Capítulo 10 - Funciones

Las funciones de JavaScript son el alma de este lenguaje, por eso se consideran *ciudadanos de primera clase (first-class citizen)*, además de *entidades de orden superior*.

En JavaScript, las funciones tienen *"super poderes"*. Estos son algunos de los más importantes:

- Ser pasadas como parámetros *(callbacks)*.
- Ser parte de los objetos como métodos.
- Ser asignadas a una variable (función anónima).
- Ser retornadas por otra función.

Una de las claves para entender la importancia de las funciones, aún cuando estamos dando nuestros primeros pasos en JavaScript es la reusabilidad. Podemos crear partes de código que fácilmente podremos reutilizar a lo largo de una aplicación o incluso a lo largo de muchos programas y aplicaciones… llegando incluso a crear nuestras propias librerías.

Pero para dominar la reusabilidad y respetar con profundidad el principio de programación *DRY (Don't Repeat Yourself)*, deberemos en cualquier caso ser capaces de manejar los parámetros y el retorno de las funciones, algo de lo que hablaremos mucho en este capítulo.

Las funciones, especialmente como *parámetro (callback)*, también será nuestra puerta de entrada al maravilloso, caótico y paradigmático *mundo de la asincronía*.

Manejo

- **Declarar funciones**

  Como sentencia:

```
function miFuncion (){
        console.log("Hola!")
}
```

  Como valor de una variable:

```
var miFuncion = function(){
        console.log("Hola!")
}
```

Como método en un objeto:

```
var miObjeto = {
        propiedad: "Soy una propiedad",
        metodo: function(){
                console.log("Hola!")
        }
}
```

- **Ejecutar funciones**

  Aunque pueda parecer algo extraño, desde el principio, ya estábamos ejecutando funciones.

```
// Recuerdas isNaN?
console.log("Recuerdas isNaN?", isNaN(NaN))
```

Ahora ejecutamos nuestras propias funciones y métodos.

```
var miFuncion = function(){
        console.log("Hola!");
}

        function otraFunción() {
                console.log("Hola de nuevo!");
        }

        var obj = {
                metodo: function () {
                        console.log("Hola... ahora como método!");
                }
        }

miFuncion();
otraFunción();
obj.metodo();
```

Argumentos y parámetros

Cuando queremos hacer funciones con un nivel de abstracción realmente alto, tenemos que recurrir al aislamiento. De tal forma que nuestra función no dependa de ciertas variables o datos externos a ella.

Cuando definimos (creamos) una función, podemos incluir ciertos parámetros entre los paréntesis que actuarán como referencias. Funcionarán internamente igual que variables, de tal forma que a la hora de ejecutar la función… podremos pasarle ciertos argumentos y así tener funciones con un mayor nivel de abstracción.

Uso Normal

```
// Declarando Parámetros
function sumar (p1, p2){
        console.log("suma:", p1 + p2)
}

// Pasando Argumentos
sumar(2, 3);
```

El exceso de argumentos no es un problema.

```
// Declarando Parámetros
function sumar (p1, p2){
        console.log("suma:", p1 + p2)
}

// Pasando Argumentos
sumar(2, 3, "más datos...", 45, true);
```

La falta de argumento crea un valor indefinido.

```
function testeando (p1, p2){
        console.log("p1:", p1);
        console.log("p2:", p2)
}

// Pasando Argumentos
testeando(2);
```

Parámetros opcionales

Podremos simplificar enormemente la ejecución de las funciones si, definimos ciertos valores por defecto para aquellos parámetros que consideremos opcionales.

Este trabajo adicional por nuestra parte, se verá recompensado posteriormente en tareas de soporte y documentación que no tendremos que realizar.

Trabajar con valores por defecto nos ayudará mucho para construir librerías y un código modular eficiente.

Básicamente existen dos maneras de hacer esto.

- **Utilizando el operador ||**

```
function userID(nombre, numero) {
  numero = numero || "000000E";
  console.log("ID:", nombre + "-" + numero)
}

userID("Ulises", 31); // Ulises-31
userID("Oscar");      // Oscar-000000E
userID("Pepe", 0)     // Pepe-000000E
```

Aunque este operador hace un buen trabajo se equivoca con el 0 -entre otros- por eso no es recomendable utilizarlo, especialmente cuando se encarga de gestionar el parámetro por defecto de valores numéricos.

- **Utilizando un *if***

Podemos hacer una validación por tipo, lo que descartará ciertos falsos positivos como en el caso del 0.

```
function sumar(a, b) {
  if(typeof b === 'undefined'){
          b = 0;
  }

  return a+b;
}

sumar(2);    // 2
sumar(2, 8); // 10
```

Con un operador ternario se hace más compacto pero menos legible:

```
function sumar(a, b) {
  b = typeof b !== 'undefined' ?  b : 0;
  return a+b;
}

sumar(2);    // 2
sumar(2, 8); // 10
```

El orden es clave

El orden de los parámetros es muy importante, ya que su posición puede alterar enormemente la usabilidad a la hora de la ejecución, por eso el orden siempre será:

- Parámetros fijos (primero).
- Parámetros opcionales (después).

Objetos como argumento

Se considera una buena práctica, pasar un único objeto como parámetro si estamos manejando más de tres parámetros fijos.

De esta forma además de agrupar todo fácilmente, también podemos cambiar el orden de entrada de datos.

Es importante recordar que debemos documentar muy bien lo que esperamos, que contenga el objeto, de lo contrario nuestros métodos y funciones pueden ser un infierno para cualquier otro programador e incluso para nosotros mismos pasado un tiempo.

```
contactos = [];

function crearContacto (nombre, usuarioTwitter, referencias, notas, fotoUrl){
        contactos.push({
                "nombre": nombre,
                "@": "@" + twitter
        })
}

crearContacto("Oscar", "inventado", "amigos...", "etc...", "más cosas...");
```

¡Refactorizemos!

```
contactos = [];

function crearContacto (datos){
        contactos.push({
                "nombre": datos.nombre,
                "@": "@" + datos.twitter
        })
}

// Puedo pasar los atributos en el orden que quiera
crearContacto({twitter: "inventado", nombre: "Pepe", fotoUrl: "http..."});
```

Avanzado: Objeto *arguments*

El Objeto Arguments no es un array, solo es similar.

```
function pruebaArgumentos () {
        console.log(arguments);
        console.info(arguments[0]);
        console.info(arguments[1]);
}

pruebaArgumentos (1, "vale", true);
```

Conversión array requiere de ciertos conocimientos avanzados en el uso de *prototype* y *this*. Os dejo una función que os ayudará a realizar esta conversión de una forma fácil.

```
function conversorArgumentos(arguments) {
    var argumentos = Array.prototype.slice.call(arguments);
    return argumentos.sort();
}
```

Retorno

Otro de los puntos fuertes a la hora de plantear estructuras de código modulares y reutilizables, es tener en cuenta el retorno.

El retorno nos permite devolver un valor al terminar de ejecutarse la función. Este valor puede ser cualquier tipo de dato de los muchos que tenemos en JavaScript. Por supuesto, también funciones y objetos.

Cómo utilizar funciones que retornen valores en función de ciertas operaciones realizadas.

```
        function validarPar(numero){
                var esPar = numero % 2 !== 1;
                var mensaje;

                if (esPar) {
                        mensaje = "Bravo! es un número par!";
                } else {
                        mensaje = "ERROR! No es un número par.... ¬¬\"";
                }
                return mensaje;
        };
```

```
12
13          console.log("El 5 es un número par?", validarPar(5));
14          console.log("El 2 es un número par?", validarPar(2));
```

Una suma de cuadrados en el retorno. Las operaciones también pueden ser realizadas en el retorno de la función.

```
1  function sumaCuadrados (a, b) {
2          return (a*a) + (b*b);
3  };
4
5  var resultado = sumaCuadrados(2, 3);
6  console.log("2x2 + 3x3 =", resultado)
```

Anidación

Dentro de una función, podemos crear nuevas funciones al igual que variables de todo tipo. Este es un recurso a tener en cuenta, pero no debemos abusar de la anidación… ya que, el código puede volverse muy difícil de leer y depurar.

```
1  function saludar(quien){
2          function alertaSaludo(){
3                  console.log("hola " +  quien);
4          }
5          return alertaSaludo;
6  }
7
8  var saluda = saludar("Amigo/a");
9  saluda();
```

También podemos usar parámetros, al igual que una función normal.

```
1  function saludar(quien){
2          function alertaSaludo(){
3                  console.log("hola " +  quien);
4          }
5          return alertaSaludo;
6  }
7
8  saludar("Amigo/a")();
```

Ámbito (Scope)

Por defecto en JavaScript existen dos tipos de ámbitos, local y global. Dominar los ámbitos nos hará llegar a ser grandes artesanos, pero no es una tarea sencilla.

En principio aquellas variables que se han declarado fuera de la función, son de ámbito global, y las variables que se declaran en el interior serán consideradas de ámbito local.

Desde cualquier función siempre podremos acceder a todas las variables que se han declarado en el ámbito global, pero desde el exterior de una función no podremos acceder a su ámbito local. Para poder solventar esta limitación se utilizan los retornos que vimos anteriormente y algunos recursos adicionales que veremos más adelante.

```
1  var ambitoGlobal = "Soy una variable Global!";
2
3  function miFuncion () {
4          var ambitoLocal = "Soy una variable Local!";
5          console.log("Desde local puedo ver ambitoLocal?", ambitoLocal);
6          console.log("Desde local puedo ver ambitoGlobal?", ambitoGlobal);
7  }
8
9  console.log("Desde local puedo ver ambitoLocal?", ambitoLocal);
10 //Uncaught ReferenceError: ambitoLocal is not defined(…)
11
12 console.log("Desde local puedo ver ambitoGlobal?", ambitoGlobal);
```

Este juego de ámbito local y global, puede extenderse en el entorno compartido y aislado de las funciones anidadas.

Duplicando Variables

Una mala práctica a la hora de planificar nombres de las variables en nuestra aplicación puede llevarnos a la situación en la que tengamos variables creadas (declaradas) en el ámbito global y en el local con los mismos nombres.

Esto puede ser evitado desde la planificación en una fase temprana o posterior con algún *linter* como *JSHint* o *ESLint*.

Funciones Anónimas

En JavaScript podemos crear tantas funciones como queramos, sin embargo entre los requisitos de creación no esta incluir un nombre necesariamente.

Funciones que retornan funciones

Cuando una función retorna una nueva función, esta nueva función lógicamente será anónima.

```
function saludo(quien){
    return function(){
        console.log("hola " + quien);
    }
}

var saluda = saludo("Amigo/a");
saluda();
```

Podemos ejecutar ambas funciones, sin asignar una variable necesariamente.

```
function saludo(quien){
    return function(){
        console.log("hola " + quien);
    }
}

saludo("Amigo/a")();
```

Funciones anónimas autoejecutadas

Es uno de los patrones más clásicos y utilizados en JavaScript, para encapsular nuestro código y prevenir que pueda ser alterado desde el exterior.

Esta técnica da mucho juego, si tenemos en cuenta que podemos usar el retorno.

Al aislar nuestro código tanto del exterior, podemos pensar que nuestro programa se queda lejos de ser capaz de interactuar con el usuario, pero esto es incorrecto, ya que en JavaScript podremos recurrir a la *programación dirigida por eventos*. Hablaremos en próximos capítulos sobre ello.

```
(function() {
        console.log("hola Amigo/a")
})();
```

Resulta más sencillo de entender esta estructura si entendemos el juego de los paréntesis.

Declaramos una función:

```
(//código)()
```

Lo contenido en el primer paréntesis contiene el código encapsulado, al igual que hacíamos con las operaciones matemáticas en capítulos anteriores.

El segundo paréntesis es el encargado de ejecutar el bloque de código anterior, asi es como logramos que la función sea inmediatamente ejecutada dentro de un ámbito al que no podremos acceder.

Como podemos, ver la estructura básica sería algo así:

```
(function(){})();
```

Aunque existen bastantes variantes y debates:

```
(function(){}());
!function(){}();
+function(){}();
!1%-+~function(){}();
//...
```

Al igual que el resto de funciones podemos hacer uso de los parámetros.

```
( function(quien){
   console.log("hola " + quien);
})("Amigo/a");
```

Objeto Window como parámetro

Aunque por temas de rendimiento -lo más habitual- es pasar como argumento el *objeto window*, así disponemos de una copia dentro del propio ámbito de la función.

```
(function(window){
  // código
})(window);
```

Recursión

Otra manera más *funcional* y divertida de hacer bucles es utilizando la recursión. Básicamente una función es capaz de llamarse a sí misma durante su ejecución, lo que resulta ser una funcionalidad muy atractíva para ciertas operaciones.

Por otro lado, aunque es una práctica muy habitual entre los programadores -que defienden- la *programación funcional* en JavaScript, puede ser complicado prevenir el riesgo de caer en bucles infinitos.

Un clásico donde podemos aplicar recursividad es en el cálculo del *factorial*.

```
function factorial(n){
        if(n <= 1){
            return 1
          } else {
            return n * factorial(n-1)
        }
}

factorial(0); // n! = 1
factorial(1); // n! = 1
factorial(2); // n! = 2
factorial(3); // n! = 6 (3*2*1)
factorial(4); // n! = 24 (4*3*2*1)
factorial(5); // n! = 120 (5*4*3*2*1)
factorial(6); // n! = 720 (...)
```

Callbacks

La primera curiosidad sobre los callbacks:

Es una técnica de programación y no una facilidad del lenguaje, por ello *callback* no es una palabra reservada en JavaScript, y puedes usarla en tu código, si te resulta más legible.

Callbacks en Wikiwand:

"En programación de computadoras, una devolución de llamada o retrollamada (en inglés: callback) es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A"."

Esto quiere decir, **que cuando cierta función termina de realizar todo lo que tiene que hacer, ejecutará una función que le fue pasada como argumento**.

En un principio, este concepto parece complicado, y sin duda lo es, pero este sistema es el primer paso para manejar la asincronía. Esto sucederá cuando nuestro código deja de ejecutarse de manera estructurada línea a línea, por ejemplo con las *peticiones AJAX*, lo que veremos en próximos capítulos.

Comparando por contexto

Cuando tenemos un código síncrono, fácilmente podemos obviar el uso de callbacks, y llegar al mismo resultado, ya que nuestro código sigue un orden lógico.

Sin Callbacks:

```
function primerPaso() {
        console.log("Este es el primer paso");
};

function segundoPaso() {
        console.log("Este es el segundo paso");
};

primerPaso();
segundoPaso();
```

Con Callbacks:

```
function primerPaso(callback) {
        console.log("Este es el primer paso");
        callback();
};

function segundoPaso() {
        console.log("Este es el segundo paso");
};

primerPaso(segundoPaso);
```

Cuando nuestro código se ejecute de forma asíncrona, la única forma de conservar el flujo en orden, será utilizando entre otras cosas *Callabcks* o *Promesas*, como veremos a continuación.

Si has desarrollado alguna vez con JQuery, habrás notado que tiene unas características ligeramente diferentes al JavaScript al que estamos acostumbrados.

```
$('#elemento').fadeIn('slow', function() {
   // código del callback
});
```

Como puedes ver… en muchos métodos, pasamos como argumento una función que declaramos en línea. Básicamente… **¡ya estábamos usando callabacks!** pero no eramos conscientes.

Veamos un ejemplo, un poco condensado. Os ayudaré comentando el código:

```
/*
        Declaramos una función que espera dos parámetros
        - parametro
        - callback
*/
var quieroCallback = function(p1, callback){
    // Consideramos el callback como algo opcional.
    if ((callback){
            // Validamos si es una función o no.
            if (typeof callback === 'function')){
                    /*
                      De ser una función lo ejecutamos y
                      y pasamos como argumento "p1"
                    */
                callback(p1);
            } else {
                    /*
                      Si no se trata de una función...
                      simplemente mostramos ambos datos.
                     */
                console.log(p1, callback);
                }
    }
}

quieroCallback('a', 'b');

quieroCallback('a', function(val){
    console.log(val);
});
```

Asincronía

La naturaleza de la Asincronía

Hasta ahora todo el código que vimos se ejecutaba de una manera lógica, previsible y secuencial. Cada línea de código era ejecutada después de la anterior, tardará lo que tardará. Este estilo de programación es ineficiente y bloqueante, lo que en el mundo de la web es intolerable.

La asincronía es una caracteristica propia de ciertos métodos que permiten su ejecución en un segundo plano. De tal forma que resulta imposible saber cuando terminarán y además antes de terminar su ejecucción se ejecutan la siguiente línea de código.

Cuando en JavaScript se habla de asincronía, lo que realmente esta ocurriendo es que dejamos de ejecutar partes de nuestro script de manera secuencial. Esto crea un efecto curioso que tiene como consecuencia, un script muy escalable y rápido, ya que el sistema no espera a que algo termine para seguir ejecutando el resto del script.

La mala noticia, es que recaerá en el lector todo el peso de controlar esos caballos desbocados. La asincronía es tan potente, que no existe otra forma de trabajar sobre Node.js. Por eso Nodejs está concebido -de principio a fin- como un sistema asíncrono.

Existen muchas formas de manejar la asincronía.

- Paso de continuadores (Callbacks).
- Eventos.
- Promesas (ECMA6 y librerías…).
- Generadores (ECMA6, Closures, etc…).

Nosotros veremos en este capítulo -exclusivamente- la gestión de asincronía por medio de callbacks.

En el próximo capítulo hablaremos de *programación dirigida por eventos* y como gestionar con ello la asincronía.

Para hacer un poco más fluido esta explicación, utilizaremos *setTimeout* que por defecto es una función asíncrona.

Veamos como funciona el código sin gestionar la asincronía:

```
1  function traigoDatos (){
2      // Asincrona
3      setTimeout (function(){
4          console.log ("Esto son mis datos");
5      },2000)
6  }
7
8  function pintoDatos(){
9      // No asincrona
10     console.log("ya tengo los datos");
11 }
12 traigoDatos();
13 pintoDatos();
```

Como puedes ver… los mensajes no salen en el orden correcto. Recuerda que, para pintar datos, el paso previo -siempre- es tener esos datos disponibles.

Ahora vamos a intentar resolver este problema de una manera sencilla. Si introducimos un callback en la función asíncrona, seremos capaces de resolver el problema… aunque tarde 3 segundos o 5 minutos.

```
function traigoDatos (callback){
    // Asincrona
    setTimeout (function(){
        console.log ("Esto son mis datos");
        // Llamamos a Callback cuando haya llegado el fin de traigoDatos.
        callback();
    },2000)
}

function pintoDatos(){
    // No asincrona
    console.log("ya tengo los datos");
}

traigoDatos(pintoDatos);
```

Al ejecutarlo podemos ver que el problema de la asincronía ha sido resuelto.

Normalmente, a la hora de hacer peticiones asíncronas, solemos *pedir/enviar* información al servidor... y hacemos esto a través de peticiones AJAX (también asíncronas). Cuando realizamos ese tipo de llamadas, queremos pasarle al callback los datos que nos han llegado del servidor.

¡Veamos como hacerlo!

```
function traigoDatos (callback){
    // Asíncrona
    setTimeout (function(){
        // muchas cosas pasan...
        var resultado = "Esto son mis datos";
        // Llamamos a Callback y pasamos el resultado
        callback(resultado);
    },2000)
}

function pintoDatos(data){
    // No asíncrona
    console.log("ya tengo los datos:");
    console.log(data);
}

traigoDatos(pintoDatos);
```

Sobrevivir al *Callback Hell*

Callback Hell es una situación que se suele producir cuando los programadores no dominan el manejo de la asincronía, ni el uso de los callbacks. También se produce, cuando no han respetado conceptos básicos de modularización y prevención de anidación desmedida.

Algunas soluciones a este problema:

- No anidar en exceso… ¿Has oído hablar de la *complejidad ciclomática*?.
- Cualquier anidación de funciones a más de dos o tres niveles esta pidiendo a gritos una refactorización.
- No todas las funciones de tu código han de ser anónimas…
- Modularizar y refactorizar son tus dos mejores amigos en JavaScript.
- Gestiona los errores en cada función y no al final de la pila.

Si aún así te ves totalmente incapaz de prevenir este error, siempre puedes recurrir a *Generadores*, *Promesas*, *Funciones Async*… o librerías como *Async*, *Q*, etc…

Documentar

Si recordamos el tercer capítulo, dijimos que *JSDoc* nos resultaría muy útil en el futuro para entender y documentar especialmente nuestras funciones. Veamos de nuevo aquel ejemplo, esta vez con una mirada más crítica.

```
1  /**
2   * Retorna los detalles del libro.
3   * @param {string} title - Título del libro.
4   * @param {string} author - Autor del libro.
5   * @returns {object} title, author, picture (referencia local), code
6   */
7  function Book(title, author) {
8          return {
9                  title: title,
10                 author: author,
11                 picture: "../images/"+author+"/"+title+".jpg",
12                 code: 010203 + author + "/" + title
13         }
14 }
```

¡Volver atrás!:

Ahora puede ser un buen momento para volver a capítulos anteriores, donde era necesario hacer uso de las funciones para gestionar ciertos métodos complejos en arrays y objetos.