de la flexibilidad de las funciones en JavaScript, pero al menos al inicio se utilizará la conocida como **notación declarativa**, como en este ejemplo:

```
function mensajeAlerta() {
    alert("Cuidado, errores en el formulario");
}
```

Como se observa, la definición la inicia la palabra reservada **function**, seguida del identificador de la función, **mensajeAlerta**, una lista (en este caso vacía) de parámetros y un cuerpo de la función donde se incluyen las instrucciones que la componen. No se ha indicado un valor de retorno con **return** porque en este caso no tiene sentido, ya que simplemente su cometido es mostrar una ventana del navegador con el mensaje que aparece entre comillas.

Al incluir este código en un fichero y ejecutarlo en el navegador no se vería nada, puesto que para usar funciones no es suficiente con definirlas, sino que hay que invocarlas.

4.2.2. Invocación de una función

Para invocar a una función (llamarla o usarla son sinónimos) se utiliza su identificador junto con los parámetros que se desea pasarle. Por ejemplo:

```
mensajeAlerta();
```

Tras la ejecución de esa instrucción ya sí que se vería un resultado en el navegador (Figura 4.2).



Figura 4.2. Diálogo de alerta del navegador.

También es importante recordar que, aunque una función se haya definido sin parámetros, hay que escribir los paréntesis igualmente.

¿Cómo sería el uso de una función que sí recibe parámetros? El siguiente ejemplo es una actualización de la función anterior:

```
function mensajeAlerta(mensajeExterior) {
    alert(mensajeExterior);
}
let mensajePersonalizado = "Error definido por el usuario";
mensajeAlerta(mensajePersonalizado);
```



Figura 4.3. Cuadro de diálogo de alerta del navegador.

Ahora la función **mensajeAlerta** recibe un parámetro desde el exterior, precisamente el mensaje que se crea fuera de la función y que se ha incluido en la llamada de esta.

Para terminar de completar el repaso a la definición declarativa de funciones solo resta incluir algún valor de retorno. La siguiente función permite calcular la raíz cuadrada del número que se le pasa como parámetro:

```
function raizCuadrada(numero) {
    return (Math.sqrt(numero));
}
console.log(raizCuadrada(4)); // muestra 2
```

Se define la función **raizCuadrada** que recibe un parámetro llamado **número** y devuelve la raíz cuadrada de ese número con la ayuda de la librería **Math** integrada en JavaScript. Para ejecutar la función se la invoca incluyendo entre paréntesis el número 4, que será con el que trabaje la función en su interior.

Además, al tener la función definida con un valor de retorno, puede tratarse como si fuera una variable que contiene un valor (el que devuelve), de modo que es posible realizar invocaciones como las siguientes:

```
console.log(raizCuadrada(4+5)); // muestra 3
console.log(raizCuadrada(4)+raizCuadrada(9)); // muestra 5
console.log(raizCuadrada(raizCuadrada(16))); // muestra 2
console.log(raizCuadrada(11-3)); // muestra 2.8284271247461903
```

Otra característica interesante de las funciones es que desde su propio cuerpo se puede llamar a otras funciones, ya sean predefinidas o creadas por el programador. La prueba está en la función anterior. Tanto **sqrt()** como **log()** son funciones que se han usado dentro de una función definida por el programador, aunque podría haberse hecho con dos funciones personalizadas. Por ejemplo, el siguiente programa utiliza la función **raizCuadrada** dentro de otra función llamada **calcularMayor** cuyo cometido es calcular la mayor raíz cuadrada de los elementos de un **array**.

```
function raizCuadrada(numero) {
    return (Math.sqrt(numero));
}
function calcularMayor(vector) {
    let mayor = raizCuadrada(vector[0]);
    for (let i=0; i<vector.length; i++) {
        if (raizCuadrada(vector[i]) > mayor)
            mayor = raizCuadrada(vector[i]);
    }
    return mayor;
}
console.log(calcularMayor([64,128,4,1024,16]));
// muestra 32
```

4.2.3. Valores de retorno

El valor de retorno de una función es realmente el dato más importante de la misma, puesto que es el resultado que la función devuelve a quien la invocó.

Siempre debe tenerse mucho cuidado con las expresiones que se indican en el **return** de una función puesto que, en ocasiones, la instrucción que invocó a la función está esperando un tipo de dato en concreto que no coincide con el tipo de dato que le está devolviendo la función. Lo cual suele ser una frecuente fuente de errores o inconsistencias en los programas.

Este otro ejemplo muestra una función que devuelve **true** si el número que recibe como parámetro es par, o **false** si es impar:

```
function esPar(numero) {
    if (numero%2 == 0)
        return true;
    else
        return false;
}
```

Si la función se invoca en esta pieza de código, su funcionamiento es el correcto:

```
if (esPar(4))
    console.log("El número es par");
else
    console.log("El número es impar");
```

Puesto que el resultado de evaluar la condición de un **if** debe ser un valor booleano, el trabajo se está haciendo correctamente. Sin embargo, véase este otro caso:

```
console.log(esPar(4)+5);
```

La operación se está realizando entre un valor booleano (el valor **true** devuelto por la función) y un número entero (el 5), lo que carece de sentido.

Por otro lado, la definición de la función **esPar** contiene una forma poco recomendada de trabajar con funciones. Al utilizar varios **return** dentro de una función no se obtiene un error, ni siquiera en el caso de que los dos **return** se encuentren uno a continuación del otro. La función termina en cuanto encuentre un **return** y es ese el valor que devuelve a quien la invocó. No obstante, es una práctica que no se recomienda.

En una función cuyo cuerpo son cuatro líneas de código, cualquier programador entenderá qué está ocurriendo, pero cuando el cuerpo de una función tiene cientos de líneas de código con múltiples **return** en distintas partes del mismo, la legibilidad y el seguimiento del flujo de ejecución se convierten en tareas extenuantes. Por todo ello, se recomienda que se incluya un solo **return** en cada función.

De esta manera, la función **esPar** se puede escribir de este modo:

```
function esPar(numero) {
    let resultado = false;
    if (numero%2 == 0)
        resultado = true;
    return resultado;
}
```

Actividad resuelta 4.2

Soluciones de una ecuación de segundo grado

Crea una función que reciba tres parámetros (**a**, **b**, **c**) y devuelva un **array** con las soluciones de una ecuación de segundo grado, tras aplicar la famosa fórmula $(-b \pm \sqrt{b^2 - 4ac}) / 2a$.

Solución

```
function ecuacionGrado2(a,b,c) {
    let soluciones = new Array();
    let parcial = b**2 - 4*a*c;
    if (parcial > 0) {
        soluciones[0] = (-b + Math.sqrt(parcial)) / (2*a);
        soluciones[1] = (-b - Math.sqrt(parcial)) / (2*a);
    }
    return soluciones;
}
```

Actividad propuesta 4.1**Descuentos**

Escribe una función que reciba como parámetros un precio y un porcentaje de descuento y devuelva el precio nuevo.

4.3. Ámbito de variables y tipos de parámetros

Tanto el ámbito de las variables como el de los parámetros de las funciones son dos conceptos clave para sacar el mayor provecho posible a la utilidad que ofrecen las funciones. Es prácticamente imposible leer código JavaScript y entenderlo si no se dominan estas dos características del lenguaje.

4.3.1. Variables locales y globales

Cuando se habla de ámbito de las variables, se está haciendo referencia a aquellas zonas del programa donde una variable es «visible». Si una variable es visible (accesible) solo desde el interior de una función se dice que la variable es local a la función. Si, por el contrario, la variable es accesible desde cualquier parte del programa, se dice que la variable es global.

```
var mensaje = "Fuera de la función";
function mostrarAnuncio() {
    var mensaje = "Dentro de la función";
    console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```

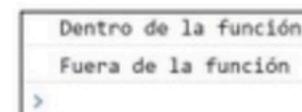


Figura 4.4. Resultado en consola de la ejecución de un programa que muestra el ámbito de las variables.

En este programa hay dos variables que se llaman igual, **mensaje**, pero a pesar de ello realiza su tarea correctamente. ¿Por qué la variable **mensaje** del interior de la función no ha machacado el valor de la variable **mensaje** de la primera línea de código? La respuesta está en el ámbito de las variables. La primera es global, se puede acceder desde cualquier parte del programa. La segunda es local, solo es accesible desde el interior de la función.

Para probar que la primera variable es global, en el siguiente ejemplo se elimina la variable **mensaje** del interior de la función:

```
var mensaje = "Fuerza de la función";
function mostrarAnuncio() {
    console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```

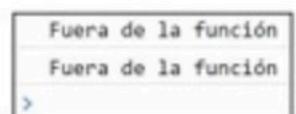


Figura 4.5. Resultado de la ejecución evidenciando la visibilidad de la variable global.

El resultado de la ejecución prueba que desde el interior de la función se puede acceder a la variable definida en su exterior.

En este otro ejemplo se hace lo contrario, se elimina la variable **mensaje** exterior y se deja la interior:

```
function mostrarAnuncio() {
    var mensaje = "Dentro de la función";
    console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```



Figura 4.6. Error lanzado por la consola tras no encontrar la definición de una variable.

El intérprete lanza un error: la variable **mensaje** no está definida. Es decir, en el ámbito de la última instrucción del programa (ámbito global) no existe ninguna variable llamada **mensaje**. La variable **mensaje** que aparece en el código es local a la función y vive únicamente dentro de ella.

Recuerda

No se recomienda la utilización de variables globales en los programas. A menudo son muy problemáticas a la hora de depurarlos. Al ser accesibles desde cualquier punto del programa, las puede modificar cualquier función, por lo que localizar un error causado por una variable global suele convertirse en un auténtico dolor de cabeza.



4.3.2. Parámetros por valor y por referencia

Anteriormente se ha estudiado que los parámetros son esos valores que permiten la comunicación con las funciones. El carácter de los parámetros depende de si su modificación en el interior de la función afecta a su valor en el exterior de la función.

Para empezar este estudio, un primer ejemplo:

```
var numero1 = 7;
var numero2 = 8;
function menor(primer,segundo) {
    var elmenor = primer;
    if (segundo < primer)
        elmenor = segundo
    return elmenor;
}
console.log(menor(numero1,numero2));
```

En este código se crean dos variables en el programa principal llamadas **numero1** y **numero2**. Luego se define una función que calcula el menor de los dos números que recibe en los parámetros **primer** y **segundo**. Finalmente se realiza la invocación a la función pasándole como parámetros **numero1** y **numero2**.

Cuando la ejecución del programa llega a la última línea y se invoca a la función, automáticamente el valor de **numero1** (7) se copia en la variable **primer**, y el valor de **numero2** (8) se copia en la variable **segundo**. Así, la función hace sus cálculos y devuelve el valor 7. La clave de este proceso reside en la expresión «se copia», porque si en el interior de la función se modifica el valor de **primer** o el de **segundo**, las variables globales **numero1** y **numero2** seguirían teniendo sus propios valores iniciales, como se muestra a continuación:

```
var numero1 = 7;
var numero2 = 8;
function menor(primer,segundo) {
    primer = 10;
    segundo = 21;
    var elmenor = primer;
    if (segundo < primer)
        elmenor = segundo
    return elmenor;
}
console.log(menor(numero1,numero2)); // muestra 10
console.log(numero1); // muestra 7
console.log(numero2); // muestra 8
```

Hasta ahora se ha visto el funcionamiento de los parámetros por valor. Los parámetros por referencia serían justo lo contrario. Si los parámetros **primer** y **segundo** se hubieran definido por referencia, las variables **numero1** y **numero2** tendrían los valores 10 y 21 respectivamente al acabar el programa.

Para establecer valores por referencia la mayoría de los lenguajes de programación tienen su propia sintaxis. Unos lenguajes definirían los parámetros como **menor (ref primer, ref segundo)**, otros como **menor (&primer, &segundo)** y otros como **menor (&\$primer, &\$segundo)**, o cualquier otra variante propia de los mismos. En JavaScript, en cambio, no existe esa posibilidad de calificar un parámetro cualquiera por referencia, sino que existen ciertas variables que cuando hacen referencia a un objeto, su propio identificador es ya una referencia.

En el Apartado 3.1.4 se estudió la operación de asignación de arrays. Al asignar un array a otro, no se copiaba el contenido de uno de ellos en el otro, sino que los dos identificadores de las variables apuntaban al mismo array, de manera que al operar con una variable se veía afectado el contenido de la otra. Ambos identificadores eran referencias al mismo array.

Reescribiendo la función anterior puede calcularse el menor de los elementos de un array de enteros positivos. Además, en aquella posición donde se detecte el menor, se cambiará su valor a -1. Esto es lo que ocurre:

```
var vector = [6,2,9,5,3];
function menor(elarray) {
    var elmenor = elarray[0];
    var posicion = 0;
    for (let i=0; i<elarray.length; i++)
        if (elarray[i] < elmenor) {
            elmenor = elarray[i];
            posicion = i;
        }
    elarray[posicion] = -1;
    return elmenor;
}
console.log(menor(vector));
console.log(vector);
```

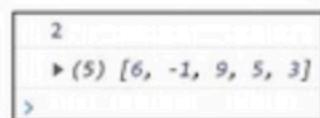


Figura 4.7. Evidencia del paso de arrays por referencia.

El programa ha realizado correctamente el cálculo y ha devuelto el menor de los elementos. También ha colocado -1 en la posición que ocupaba el menor. Pero al mostrar la variable **vector** se ve que su contenido ya no es el que era en la primera línea del programa, [6,2,9,5,3], sino que la modificación realizada por la variable **elarray** en el interior de la función ha afectado al contenido de la variable **vector**. ¿Por qué? Porque los arrays siempre se pasan por referencia.

¿Cuáles son los tipos de datos que se pasan por referencia entonces? Los objetos (entre los que se encuentran los arrays, los conjuntos y los mapas), aunque para entender completamente esto hay que seguir avanzando en el estudio de JavaScript y descubrir otros objetos comunes del lenguaje.

Actividad resuelta 4.3

Limpieza de los mapas

Escribe una función que elimine de un mapa que almacena los códigos postales de localidades españolas todos los códigos postales que no tengan cinco dígitos. Crea el mapa, añade cinco localidades con un código postal erróneo y comprueba que la función realiza su cometido correctamente. Indica qué tipo de parámetro es el que has usado y por qué.

Solución

```
let mapa = new Map();
mapa.set("41700","Dos Hermanas");
mapa.set("21440","Lepe");
mapa.set("11160","Barbate");
mapa.set("1138","Tarifa");
mapa.set("41013","Sevilla");
function checkMap(unMapa) {
    for (let [clave,valor] of unMapa)
        if (clave.length != 5)
            unMapa.delete(clave);
}
checkMap(mapa);
console.info(mapa);
```

El parámetro `unMapa` ha pasado por referencia a la función, puesto que los cambios que se le han hecho en su interior siguen estando vigentes en el exterior.

4.3.3. Parámetros por defecto

En algunas ocasiones, al definir una función se da la circunstancia de usar la función con los mismos parámetros. En otros escenarios se quiere prevenir que al programador se le olvide especificar uno o varios parámetros. Y, por último, en otras situaciones se desea que la llamada a ciertas funciones sea lo más rápida y sintética posible. En todos estos casos descritos se necesita un mecanismo que permita obviar el uso de parámetros y que la función siga haciendo su trabajo correctamente. La solución consiste en utilizar parámetros por defecto (o predeterminados), de manera que en la definición de la función puedan establecerse unos valores predeterminados en aquellos parámetros que interesen. Si en la invocación no se especifican valores para esos parámetros, tomarán los que se asignaron por defecto. Y si se especifican, sustituirán a los valores predeterminados.

En el siguiente ejemplo se define una función que recibe dos parámetros, el segundo de ellos con un valor predeterminado. Así, en la primera invocación se está dividiendo 4

entre 1, en la segunda 4 entre 2 y en la tercera `undefined` entre 1. Esto quiere decir que en JavaScript, en realidad, todos los parámetros tienen establecido un valor predeterminado: `undefined`.

```
function dividir(numerador, denominador=1) {
    return (numerador/denominador);
}
console.log(dividir(4)); // muestra 4
console.log(dividir(4,2)); // muestra 2
console.log(dividir()); // muestra NaN
```

Por otro lado, hay que tener cuidado al establecer los valores predeterminados y no olvidar que la asignación de valores a los parámetros se hace siempre de izquierda a derecha. Si se modifica la función anterior para que el primer parámetro tenga un valor predeterminado y el segundo no, no se puede invocar a la función con un solo parámetro, porque este se asignará al primer parámetro de la función, no al que no tenga valor, que es el segundo:

```
function dividir(numerador=1, denominador) {
    return (numerador/denominador);
}
console.log(dividir(4)); // muestra NaN
console.log(dividir(4,2)); // muestra 2
console.log(dividir()); // muestra NaN
```

En la primera invocación a la función se pasa un parámetro que se copia en `numerador`, sustituyendo a su valor predeterminado, y al no haber indicado un segundo parámetro la operación que se está haciendo es 4 entre `undefined`, o sea `NaN`.

En resumen, para utilizar parámetros predeterminados deben colocarse al final de la lista de parámetros.

Actividad propuesta 4.2

Partida de dados

Escribe una función que reciba como parámetro un número entero (número de rondas de la partida), por defecto establecido a 5. Debes simular una partida de dados con lanzamientos aleatorios. Hay dos jugadores que lanzan dos dados una vez en cada ronda. La suma de los dados se anota y se pasa a la siguiente ronda. Al final de todas las rondas el programa debe proporcionar un ganador y mostrar las puntuaciones acumuladas de ambos. El programa debe funcionar correctamente tras atender estas llamadas:

```
jugar();
jugar(3);
jugar(10);
```

4.3.4. Parámetros variables

Existe otra manera muy útil y flexible de utilizar parámetros, aunque propensa a errores si no se conoce bien el programa, que es utilizar un número variable de ellos.

En esta variante no es necesario especificar en la definición de la función cuántos son ni cómo se llaman los parámetros que utiliza. En lugar de ello, se utiliza un objeto disponible en JavaScript llamado **arguments**, que contiene un *array* que almacena todos los argumentos que se han indicado en la invocación a la función.

```
function sumaTodo() {
    let sum = 0;
    for (let i = 0; i < arguments.length; i++)
        sum += arguments[i];
    return sum;
}
x = sumaTodo(11, 22, 33, 44, 55);
console.log(x);
```

En la definición de esta función no se han especificado parámetros; sin embargo, es capaz de gestionar una invocación con cinco parámetros. Se consigue usando el objeto **arguments** de JavaScript.

La otra opción para trabajar con parámetros variables es utilizar un viejo amigo, el operador de propagación. En este contexto el operador de propagación convierte la lista de parámetros usados en la invocación a una función en un *array* donde cada posición almacena un parámetro:

```
function sumaTodo(...parametros) {
    let sum = 0;
    for (let i = 0; i < parametros.length; i++)
        sum += parametros[i];
    return sum;
}
x = sumaTodo(11, 22, 33, 44, 55);
console.log(x);
```

Actividad propuesta 4.3

Adivina la palabra

Escribe una función que, a partir de una palabra establecida por el programador, reciba como parámetros un número variable de letras y muestre en pantalla solo las coincidencias entre la palabra secreta y las letras proporcionadas.

Por ejemplo, si la palabra secreta establecida es «sargento» y se ha invocado a la función como `adivina('a','e','i','o','u','d','n')`, la salida del programa debe ser `-a--en-o`.

4.4. Tipos de funciones

Esta sección podría comenzar diciendo «bienvenidos al corazón de JavaScript», porque en las próximas líneas se estudiarán todas las formas que existen de definir y utilizar funciones, auténtico motor de la lógica de los programas. Tras interiorizar esta sección, se habrá dado un paso de gigante para poder leer y entender cualquier pieza de código JavaScript.

4.4.1. Funciones por declaración

Estas son las funciones que se han visto a lo largo de esta unidad. Esta forma de definir funciones es la más común, la más sencilla de entender y la que más se parece a la forma de definir funciones en todos los lenguajes de programación.

Las funciones definidas por declaración existen y están disponibles a lo largo de todo el código del programa. Además, pueden invocarse incluso antes de ser definidas, puesto que JavaScript «peina» el código para buscar sus definiciones y luego ejecuta todo el código secuencialmente.

```
let resultado = multiplicar(7,5);
function multiplicar(a,b) {
    return a*b;
}
console.log(resultado); // muestra 35
```

4.4.2. Funciones por expresión

En este tipo de funciones lo que se pretende es relacionar la definición de una función con el identificador de una variable. Por así decirlo, es como almacenar una función en una variable, de manera que al utilizar esa variable lo que se hace es invocar a su función almacenada.

```
const bienvenido = function sesionIniciada() {
    console.log("Bienvenido de nuevo");
};
bienvenido();
sesionIniciada();
```

Al observar la ejecución del código anterior se aprecia que, en realidad, el identificador de la función (`sesionIniciada`) carece completamente de utilidad, puesto que ahora para acceder a la función hay que hacerlo usando el identificador de la variable `bienvenido`.

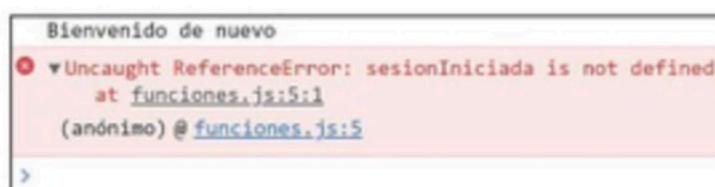


Figura 4.8. Resultado en consola donde se evidencia que el nombre de la función no se puede referenciar directamente.

4.4.3. Funciones como objetos

Se ha incluido esta variante, muy poco utilizada, solo para ilustrar hasta qué punto en JavaScript todo es un objeto (en la próxima unidad se estudiará todo lo relacionado con ellos).

```
const bienvenido = new Function("console.log('Bienvenido de nuevo');");
bienvenido();
```

4.4.4. Funciones anónimas

También llamadas **funciones lambda**. Se llaman anónimas porque no tienen un identificador de función asociado. Se ha visto recientemente, repasando las funciones por expresión, que realmente el identificador de la función, a pesar de estar, no tiene utilidad. ¿Por qué indicarlo entonces? Al retirarlo la función ya es anónima:

```
const bienvenido = function () {
    console.log("Bienvenido de nuevo");
};

bienvenido();
```

Además, es importante destacar que existe una diferencia significativa entre este tipo de funciones y las funciones declarativas. En el caso de las funciones anónimas, como se ha ligado su uso a una variable, hasta que la variable no esté inicializada no se puede utilizar, como el resto de las variables. Es preciso recordar que esto no ocurría con las funciones declarativas, que pueden invocarse en el código antes de que estén definidas.

Actividad propuesta 4.4

Cadenas invertidas

Crea una función anónima vinculada a una variable llamada **invertida** que reciba una cadena de texto y la devuelva invertida (transformada de derecha a izquierda).

Por ejemplo, si la cadena recibida es «almadraba» debe devolver «abardamla».

4.4.5. Callbacks

Una **callback** es una función anónima que puede pasarse como parámetro a otra función. La estrategia es definir una función que, al ser anónima, se vincula a una variable, y luego se utiliza esa variable como parámetro de otra función, de manera que esta última función pueda ejecutar el contenido de la primera.

```
const bienvenido = function () {
    return "Bienvenido de nuevo, ";
};

const usuario = function (callback) {
    console.log(callback() + "Javier");
};

usuario(bienvenido); // muestra "Bienvenido de nuevo, Javier"
```

En muchas ocasiones, como en el ejemplo que se está viendo, ni siquiera sale a cuenta vincular una función anónima a una variable, sino que directamente en la invocación a la función se utilizan los parámetros para definir la función anónima:

```
const usuario = function (callback) {
    console.log(callback() + "Javier");
};

usuario(
    function () {
```

```
        return "Bienvenido de nuevo, ";
    });
}
```

Sin embargo, para funciones más extensas siempre es recomendable utilizar la primera forma de programar *callbacks* que se ha visto, ya que mejora mucho la legibilidad y el seguimiento del código.

4.4.6. Funciones autoejecutables

Podría tener sentido también, en ciertos escenarios, ejecutar una función inmediatamente después de crearla, por ejemplo, en cuanto el intérprete de JavaScript esté disponible, tras la carga de una web. Para eso están las funciones autoejecutables. La idea es incluir entre paréntesis una función sin identificador que será invocada tan pronto como el intérprete pase por su definición:

```
(function () {
    console.log("Bienvenido de nuevo");
})();
```

Es conveniente observar cómo termina esta definición. Efectivamente, no hay ningún motivo por el que no se pueda pasar parámetros a esta función:

```
(function (usuario) {
    console.log("Bienvenido de nuevo, "+usuario);
})("Javier");
```

Por último, hay que tener cuidado y entender qué ocurre al asignar una función autoejecutable a una variable. La función sí que se va a ejecutar de forma automática se utilice o no la variable, a diferencia de las funciones anónimas, en las que la función solo se ejecuta cuando se usa la variable.

```
const variable = (function (usuario) {
    return ("Bienvenido de nuevo, "+usuario);
})("Javier");
console.log(variable);
```

Actividad propuesta 4.5

Sello temporal

Crea una función autoejecutable que informe por consola de la fecha y la hora a la que se inició la ejecución de la función.

Nota: investiga el uso del objeto **Date** para facilitar el formato de los datos de salida.

4.4.7. Clausuras

Este concepto antintuitivo de las funciones cuesta un poco más de entender. Siempre se ha dicho que las variables locales a una función dejan de estar accesibles tan pronto como finaliza la ejecución de la función. Hasta ahora.

Como se decía anteriormente y tomando como referencia el siguiente código, y como se entendían las funciones, una vez que la función **mensaje** haya terminado de ejecutarse se supone que ya no se podrá acceder a la variable **texto**. Sin embargo, al observar la salida de la ejecución de este programa se ve que esto no es así (Figura 4.9).

```
function mensaje() {
    let texto = "Hola de nuevo.";
    function muestraMensaje() {
        console.log(texto);
    }
    return muestraMensaje;
}
let variable = mensaje();
variable();
```

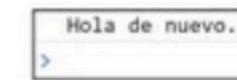


Figura 4.9. Resultado novedoso tras el estudio del ámbito de las variables.

Esto sucede porque **variable** se ha convertido en una clausura, que incluye tanto una función como el contexto en el que se creó esa función. El contexto está formado por las variables locales que estaban en el ámbito de la función cuando se creó la clausura.

Quizás el lector no sea capaz todavía de entender cuál es la utilidad de una lógica tan enrevesada, pero lo hará en cuanto estudie los objetos, los eventos y la combinación de estos.

4.4.8. Funciones flecha

Las funciones flecha, por su parte, son simplemente una forma simplificada de definir funciones, de manera que puede eliminarse la palabra **function** y usar **=>** en su lugar, de ahí su nombre.

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

Pero las ventajas de la definición de funciones flecha no queda solo en su capacidad de síntesis, sino que aporta algunas otras, como las siguientes:

- Las llaves se pueden omitir si el cuerpo de la función tiene una sola línea, como en este caso:


```
const mensaje = () => console.log ("Hola de nuevo.");
```
- En el caso anterior, si la función tuviera un **return**, también se podría obviar porque se haría de forma automática.
- En el caso de que la función tuviera un solo parámetro, también se podría prescindir de los paréntesis iniciales:


```
const mensaje = nombre => console.log("Hola de nuevo, " + nombre);
mensaje("Javier");
```

- Aumentan considerablemente la legibilidad del código.
- Mejora la productividad de los programadores.

Actividad propuesta 4.6

Mayor cadena

Escribe una función flecha que reciba dos cadenas de caracteres e informe por consola de aquella que contiene más vocales.

4.5. Recursividad

La recursividad es la capacidad que tienen las funciones de llamarse a sí mismas. Es una técnica de programación peligrosa (puede generar llamadas infinitas y desbordar la pila del sistema), pero está considerada una forma muy elegante de resolver problemas. A veces, se presentan problemas complejos de resolver y tras darles un enfoque recursivo aparecen soluciones extraordinariamente simples.

A modo de ejemplo, véase la sucesión de Fibonacci, una serie numérica en la que cada elemento es la suma de los dos anteriores: 1,1,2,3,5,8,13,21...). ¿Cómo se resolvería el problema de tener que programar un algoritmo que muestre los *n* primeros elementos de la sucesión de Fibonacci? Antes de continuar, es recomendable tomarse un tiempo e intentar resolverlo.

```
function fibo(numero) {
    if (numero < 2)
        return 1;
    else
        return fibo(numero-1) + fibo(numero-2);
}
let elementos = 10;
for (let i=0; i<elementos; i++)
    console.log(fibo(i));
```

Como se puede apreciar, se ha creado una función que calcula cada uno de los elementos de la sucesión y luego con un bucle se muestran en pantalla.

Lo realmente interesante de la recursividad es que no es un proceso en el que se vaya completamente a ciegas, sino que existe una estrategia básica que incorporan todos los algoritmos que usan este recurso: **caso base + caso recursivo**.

Siempre que se plantee un problema de este tipo debe buscarse el caso base, es decir, aquel caso en el que no es necesario seguir invocando a la función, el caso en el que la recursividad se detiene. En este ejemplo, la recursividad debe parar al estar en el elemento 0 o 1 (**numero < 2**), puesto que sería el inicio de la sucesión. Todos los demás casos necesitarán invocar a la función puesto que cada elemento de Fibonacci es la suma de los dos anteriores [**fibo (numero-1) + fibo (numero-2)**].

Véase cómo se ha calculado el quinto elemento de la serie [fib(4)], porque solo viendo la sucesión de llamadas a la función puede entenderse qué ha ocurrido:

```

fib(4)
fib(3)+fib(2)
(fib(2)+fib(1))+(fib(1)+fib(0))
((fib(1)+fib(0))+fib(1))+(fib(1)+fib(0))
((fib(1)+fib(0))+fib(1))+(fib(1)+1)
((fib(1)+fib(0))+fib(1))+(1+1)
((fib(1)+fib(0))+1)+(1+1)
((fib(1)+1)+1)+(1+1)
((1+1)+1)+(1+1)
((1+1)+1)+2
((2)+1)+2
3+2
5

```

Quizás quede más claro planteando el problema de calcular el factorial de un número n (producto de todos los números anteriores). El factorial de 5, expresado como $5!$, sería $5*4*3*2*1$. Es decir, $n*(n-1)$, tantas veces como sea necesario hasta llegar a 1.

Siguiendo el razonamiento anterior, la recursividad debe parar cuando n valga 1, pues ese es el caso base. En el resto de los casos, debe ser $n*$ el resultado de llamar otra vez a la función con $n-1$.

```

function factorial(n) {
    if (n<=1)
        return 1;
    else
        return n*factorial(n-1);
}
console.log(factorial(4)); // muestra 24

```

Volviendo a hacer la sucesión de invocaciones, se observa lo siguiente:

```

factorial(4)
4 * factorial(3)
4 * 3 * factorial(2)
4 * 3 * 2 * factorial(1)
4 * 3 * 2 * 1
4 * 3 * 2
4 * 6
24

```

Importante

En general, las soluciones recursivas son mucho más costosas computacionalmente que las soluciones iterativas, por lo que se recomienda que se recurra a la recursividad solo en aquellos casos en los que la solución iterativa no sea posible.

Actividad resuelta 4.4

¿Es palíndroma?

Escribe una función recursiva que indique si la palabra que le pasas como parámetro es palíndroma (devuelve true) o no (devuelve false).

Nota: una palabra palíndroma es aquella que se lee igual de izquierda a derecha que de derecha a izquierda, como «SOS», «rajar» u «orejero».

Solución

```

function palindroma(cadena) {
    if (cadena.length <= 1) {
        return true;
    }
    if (cadena[0] !== cadena[cadena.length - 1]) {
        return false;
    }
    else {
        return palindroma(cadena.substring(1,cadena.length-1));
    }
}
console.log(palindroma("orejero")); // devuelve true

```

4.6. Más funcionalidad para las estructuras de datos

En la unidad anterior, al estudiar las estructuras de datos, quedaron pendientes algunos conceptos que mejoran su funcionalidad, porque era preciso conocer antes y en profundidad las funciones. Ha llegado el momento de abordar su estudio.

4.6.1. Ordenación avanzada de arrays

El siguiente ejemplo es la pieza de código con la que se estudió la ordenación de un array de cadenas de caracteres.

```

let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort();
// ordenación esperada -> ['casa', 'Casado', 'ñam', prueba', 'zancos']
// ordenación obtenida -> ['Casado', 'casa', 'prueba', 'zancos', 'ñam']

```

El problema, como se aprecia en los comentarios del código anterior, es que la ordenación se realiza considerando la posición que ocupa cada carácter en la tabla Unicode.

Pero la función **sort** deja abierta la posibilidad de incluir una **callback** que permita establecer un criterio de ordenación personalizado. Esta función debe recibir dos parámetros, de manera que la función devuelva:

- < 0: se sitúa el primero en un índice menor que el segundo, es decir, aparece antes.
- = 0: no se realizan cambios entre ellos.
- > 0: se sitúa el segundo en un índice menor que el primero, es decir, aparece antes.

La primera mejora que se puede añadir es situar las palabras más cortas al principio del array:

```

let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort((primera,segunda)=>primera.length-segunda.length);
// Ordenación obtenida -> ['ñam', 'casa', 'Casado', 'prueba', 'zancos']

```

Una vez hecho esto (por la simplicidad de la `callback` se opta por utilizar la notación flecha), se aplica la ordenación estricta del español, es decir, situar a la «ñ» entre la «n» y la «o», y también obviar las mayúsculas y minúsculas. Para ello se puede utilizar una función destinada a la comparación de cadenas de caracteres de un idioma determinado que se pasa como parámetro, `localeCompare()`:

```
let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort((primera,segunda)=>primera.length-segunda.length);
vector.sort((primera,segunda)=>primera.localeCompare(segunda,"es"));
// Ordenación obtenida -> ['casa', 'Casado', 'ñam', 'prueba', 'zancos']
```

Como se ve, el conocimiento de las funciones aporta en solo dos líneas una ordenación completamente personalizada, que de otro modo habría necesitado docenas de líneas.

Actividad resuelta 4.5

Pares primero, pero en orden

Escribe una función que dado un array de enteros positivos lo ordene, pero además sitúe a los pares al principio del array. Ambos grupos deben estar también ordenados.

Solución

Para abordar la solución propuesta, hemos realizado una primera ordenación por defecto para colocar todos los elementos de menor a mayor. Después, con una segunda ordenación, hemos desplazado los pares al inicio del array. Y, finalmente, con la tercera ordenación, hemos ordenado los pares, porque los impares ya aparecen ordenados.

```
let vector = [5,7,3,1,4,9,2,6,8];
vector.sort();
vector.sort(
    (primero,segundo)=>{
        if (primero%2 == 0)
            return -1;
        else
            return 0;
    }
);
vector.sort(
    (primero,segundo)=>{
        if ((primero%2 == 0) && (segundo>primero))
            return -1;
        else
            return 0;
    }
);
console.info(vector);
```

4.6.2. Recorridos elegantes con `forEach`

Se trata de la última variante de `for` que quedó pendiente de estudio tras repasar los recorridos de arrays, conjuntos y mapas.

`forEach` permite configurar un recorrido usando una función que recibe dos parámetros, el primero almacenará automáticamente cada elemento y el segundo (opcional) su índice.

Además, aplica a los elementos `undefined` el mismo tratamiento que `for..in`, es decir, no los tiene en cuenta.

■ Arrays:

```
let vector = [12,334,111,52,98];
vector.forEach(function (elemento, posicion){
    console.log(`Posición ${posicion}: ${elemento}`);
});
```

Posición 0: 12
Posición 1: 334
Posición 2: 111
Posición 3: 52
Posición 4: 98

Figura 4.10. Salida de `forEach` tras recorrer un array.

■ Conjuntos:

```
let conjunto = new Set();
conjunto.add(12).add(334).add(111).add(52).add(98);
conjunto.forEach(function (elemento){
    console.log(`Elemento: ${elemento}`);
});
```

Elemento: 12
Elemento: 334
Elemento: 111
Elemento: 52
Elemento: 98

Figura 4.11. Salida de `forEach` tras recorrer un conjunto.

■ Mapas:

```
let mapa = new Map();
mapa.set('a',12).set('b',334).set('c',111).set('d',52).set('e',98);
mapa.forEach(function (valor,clave){
    console.log(`Clave: ${clave} / Valor: ${valor}`);
});
```

Clave: a / Valor: 12
Clave: b / Valor: 334
Clave: c / Valor: 111
Clave: d / Valor: 52
Clave: e / Valor: 98

Figura 4.12. Salida de `forEach` tras recorrer un mapa.

4.6.3. Recorridos avanzados de arrays

Los dos métodos que se verán a continuación para recorrer arrays son algunas de esas funcionalidades que a veces se olvidan y que pueden ahorrar mucho tiempo de desarrollo.

Map

Es una función, establecida mediante una *callback* con un parámetro, que no modifica el contenido del array y que devuelve una copia con los cambios aplicados por la función.

En el siguiente ejemplo se ve cómo puede ayudar para obtener un array de precios con IVA a partir de otro sin IVA:

```
let sinIVA = [12.45, 34.42, 99.90, 49.95];
let conIVA = sinIVA.map(x=>(x*1.21).toFixed(2));
```

La función *toFixed()* se ha usado simplemente para formatear el resultado de las operaciones a dos decimales.

```
> (4) [12.45, 34.42, 99.9, 49.95]
> (4) ['15.06', '41.65', '120.88', '60.44']
```

Figura 4.13. Resultado del recorrido de un array con map.

Filter

Es una función muy utilizada. Al igual que la anterior, recibe una *callback* con un parámetro que va recogiendo el valor de cada elemento del array. En cada iteración comprueba si el elemento cumple con una condición específica. Finalmente devuelve un array con aquellos elementos que han cumplido la condición.

```
let playas = ["Hierbabuena", "Caños", "Zahara", "Carmen", "Palmar"];
let filtradas = playas.filter(elemento=>elemento.length!=6);
```

De esta forma, se crea un array con aquellos elementos cuya longitud es mayor o menor de seis caracteres.

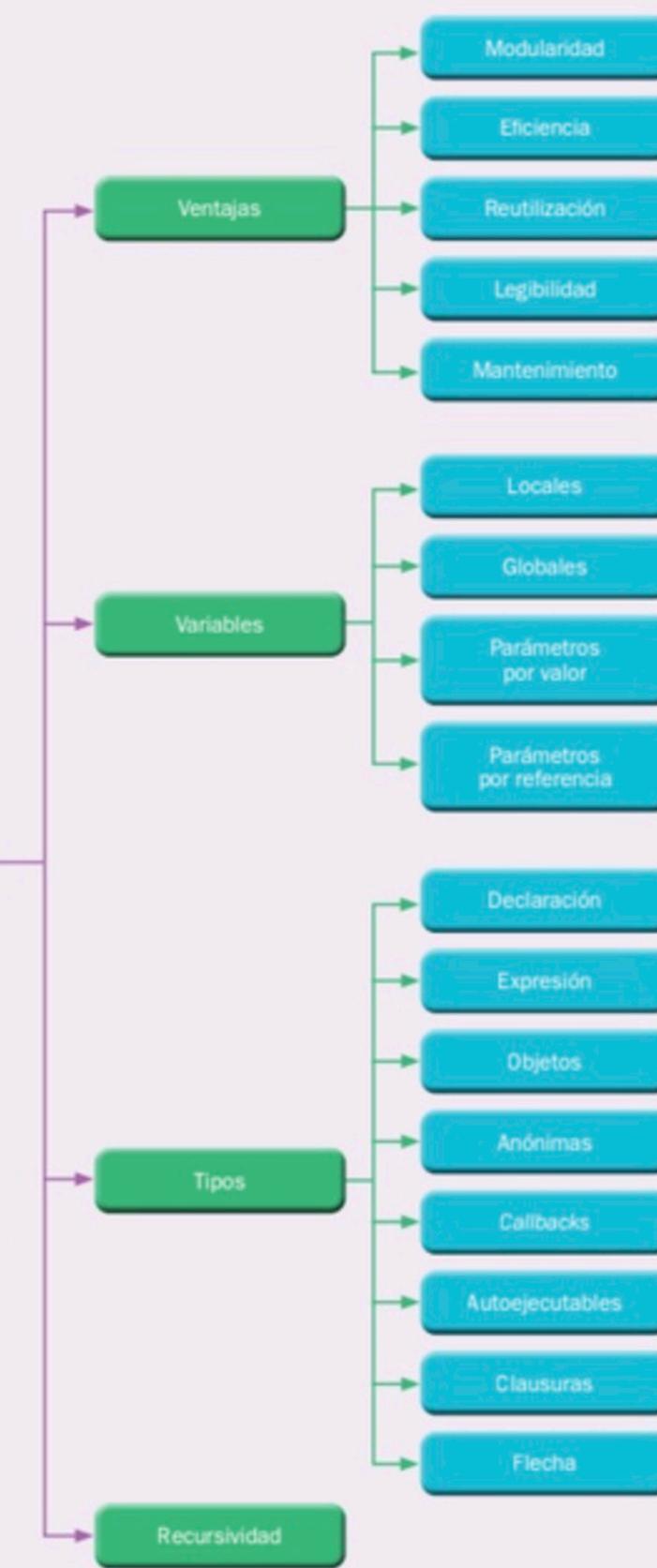
```
> (5) ['Hierbabuena', 'Caños', 'Zahara', 'Carmen', 'Palmar']
> (2) ['Hierbabuena', 'Caños']
```

Figura 4.14. Resultado del filtrado de un array con filter.

Actividad propuesta 4.7

Sin tildes

Escribe una función que haciendo uso de *filter* genere un array a partir de los elementos de otro array que contiene cadenas de caracteres, sin los elementos que contengan tildes.



Actividades de comprobación

- 4.1.** ¿Cómo se llama la estrategia de resolver problemas en subproblemas más pequeños?
- División.
 - Modularidad.
 - Distribución.
 - Desacoplamiento.
- 4.2.** ¿Cómo se llama el nombre que los programadores asignan a una función?
- Label.
 - Etiqueta.
 - Identificador.
 - Código.
- 4.3.** ¿Qué nombre reciben los datos que se pasan a una función desde el exterior?
- Señales.
 - Mensajes.
 - Variables.
 - Parámetros.
- 4.4.** Al definir una función, el valor de retorno:
- Es obligatorio.
 - Es opcional.
 - Da igual si se pone o no.
 - Está prohibido.
- 4.5.** Cuando se invoca a una función que se ha definido sin parámetros:
- Es necesario igualmente escribir los paréntesis.
 - No es necesario escribir los paréntesis.
 - No puede haber funciones definidas sin parámetros.
 - Es una función inútil.
- 4.6.** ¿Cómo se llama el concepto que se define como «aquellos lugares donde son visibles las variables»?
- Rango.
 - Sección.
 - Ámbito.
 - Área.
- 4.7.** Las variables locales a las funciones:
- Son accesibles desde cualquier parte del programa.
 - Son accesibles solo desde fuera de las funciones.
 - Son accesibles solo en el interior de las funciones.
 - No son accesibles desde ninguna parte del código.

- 4.8.** ¿Qué nombre reciben los parámetros que, tras ser modificados en el interior de una función, permanecen modificados fuera de la función?
- Parámetros locales.
 - Parámetros globales.
 - Parámetros por valor.
 - Parámetros por referencia.
- 4.9.** Si en la definición de una función tuvieras que establecer tres parámetros y dos de ellos tuvieran valores predeterminados, ¿dónde los situarías en la lista de parámetros?
- Ambos al principio.
 - Ambos al final.
 - Uno al principio y otro al final.
 - Solo se permite un parámetro por defecto en cada función.
- 4.10.** ¿Cuáles son las estrategias para gestionar parámetros variables?
- Arrays y conjuntos.
 - Mapas y vectores.
 - Objeto **arguments** y operador de propagación.
 - Objeto **string** y operador de concatenación.
- 4.11.** ¿De qué tipo es esta función?
- ```
const bienvenido = function () {
 console.log("Bienvenido de nuevo");
};
```
- Declarativa.
  - Flecha.
  - Anónima.
  - Callback.
- 4.12.** Si una función autoejecutable se asigna a una variable, ¿cuándo se ejecuta?
- En cuanto se use la variable.
  - En cuando el intérprete lea la variable.
  - En cuanto se use el identificador de la función.
  - En cuanto el intérprete lea la función.
- 4.13.** En una clausura, las variables locales a la función:
- Dejan de ser accesibles tan pronto como finaliza la ejecución de la función.
  - Son accesibles incluso después de finalizar la ejecución de la función.
  - Solo son accesibles antes de que comience la ejecución de la función.
  - No existen.
- 4.14.** Cuando el cuerpo de una función flecha está formado por una sola instrucción:
- Puedes omitir los paréntesis de los parámetros.
  - Puedes omitir el nombre de la función.
  - Puedes omitir **return**.
  - Puedes omitir la flecha, =>.

**4.15. De forma general se recomienda escribir funciones recursivas:**

- a) Siempre.
- b) Nunca.
- c) Cuando se puede resolver con métodos iterativos.
- d) Cuando no se puede resolver con métodos iterativos.

**4.16. En una función recursiva, ¿qué representa el caso base?**

- a) El caso en el que la recursividad para.
- b) El caso en el que la recursividad continúa con más llamadas a la función.
- c) El caso en el que se devuelven los resultados.
- d) El caso en el que se desborda la pila del sistema.

**4.17. ¿Qué tipo de funciones usa el método forEach?**

- a) Autoejecutables.
- b) Recursivas.
- c) Anónimas.
- d) Declarativas.

**4.18. ¿Qué es una callback?**

- a) Una función que forma parte de la API de llamadas (perdidas) en los smartphones.
- b) Una función que se llama siempre la última.
- c) Una función anónima que puedes pasar como parámetro a otra función.
- d) Una función autoejecutable que no espera.

**4.19. ¿Qué ocurre si invocas a una función por expresión usando su identificador?**

- a) Nada, se ejecuta con normalidad.
- b) Las funciones por expresión no tienen identificador.
- c) Generará un error.
- d) Devolverá **true** si la función existe o **false** en caso contrario.

**4.20. ¿Qué significa ...x como parámetro?**

- a) x es el nombre del cuarto parámetro.
- b) x es un array donde cada posición almacena un parámetro.
- c) Cada punto significa que es un parámetro sin nombre y el cuarto lleva por nombre x.
- d) Nada, es una expresión mal escrita que generará un error.

## Actividades de aplicación

**4.21.** Crea una función que devuelva como grados Celsius la cantidad en grados Fahrenheit que recibe como parámetro.

**4.22.** Escribe una función que indique si un número que recibe como parámetro es múltiplo de 10.

**4.23.** Programa una función que determine si un año que recibe como parámetro es bisiesto.

**4.24.** Crea una función que muestre la tabla de multiplicar del número que recibe como parámetro.

**4.25.** Crea una función que devuelva la letra del DNI que recibe como parámetro.

**4.26.** Escribe una función que reciba como parámetro un array de 3x3 y lo devuelva modificado con todos sus elementos a 0 excepto la diagonal principal.

**4.27.** Crea una función autoejecutable que muestre el mensaje «Comenzando...» y tres segundos después escriba «Finalizado.».

**4.28.** Programa una función que reciba como parámetros dos arrays de 4x4 y devuelva un tercer array lleno de ceros excepto en aquellas posiciones en las que los dos primeros arrays tienen valores iguales.

**4.29.** Crea una función que reciba un array de 10 elementos, los rellene con números aleatorios entre 1 y 100 y los ordene, de manera que aparezcan primero aquellos que terminan en 0.

**4.30.** Escribe una función que diga si un número que recibe como parámetro es primo o no.

**4.31.** Crea una función que reciba un número variable de parámetros numéricos (al menos cuatro) y devuelva su suma, su media aritmética, la multiplicación del primero con el último, y la división del segundo con el penúltimo.

**4.32.** Escribe una función que devuelva **true** si dos palabras que recibe como parámetros contienen las mismas letras, aunque se encuentren en posiciones distintas.

**4.33.** Programa una función que reciba un array de cadenas de caracteres y que por medio del método **filter** devuelva aquellos caracteres que no forman parte del alfabeto español. Prueba la función incluyendo palabras con caracteres como ^, \$ o &.

**4.34.** Escribe un programa que con la ayuda de una función recursiva muestre en la consola una variante de la sucesión de Fibonacci en la que cada elemento sea la suma de los tres anteriores.

**4.35.** Escribe una función que lance dos dados tantas veces como indique un parámetro y devuelva el lanzamiento que ha obtenido la puntuación ganadora.

**4.36.** Crea un programa que simule el funcionamiento de un bingo. El número de cartones que participan será siempre de cinco. Cada cartón tendrá 8 filas y 4 columnas con 20 números del 1 al 90 distribuidos de forma aleatoria (máximo de cinco por fila). El programa irá sacando bolas y los cartones se irán comprobando en tiempo real. El programa termina cuando un cartón alcanza el bingo. En ese momento se debe mostrar en pantalla cuál de los cinco cartones es el ganador y los números que han salido para comprobar que el bingo es correcto. Plantea el problema, analízalo con detenimiento y diseña previamente todas las funciones que vas a necesitar para resolverlo. Luego, implementa tu solución.