¿Qué es una función?

Fuente: lenguajejs.com

Las **funciones** nos permiten agrupar líneas de código en tareas con un nombre, para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea. Para usar funciones hay que hacer 2 cosas:

- **Declarar la función**: Preparar la función, darle un nombre y decirle las tareas que realizará.
- **Ejecutar la función**: «Llamar» a la función para que realice las tareas de su contenido.

Declaración

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}
```

El contenido de la función es una línea que mostrará por consola un saludo. Sin embargo, si escribimos estas 4-5 líneas de código en nuestro programa, no mostrará nada por pantalla. Esto ocurre así porque solo hemos declarado la función (*le hemos dicho que existe*), pero aún nos falta el segundo paso, **ejecutarla**, que es realmente cuando se realizan las tareas de su contenido.

Veamos, ahora sí, el ejemplo completo con declaración y ejecución:

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}

// Ejecución de la función
saludar();
```

En este ejemplo hemos **declarado la función** y además, hemos **ejecutado** la función (*en la última línea*) llamándola por su nombre y seguida de ambos paréntesis, que nos indican que es una función. En este ejemplo, si se nos mostraría en la consola Javascript el mensaje de saludo.

Ejemplo

Veamos un primer ejemplo que muestre en la consola Javascript la **tabla de multiplicar del 1**:

```
function tablaDelUno() {
  for (let i = 0; i <= 10; i++)
        console.log("1 x", i, "=", 1 * i);
}

// Bucle que ejecuta 3 veces la función tablaDelUno()
for (let i = 0; i < 3; i++){
        tablaDelUno();
}
```

¿Qué son los parámetros?

Pero las funciones no sirven sólo para esto. Tienen mucha más flexibilidad de la que hemos visto hasta ahora. A las funciones se les pueden pasar **parámetros**, que no son más que variables que existirán sólo dentro de dicha función, con el valor pasado desde la ejecución.

Veamos el siguiente ejemplo, utilizando el parámetro **hasta**:

```
// Declaración
function tablaDelUno(hasta) {
        for (let i = 0; i <= hasta; i++) {
                console.log("1 x", i, "=", 1 * i);
        }
}

// Ejecución
tablaDelUno(10);
tablaDelUno(5);
```

Como podemos ver, en el interior de los paréntesis de la función se ha indicado una variable llamada **hasta**. Esa variable contiene el valor que se le da a la hora de ejecutar la función, que en este ejemplo, si nos fijamos bien, se ejecuta dos veces: una con valor **10** y otra con valor **5**.

Si analizamos el código de la declaración de la función, vemos que utilizamos la variable **hasta** en la condición del bucle, para que el bucle llegue hasta ese número (*de ahí su nombre*). Por lo tanto, en la primera ejecución se nos mostrará la tabla de multiplicar del 1 hasta llegar al **1 x 10** y en la segunda ejecución se nos mostrará la tabla de multiplicar del 1 hasta llegar al **1 x 5**.

> La idea de las funciones es enfocarnos en el código de la declaración, y una vez lo tengamos funcionando, nos podemos olvidar de él porque está **encapsulado** dentro de la función. Simplemente tendremos que recordar el nombre de la función y los parámetros que hay que pasarle. Esto hace que sea mucho más fácil trabajar con el código.

Parámetros múltiples

Hasta ahora sólo hemos creado una función con **1 parámetro**, pero una función de Javascript puede tener muchos más parámetros. Vamos a crear otro ejemplo, mucho más útil donde convertimos nuestra función en algo más práctico y útil:

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
  for (let i = 0; i <= hasta; i++)
        console.log(tabla, "x", i, "=", tabla * i);
}

// Ejecución
tablaMultiplicar(1, 10); // Tabla del 1
tablaMultiplicar(5, 10); // Tabla del 5
```

En este ejemplo, hemos modificado nuestra función **tablaDelUno()** por esta nueva versión que hemos cambiado de nombre a **tablaMultiplicar()**. Esta función necesita que le pasemos dos parámetros: **tabla** (*la tabla de multiplicar en cuestión*) y **hasta** (*el número hasta donde llegará la tabla de multiplicar*).

> Podemos añadir **más parámetros** a la función según nuestras necesidades. Es importante recordar que el orden de los parámetros es importante y que los nombres de cada parámetro no se pueden repetir en una misma función.

Parámetros por defecto

Es posible que en algunos casos queramos que ciertos parámetros tengan un valor sin necesidad de escribirlos en la ejecución. Es lo que se llama un **valor por defecto**.

En nuestro ejemplo anterior, nos podría interesar que la tabla de multiplicar llegue siempre hasta el 10, ya que es el comportamiento por defecto. Si queremos que llegue hasta otro número, lo indicamos explícitamente, pero si lo omitimos, queremos que llegue hasta 10. Esto se haría de la siguiente forma:

```
function tablaMultiplicar(tabla, hasta = 10) {
	for (let i = 0; i <= hasta; i++) {
		console.log(tabla, "x", i, "=", tabla * i);
	}
}

// Ejecución
tablaMultiplicar(2); // Esta tabla llegará hasta el número 10
tablaMultiplicar(2, 15); // Esta tabla llegará hasta el número 15
```

> Hay que remarcar que esta característica se añade en **ECMAScript 6**, por lo que en navegadores sin soporte podría no funcionar correctamente.

Devolución de valores

Hasta ahora hemos utilizado funciones simples que realizan acciones o tareas (*en nuestro caso, mostrar por consola*), pero habitualmente, lo que buscamos es que esa función realice una tarea y nos devuelva la información al exterior de la función, para así utilizarla o guardarla en una variable, que utilizaremos posteriormente para nuestros objetivos.

Para ello, se utiliza la palabra clave **return**, que suele colocarse al final de la función, ya que con dicha devolución terminamos la ejecución de la función (*si existe código después, nunca será ejecutado*).

Veamos un ejemplo con una operación muy sencilla, para verlo claramente:

```
// Declaración
function sumar(a, b) {
  return a + b; // Devolvemos la suma de a y b al exterior de la función
  console.log("Ya he realizado la suma."); // Este código nunca se ejecutará
}

// Ejecución
let resultado = sumar(5, 5); // Se guarda 10 en la variable resultado
```

Como podemos ver, esto nos permite crear funciones más modulares y reutilizables que podremos utilizar en multitud de casos, ya que la información se puede enviar al exterior de la función y utilizarla junto a otras funciones o para otros objetivos.

Una vez conocemos las bases de las funciones que hemos explicado en el tema de introducción funciones básicas, podemos continuar avanzando dentro del apartado de las funciones. En Javascript, las **funciones** son uno de los tipos de datos más importantes, ya que estamos continuamente utilizándolas a lo largo de nuestro código.

Y no, no me he equivocado ni he escrito mal el texto anterior; a continuación veremos que las funciones también pueden ser tipos de datos:

```
typeof function () {}; // 'function'
```

Creación de funciones

Hay varias formas de crear funciones en Javascript: por **declaración** (*la más usada por principiantes*), por **expresión** (*la más habitual en programadores con experiencia*) o mediante constructor de **objeto** (*no recomendada*):

Constructor	Descripción
`function nombre(p1, p2...) { }`	Crea una función mediante **declaración**.
`let nombre = function(p1, p2...) { }`	Crea una función mediante **expresión**.
`new Function(p1, p2..., code);`	Crea una función mediante un constructor de **objeto**.

Funciones por declaración

Probablemente, la forma más popular de estas tres, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la **creación de funciones por declaración**. Esta forma permite declarar una función que existirá a lo largo de todo el código:

```
function saludar() {
  return "Hola";
}

saludar(); // 'Hola'
typeof saludar; // 'function'
```

De hecho, podríamos ejecutar la función `saludar()` incluso antes de haberla creado y funcionaría correctamente, ya que Javascript primero busca las declaraciones de funciones y luego procesa el resto del código, este concepto se conoce como **hoisting**.

Funciones por expresión

Sin embargo, en Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por **expresión**, que fundamentalmente, hacen lo mismo con algunas diferencias:

```
// El segundo "saludar" (nombre de la función) se suele omitir: es redundante
const saludo = function saludar() {
  return "Hola";
};

saludo(); // 'Hola'
```

Con este nuevo enfoque, estamos creando una función **en el interior de una variable**, lo que nos permitirá posteriormente ejecutar la variable (*como si fuera una función*). Observa que el nombre de la función (*en este ejemplo: saludar*) pasa a ser inútil, ya que si intentamos ejecutar **saludar()** nos dirá que no existe y si intentamos ejecutar **saludo()** funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las **funciones anónimas** (*o funciones lambda*).

Funciones como objetos

Como curiosidad, debes saber que se pueden declarar funciones como si fueran **objetos**. Sin embargo, es un enfoque que no se suele utilizar en producción. Simplemente es interesante saberlo para darse cuenta que en Javascript todo pueden ser objetos:

```
const saludar = new Function("return 'Hola';");

saludar(); // 'Hola'
```

Funciones anónimas

Las **funciones anónimas** o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};

saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

Observa que en la última línea del ejemplo anterior, estamos **ejecutando** la variable, es decir, ejecutando la función que contiene la variable. Sin embargo, en la línea anterior hacemos referencia a la variable (*sin ejecutarla, no hay paréntesis*) y nos devuelve la función en sí.

La diferencia fundamental entre las funciones por declaración y las funciones por expresión es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si «ejecutamos la variable» antes de declararla, nos dará un error.

Callbacks

Ahora que conocemos las **funciones anónimas**, podremos comprender más fácilmente como utilizar **callbacks** (*también llamadas funciones callback o retrollamadas*). A grandes rasgos, un **callback** (*llamada hacia atrás*) es pasar una **función B por parámetro** a una **función A**, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función:

```
// fB = Función B
const fB = function () {
  console.log("Función B ejecutada.");
};

// fA = Función A
const fA = function (callback) {
  callback();
};

fA(fB);
```

Funciones autoejecutables

Pueden existir casos en los que necesites crear una función y ejecutarla sobre la marcha. En Javascript es muy sencillo crear **funciones autoejecutables**. Básicamente, sólo tenemos que envolver entre paréntesis la función anónima en cuestión (*no necesitamos que tenga nombre, puesto que no la vamos a guardar*) y luego, ejecutarla:

```
// Función autoejecutable
(function () {
  console.log("Hola!!");
})();

// Función autoejecutable con parámetros
(function (name) {
  console.log(`¡Hola, ${name}!`);
})("Manz");
```

De hecho, también podemos utilizar parámetros en dichas funciones autoejecutables. Observa que sólo hay que pasar dichos parámetros al final de la función autoejecutable.

Ten en cuenta, que si la función autoejecutable devuelve algún valor con **return**, a diferencia de las **funciones por expresión**, en este caso lo que se almacena en la variable es el valor que devuelve la función autoejecutada:

```
const f = (function (name) {
  return `¡Hola, ${name}!`;
})("Manz");

f; // '¡Hola, Manz!`
typeof f; // 'string'
```

Clausuras

Las **clausuras** o cierres, es un concepto relacionado con las funciones y los ámbitos que suele costar comprender cuando se empieza en Javascript. Es importante tener las bases de funciones claras hasta este punto, lo que permitirá entender las bases de una clausura.

A grandes rasgos, en Javascript, una clausura o cierre se define como una función que «encierra» variables en su propio ámbito (*y que continúan existiendo aún habiendo terminado la función*). Por ejemplo, veamos el siguiente ejemplo:

```
// Clausura: Función incr()
const incr = (function () {
  let num = 0;
  return function () {
    num++;
    return num;
  };
})();

typeof incr; // 'function'
incr(); // 1
incr(); // 2
incr(); // 3
```

Tenemos una **función anónima** que es también una función autoejecutable. Aunque parece una función por expresión, no lo es, ya que la variable **incr** está guardando lo que devuelve la función anónima autoejecutable, que a su vez, es otra función diferente.

La «magia» de las clausuras es que en el interior de la función autoejecutable estamos creando una variable **num** que se guardará en el ámbito de dicha función, por lo tanto existirá con el valor declarado: **0**.

Por lo tanto, en la variable **incr** tenemos una función por expresión que además conoce el valor de una variable **num**, que sólo existe dentro de **incr**. Si nos fijamos en la función que devolvemos, lo que hace es incrementar el valor de **num** y devolverlo. Como la variable **incr** es una clausura y mantiene la variable en su propio ámbito, veremos que a medida que ejecutamos **incr()**, los valores de **num** (*que estamos devolviendo*) conservan su valor y se van incrementando.

Arrow functions

Las **Arrow functions**, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de **ECMAScript 6**. Básicamente, se trata de reemplazar eliminar la palabra **function** y añadir **=>** antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};

const func = () => {
  return "Función flecha.";
};
```

Sin embargo, las **funciones flechas** tienen algunas ventajas a la hora de simplificar código bastante interesantes:

- Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves (*{}*).
- Además, en ese caso, automáticamente se hace un **return** de esa única línea, por lo que podemos omitir también el **return**.
- En el caso de que la función no tenga parámetros, se indica como en el ejemplo anterior: **() =>**.
- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo: **e =>**.
- En el caso de que la función tenga 2 ó más parámetros, se indican entre paréntesis: **(a, b) =>**.
- Si queremos devolver un objeto, que coincide con la sintaxis de las llaves, se puede englobar con paréntesis: **({name: 'Manz'})**.

Por lo tanto, el ejemplo anterior se puede simplificar aún más:

```
const func = () => "Función flecha."; // 0 parámetros: Devuelve "Función flecha"
const func = (e) => e + 1; // 1 parámetro: Devuelve el valor de e + 1
const func = (a, b) => a + b; // 2 parámetros: Devuelve el valor de a + b
```

Las **funciones flecha** hacen que el código sea mucho más legible y claro de escribir, mejorando la productividad y la claridad a la hora de escribir código.