

Funciones 10

En este capítulo vamos a ver un elemento muy importante, sobretodo para los que no han programado nunca y con Javascript están dando sus primeros pasos en el mundo de la programación. Hablamos del concepto de **función** y los usos que tiene. Para los que ya conozcan el concepto de función también será un capítulo útil, pues también veremos la sintaxis y funcionamiento de las funciones en Javascript.

10.1 Concepto de función

Podemos definir una función como una serie de instrucciones que “empaquetamos” (englobamos) dentro de un bloque de llaves (`{}`). Este bloque se podrá luego ejecutar desde cualquier otro sitio con solo llamarlo. Por ejemplo, en una página web puede haber una función para cambiar el color del fondo y desde cualquier punto de la página podríamos llamarla para que nos cambie el color cuando lo deseemos.

Esto sucede porque a la hora de hacer un programa ligeramente grande, existen determinadas acciones que se pueden concebir de forma independiente, y que son más sencillas de resolver que el problema entero. Además, estas acciones suelen repetirse veces a lo largo de la ejecución del programa.

Estas acciones se pueden agrupar en funciones, las cuales permiten que no tengamos que repetir una y otra vez ese código en nuestros scripts, sino que simplemente llamamos a la función y ella se encarga de hacer todo lo que debe.

Las funciones se utilizan constantemente. No sólo las definimos nosotros, sino también ya hay funciones definidas en el sistema, pues todos los lenguajes de programación tienen un montón de funciones para realizar procesos habituales como por ejemplo obtener la hora, imprimir un mensaje en la pantalla o convertir variables de un tipo a otro. Ya hemos visto alguna función en nuestros sencillos ejemplos anteriores cuando hacíamos un `document.write()` en realidad estábamos llamando a la función `write()` asociada al documento de la página que escribe un texto en la página.

En este capítulo vamos primero a ver cómo realizar nuestras propias funciones y cómo llamarlas luego. A lo largo del manual veremos también muchas de las funciones definidas en Javascript que debemos utilizar para realizar distintos tipos de acciones habituales.

10.2 Cómo se define una función

Una función se debe definir con una sintaxis especial que vamos a conocer a continuación.

```
function nombrefuncion () {  
    instrucciones de la función  
    ...  
}
```

- Primero se escribe la palabra `function`, reservada para este uso.
- Seguidamente se escribe el nombre de la función, que, como los nombres de variables, puede tener números, letras y algún carácter adicional como el guion bajo.
- A continuación se colocan entre llaves las distintas instrucciones de la función. Las llaves definen el bloque de sentencias que componen la función y no son opcionales. Además es útil colocarlas siempre como se ve en el ejemplo, para que se vea fácilmente la estructura de instrucciones que engloba la función.

Veamos un ejemplo de función para escribir en la página un mensaje de bienvenida dentro de etiquetas `<H1>` para que quede más resaltado.

```
function escribirBienvenida() {  
    document.write("<H1>Hola a todos</H1>");  
}
```

Esta sencilla función se limita a sacar un texto como título en la web a través del `document.write` que ya conocemos.

Aunque en este caso solo hay una sentencia entre las llaves de la función, una función puede tener tantas sentencias como nos hagan falta.

10.3 Cómo invocar funciones

Para ejecutar una función la tenemos que **invocar**. Esto se puede hacer en cualquier parte de nuestro código. Con eso conseguiremos que se ejecuten todas las instrucciones que tiene la función entre las dos llaves.

De momento, para ejecutar la función utilizamos su nombre seguido de los paréntesis.

```
NombreDeLaFuncion();
```

10.4 Dónde colocar mis funciones

En principio, podemos colocar las funciones en cualquier parte de la página, siempre entre etiquetas `<script>`, claro está. Sin embargo, para nosotros y con objeto de tener el código ordenado, la colocación de etiquetas `<script>` queda limitada a la cabera de la página (el `<head>`)

```
<script type="text/javascript">
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</script>
```

Una limitación muy importante cuando se usan funciones es que una función no puede ser llamada si antes no ha sido definida. Este ejemplo no funcionaría:

```
<script type="text/javascript">
miFuncion()
...
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</script>
```

Estoy llamando a `miFuncion()` antes de que haya sido definida (se esta definiendo después).

Nota del autor: En realidad si funciona porque Javascript revisa todo el documento en busca de la función que se invoca. Sin embargo, esto no suele ocurrir con otros lenguajes de programación y, por tanto, es una mala practica de programación el llamar a funciones que no han sido definidas antes.

La manera correcta sería:

```
<script type="text/javascript">
function miFuncion() {
    //hago algo...
    document.write("Esto va bien")
}

...

miFuncion()
</script>
```

10.5 Parámetros

Los parámetros se usan para mandar valores a la función, con los que ella trabajará para realizar las acciones. Son los valores de entrada que recibe una función. Por ejemplo, una función que realizase una suma de dos números necesitará como parámetros a esos dos números. Los dos números son la entrada, así como la salida sería el resultado, pero eso lo veremos más tarde.

Veamos un ejemplo anterior en el que creábamos una función para mostrar un mensaje de bienvenida en la página web, pero al que ahora le vamos a pasar un parámetro que contendrá el nombre de la persona a la que hay que saludar.

```
function escribirBienvenida(nombre) {
    document.write("<h1>Hola " + nombre + "</h1>")
}
```

Como podemos ver en el ejemplo, para definir en la función un parámetro tenemos que poner el nombre de la variable que va a almacenar el dato que le pasemos. Esa variable (que en el ejemplo se llama nombre) tendrá como valor el dato que le pasemos a la función cuando la llamemos.

```
escribirBienvenida("Alberto García");
```

Como se ve, para llamar a una función que tiene parámetros se coloca entre paréntesis el valor del parámetro.

Al llamar a la función así, el parámetro nombre toma como valor *"Alberto García"* y al escribir el saludo por pantalla escribirá *"Hola Alberto García"* entre etiquetas `<h1>`.

Importante destacar que, además, la el parámetro dejará de existir cuando la función termine su ejecución. Es decir, solo puedo usarlo dentro de esa función.

10.5.1 Múltiples parámetros de entrada

Una función puede recibir tantos parámetros como queramos. Para ello se colocan los parámetros separados por comas dentro de los paréntesis.

```
function escribirBienvenida(nombre,colorTexto) {  
    document.write("<FONT color=' " + colorTexto + "'>")  
    document.write("<H1>Hola " + nombre + "</H1>")  
    document.write("</FONT>")  
}
```

Nota del autor: Recuerda que a día de hoy la etiqueta está obsoleta. Aquí se ha usado a modo de ejemplo para usar varios parámetros.

Si se define la función como el ejemplo, para invocarla debemos colocar los valores de los parámetros entre los paréntesis:

```
var miNombre = "Pepe";  
var miColor = "red";  
escribirBienvenida(miNombre,miColor);
```

Hemos colocado entre los paréntesis dos variables en lugar de dos textos entrecomillados. Cuando colocamos variables entre los parámetros en realidad lo que estamos pasando a la función son los valores que contienen las variables y no las mismas variables.

Otro ejemplo sencillo:

```
funtion sumar(n1,n2) {  
    var res = n1+n2;  
    console.log(res);  
}
```

Para invocarla usaríamos:

```
sumar(22, 8);
```

Si no lo hago de esa forma, la función falla porque n1 y n2 no tienen valores y no puede hacer la suma de ambos.

10.5.2 Parámetros de entrada opcionales

En Javascript, a diferencia de otros lenguajes, podemos definir parámetros opcionales. Estos parámetros tendrán unos valores por defecto que tomaran en caso de que en la llamada a la función no aparezcan.

La definición de esos valores por defecto se puede hacer de dos formas diferentes:

1. En los paréntesis de la función.
2. Con el operador `or` `||`

La primera forma es la más intuitiva y basta con igualar el parámetro que queramos a un valor:

```
function sumar(n1=10,n2=5) {  
    var res=n1+n2;  
    console.log(res);  
}
```

Ahora al invocar a la función puedo colocar 2 parámetros, 1 o ninguno y la función no va a fallar.

```
suma(22,8); //va a sumar el 22 y el 8  
suma(22); //va a sumar el 22 y el 5 (valor por defecto de n2)  
suma(); //va a sumar 10 y 5 (valores por defecto de n1 y n2)
```

Esta forma de actuar puede provocar problemas si no se usa correctamente. Por ejemplo:

```
function sumar(n1=10,n2) {  
    var res=n1+n2;  
    console.log(res);  
}
```

La operación de sumar daría error (NaN) si invocamos a la función con un solo parámetro:

```
Sumar(22); //el 22 se mete en n1 y n2 queda vacío
```

Para evitar estos problemas, en la definición de la función, los valores por defecto se colocan comenzando desde el último parámetro definido.

```
function sumar(n1,n2=5) {  
    var res=n1+n2;  
    console.log(res);  
}
```

Ahora funcionaría correctamente al hacer:

```
Sumar(22); //el 22 se mete en n1 y n2 toma el valor 5
```

Otra opción para evitar este problema es usar el valor `undefined` en la llamada de la función en todo parámetro en el que queramos usar su valor por defecto:

Por ejemplo, para la función antes indicada:

```
function sumar(n1=10,n2) {  
    var res=n1+n2;  
    console.log(res);  
}
```

Si llamamos a la función con solo un parámetro, hemos visto que el resultado es NaN:

```
Sumar(22); //el 22 se mete en n1 y n2 queda vacío
```

Nosotros podemos especificar que ese valor sea el del segundo parámetro haciendo:

```
Sumar(undefined,22); //10+22 = 32
```

De esta forma no habría ningún problema: n1 tomará el valor por defecto (10) y n2 tomará el valor 22.

La segunda forma es utilizando el operador OR dentro del código de la función:

```
function sumar(n1,n2){  
    n1 = n1 || 10;  
    n2 = n2 || 5;  
    var res=n1+n2;  
    console.log(res);  
}
```

El funcionamiento es exactamente igual que colocando los valores en los paréntesis:

```
suma(22,8); //va a sumar el 22 y el 8  
suma(22); //va a sumar el 22 y el 5 (valor por defecto de n2)  
suma(); //va a sumar 10 y 5 (valores por defecto de n1 y n2)
```

10.5.3 Parámetros de salida/retorno

Las funciones también pueden devolver valores. De esta forma, al ejecutar la función se pueden realizar varias acciones/cálculos y, al acabar, devolver un valor como resultado de ese proceso.

Por ejemplo, una función que calcula el cuadrado de un número tendrá como entrada (parámetros) ese número y como salida tendrá el valor resultante de hallar el cuadrado de ese número. Es decir, devolverá ese valor para ser usado en otro lugar del código, no lo muestra por pantalla/consola.

Esto es muy útil dado que conseguimos que las funciones sean una 'caja negra' que hacen cosas. Si queremos, les pasamos unos datos de entrada y recibimos unos datos de salida. De esta forma, cualquiera puede usar una función sin necesidad de saber como está programada.

Siempre que queramos devolver un valor en una función debemos hacer uso del comando `return`.

Veamos un ejemplo de función que calcula la media de dos números. La función recibirá los dos números y devolverá el valor de la media.

```
function media(valor1,valor2){  
    var resultado;  
    resultado = (valor1 + valor2) / 2;  
    return resultado;  
}
```

Como se ve en el ejemplo, para indicar el valor que devolverá la función se utiliza la palabra `return` seguida del valor que se desea devolver. En este caso se devuelve el contenido de la variable `resultado`, que contiene la media de los dos números.

Cuando queremos hacer uso del valor devuelto por una función, hay que asignar esa función a una variable. De esta forma, la variable se llenará con el valor que devuelva la función. A continuación, podemos usar esa variable normalmente dentro de nuestro código.

```
var miMedia;  
miMedia = media(12,8);  
document.write (miMedia);
```

Es muy importante destacar dos cosas sobre la sentencia `return`

- Sólo puede devolver un valor. Si queremos devolver más de un valor, debemos usar estructuras complejas como arrays.
- Cuando se ejecuta, automáticamente se sale de la función. Esto quiere decir que si tengo más instrucciones escritas a continuación de `return`, estas no se van a ejecutar nunca.

Las dos consideraciones anteriores no impiden que en una misma función podamos colocar más de un `return`.

Como se ha indicado, sólo se va retornar un valor, pero dependiendo de lo que haya sucedido en la función, podrá ser de un tipo u otro, con unos datos u otros.

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

En esta función podemos ver un ejemplo de utilización de múltiples `return`. Se trata de una función que devuelve un 0 si el parámetro recibido es par o el valor del parámetro, si este era impar.

10.6 Paso por valor y por referencia

Es importante indicar que los parámetros de las funciones en JavaScript se pasan por valor o por referencia dependiendo de su tipo.

El **paso por valor** quiere decir que los parámetros que le pasamos a una función van a ser copias de las variables originales. De modo que, aunque modifiquemos un parámetro en una función, la variable original que habíamos pasado no cambiará su valor.

Se puede ver fácilmente con un ejemplo:

```
function pasoPorValor(miParametro){
    miParametro = 32;
    console.log("he cambiado el valor a 32");
}

var miVariable = 5;
pasoPorValor(miVariable);
console.log("el valor de la variable es: " + miVariable);
```

En el ejemplo tenemos una función que recibe un parámetro y que modifica el valor del parámetro asignándole el valor 32. También tenemos una variable, que inicializamos a 5 y posteriormente llamamos a la función pasándole esta variable como parámetro.

Como dentro de la función modificamos el valor del parámetro podría pasar que la variable original cambiase de valor, pero como los parámetros no modifican el valor original de las variables (porque en realidad se le pasa una copia) esta no cambia de valor.

De este modo, al imprimir en consola el valor de `miVariable` se imprimirá el número 5, que es el valor original de la variable, en lugar de 32 que era el valor con el que habíamos actualizado el parámetro.

Todas las variables cuyos valores pertenezcan a tipos básicos (números, booleanos y cadenas) serán pasadas por valor en los parámetros de una función.

En el **paso por referencia**, los parámetros que se le pasan a una función van a ser referencias a las variables originales. Es decir, si cambio el valor de esa variable en la función, la variable original también queda modificada (por que tenemos dos referencias apuntando a la misma variable: la original y la de la función)

```
function pasoPorReferencia(miParametro) {  
    miParametro[0] = 32;  
    console.log("he cambiado el primer cajon a 32");  
}  
  
var lista = [0,1,2,3,4,5];  
pasoPorReferencia(lista);  
console.log("el valor de la lista es: " + lista);
```

En el ejemplo tenemos una función que recibe un parámetro tipo array y que modifica el valor de la primera celda del array asignándole el valor 32.

Como es normal, fuera de la función tenemos una variable tipo array, que inicializamos con 6 valores (del 0 al 5). A continuación, llamamos a la función pasándole el array como parámetro.

Como dentro de la función modificamos uno de los valores del array que se pasa como parámetro, el array original queda modificado también porque el parámetro se ha pasado por referencia.

De este modo, al imprimir en consola el valor de `lista` después de llamar a la función, se imprimirá la lista con el valor de la primera celda modificado (32).

Todas las variables cuyos valores no pertenezcan a tipos básicos (es decir, arrays, matrices y objetos en general) serán pasadas por referencia en los parámetros de una función.

10.7 Ámbito de las variables en funciones

Dentro de las funciones podemos declarar variables, aparte de los parámetros (que ya sabemos que son variables que se declaran en la cabecera de la función y que se inicializan al llamar a la función).

Todas las variables y parámetros declarados en una función son **variables locales**, es decir, solo tendrán validez durante la ejecución de la función. No puedo usar esas variables en ningún otro sitio que no sea esa función.

Las **variables globales** son variables que se definen fuera de cualquier función de mi código. Esas variables pueden ser “vistas” y utilizadas por cualquier función de mi código (con el problema que ello conlleva si no se usan con cuidado).

El uso de variables globales se considera una mala práctica en programación.

Es importante destacar que, a la hora de definir una variable en una función debemos usar las sentencias que existen para ello (`var`, `let` o `const`).

Si no lo hacemos así (y sólo ponemos el nombre cuando nos haga falta) Javascript entenderá que estamos haciendo referencia a una variable global del código. De modo que si no está creada la variable, la crea, pero siempre global en lugar de local a esa función.

```
var resultado = 3; //variable global
function girar(){
    resultado = 73; //Como solo pongo el nombre, usa la global
    alert(resultado); //sale 73
}
...
girar();
alert(resultado) //sale 73
```

Nótese como cambia el resultado con sólo poner `var` al definir la variable:

```
var resultado = 3; //variable global
function girar(){
    var resultado = 73; //Ahora usa esa variable local
    alert(resultado); //Sale 73
}
...
girar();
alert(resultado) //sale 3
```

Como vemos en el ejemplo anterior, si queremos, podemos declarar variables locales en funciones que tengan el mismo nombre que una variable global del código.

En ese caso, dentro de la función la variable que tendrá validez es la variable local (la definida dentro de la función) y fuera de la función tendrá validez la variable global a la página.

10.7.1 Formas de definir variables

Como se vio en los primeros capítulos de este manual, en Javascript, a diferencia de la mayoría de lenguajes que siempre declaran las variables de la misma forma, se pueden definir las variables usando varias palabras reservadas.

Hasta ahora solo hemos visto `var` a la hora de definir variables porque no teníamos los conocimientos necesarios para entender las otras formas. Sin embargo, en este apartado por fin vamos a explicar lo que significa declarar una variable de una forma u otra.

Existen tres tipos de declaraciones en Javascript:

1. **var**: Las variables definidas de esta forma tienen un ámbito local o global. Es decir, son visibles y se pueden usar dentro de las funciones en las que son definidas pero, si se definen fuera de funciones, serán variables globales como ya se ha explicado en este apartado.
2. **let**: declara una variable que siempre será local y cuyo ámbito será de bloque, es decir, lo que hay entre dos llaves `{}`. Lo ideal es usarla en bloques `if`, `for`, `while` ya que no podrán ser usadas fuera de esos bloques. Obviamente, si defino una variable dentro de una función usando `let`, esta se comportará como una variable definida usando `var`.
3. **const**: funciona igual que `let` pero además impide que esa variable se pueda sobre-escribir (sería una constante).

Usando var como global:

```
var i = 10;


function prueba() {
    console.log(i*3);
}
```



Como la variable `i` se ha definido fuera de cualquier función, se define como global y todas las funciones pueden usarla.

Usando var como local:

```
function prueba() {  
    var i;  
    for (i=0; i<4; i++){  
        console.log(i+1);  
    }  
    i += 10;  
    console.log("i: "+i);  
}
```



```
1  
2  
3  
4  
Valor de i: 14
```

La variable `i` ha sido definida dentro de la función usando `var` y puede ser utilizada en cualquier punto dentro de esta.

Usando let:

```
function prueba() {  
    for (let i=0; i<4; i++){  
        console.log(i+1);  
    }  
    i += 10;  
    console.log("i: "+i);  
}
```



```
1  
2  
3  
4  
Uncaught ReferenceError: i is not defined  
prueba http://127.0.0.1:5500/codeJS.html:74  
onclick http://127.0.0.1:5500/codeJS.html:1  
[Saber más]
```

La variable `i` ha sido definida sólo dentro del bucle FOR (bloque creado entre las dos llaves del bucle). Al intentar usar la variable fuera de ese bloque, da error porque no existe.

Usando *const*:

```
function prueba() {  
  for (const i=0; i<4; i++) {  
    console.log(i+1);  
  }  
  i += 10;  
  console.log("i: "+i);  
}
```



En este ejemplo, la variable *i* ha sido definida dentro del bloque definido por el bucle FOR pero como constante. Puede usarse en el bucle en expresiones o mostrándola pero no puede cambiar su valor y un bucle FOR siempre cambia el valor de su variable índice. Por eso sale la primera suma (no hay asignación en esa suma) y luego da fallo. Además, las dos líneas fuera del bucle también producirían errores: la variable *i* no existe en ese punto del programa.

10.8 funciones anónimas

Javascript permite definir un tipo de funciones que no tienen nombre (anónimas) y que se transforman en un tipo de datos, es decir, que deben asignarse a una variable. Estas funciones se denominan *funciones anónimas*. Su sintaxis es la siguiente:

```
var variable = function(parámetros) {  
  //codigo de la funcion  
}
```

Para ejecutar esa función debo hacerlo a través de la variable a la que ha sido asignada:
Por ejemplo:

```
var multi = function(a,b) {  
  return a*b;  
}
```

Ahora en la variable *multi* tengo una función.

Para ejecutarla debemos hacer cosas como:

```
multi(3,6); //no ocurre nada porque se pierde el valor devuelto

console.log(multi(3,6)); //sale por consola 18

var res = multi(3,6);
alert(res); //sale por pantalla 18
```

Las funciones anónimas se utilizan mucho para definir funciones que deben ir asociadas a variables. Los ejemplos más comunes son: asociar eventos y creación de métodos de clases.

Un ejemplo:

```
var elemento = document.getElementById("carta");
//evento al hacer click
elemento.onclick = function() {
    borrar(this);
    var nuevo = crearNuevo();
    maquetar(nuevo);
}
//evento al pasar encima
elemento.onmouseover = function() {
    if (this.src == "img/paisaje.jpg") {
        inicializar(450);
    } else {
        inicializar(this.src);
    }
}
```

10.9 Funciones autoinvocadas o autoejecutadas

Además de las comentadas en el punto anterior, existen un tipo especial de funciones anónimas que tienen la capacidad de ejecutarse por si solas sin necesidad de que sean invocadas: *las funciones autoinvocadas, funciones que son invocadas inmediatamente (IIFE: Immediately-invoked function expressions) o funciones en forma de expresión*. Su sintaxis es la siguiente:

```
(function() {  
    //codigo de la funcion  
})( parámetros si los tiene)
```

Como se ve en la plantilla, esas funciones no tienen nombre (ya que no necesitan ser invocadas) y los parámetros deben ir en los paréntesis de abajo, no en los que va junto a la palabra `function`.

Además es muy importante el que toda la definición va metida entre dos paréntesis. Haciendo eso, hacemos creer al lenguaje que lo que hay entre esos paréntesis es una expresión y, por tanto, debe resolverla (que, como es una función, la ejecuta).

```
var saludo = "hola";  
(function(nombre) {  
    var saludo = "buenos días";  
    console.log(saludo + " " + nombre);  
  
})("Pepe"); //>> buenos días Pepe
```

Como se ve en el ejemplo (y como ya aprendimos en el punto anterior), las variables que definamos dentro de la función sólo son visibles en esa función. Por eso el ejemplo anterior muestra *buenos días Pepe* y no *hola Pepe*.

El uso más común que se le da a este tipo de funciones es la privacidad de los datos en el código. Es decir, evitar que tu código se sitúe en el ámbito global del programa (afecte a variables globales), y pueda ocasionar problemas al utilizarlo con alguna otra librería o framework que sí suelen usar variables globales.

Más usos/beneficios de las IIFE: <https://programacionymas.com/blog/funciones-javascript-invocadas-inmediatamente-IIFE>

10.10 Funciones flecha (Arrow Functions)

Las funciones flecha son la representación actual de las funciones normales que hemos visto hasta ahora. Es decir, cualquier función que definamos con `function` podemos definirla usando la notación flecha y viceversa. Su sintaxis es:

Función sin parámetros:

```
//Usando function  
function nombre() {  
    . . .  
}
```

```
//Notación flecha  
const nombre = () => {  
    . . .  
}
```

Función con parámetros:

```
//Usando function  
function nombre(p1,p2) {  
    . . .  
}
```

```
//Notación flecha  
const nombre = (p1,p2) => {  
    . . .  
}
```

Función que devuelve valores (varias líneas de código):

```
//Usando function  
function nombre(p1,p2) {  
    . . .  
    return valor;  
}
```

```
//Notación flecha  
const nombre = (p1,p2) => {  
    . . .  
    return valor;  
}
```

Función que devuelve valores (solo una línea de código):

//Usando function

```
function nombre(p1,p2){  
    return valor;  
}
```

//Notación flecha

```
const nombre = (p1,p2)=> valor
```

Como se ha indicado, si bien actualmente se sigue usando la definición de funciones con `function`, cada vez es más común el uso de la notación flecha.

Esto es debido a que, aunque parezca poco intuitivo al principio (sobre todo a aquellos que vienen de programar en otros lenguajes), su sintaxis permiten reducir mucho lo que se escribe en numerosos casos, pej, funciones anónimas y, sobre todo, en funciones propias de JavaScript que tienen como parámetros otras funciones (callbacks).

Ejemplo de función anónima:

//con function

```
var multi = function(a,b){  
    return a*b;  
}
```

//Notacion flecha

```
var multi = (a,b)=> a*b;
```