Módulo 4 - Funciones y Eventos - Teoría

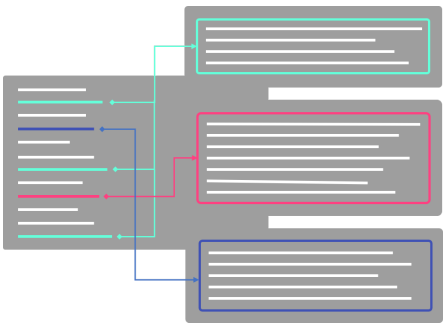
Funciones

Introducción

Hasta ahora habéis programado de la forma más sencilla posible, sin aplicar ningún tipo de organización. Simplemente nos hemos limitado a escribir líneas de código o sentencias, donde cada línea no es más que una instrucción. Así, un programa sencillo está formado instrucciones (o líneas de código) que se ejecutan una detrás de otra, siguiendo un flujo continuo, secuencial. Cuando ya no hay más líneas disponibles, el programa finaliza.

Sin embargo, en los proyectos reales, los programas se vuelven más y más complejos, pudiendo incorporar cientos, miles, decenas de miles y hasta millones de líneas de código. ¿Os imagináis tener que buscar un fallo entre millones de líneas de código? En la práctica esto sería inmanejable. Del mismo modo que en un libro esperamos ver frases, parrafos, páginas, capítulos, etc, en un programa es de esperar cierta organización que nos facilite la tarea de navegar por el código y agrupar de forma lógica nuestras instrucciones. Esto mejora enormemente la legibilidad del código y la mantenibilidad de nuestros programas.

Pero además, la potencia de las funciones no solo radica en su capacidad de organizar el código sino de evitarnos tener que repetir las mismas instrucciones una y otra vez, como veremos un poco más adelante. Una función realizará una tarea concreta, definida, y podremos ejecutar dicha tarea cuando queramos, tantas veces como queramos.



¿Qué es una función?

Una función es una agrupación/contenedor de instrucciones, definidas dentro de un bloque, destinadas a realizar una tarea. Es altamente recomendable que esta tarea sea única, específica y bien definida. Podremos llamar a nuestra función, y por tanto ejecutarla, cuando queramos, tantas veces como queramos.

Por tanto, en resumen, **una función es un bloque de código reutilizable destinado a realizar una tarea concreta**.

Fíjate en el siguiente ejemplo, el código se repite una y otra vez ¿verdad?:

```
var name1 = "Javier";
console.log("Hola! Me llamo " + name1);

var name2 = "Antonio";
console.log("Hola! Me llamo " + name2);

var name3 = "Alba";
console.log("Hola! Me llamo " + name3);

var name4 = "Braulio";
console.log("Hola! Me llamo " + name4);
```

Quedaría mucho mejor usando una función:

```
function sayName(name) {
  console.log("Hola! Me llamo " + name);
}

sayName("Javier");
sayName("Antonio");
sayName("Alba");
sayName("Braulio");
```

Importancia de las funciones

Las funciones son consideradas como una de las piezas fundamentales de cualquier programa (building blocks), y además, una de las piezas más numerosas y repetidas. Se usan masivamente. Su importancia se debe a los beneficios que aportan

- Permiten articular programas complejos y hacerlos más sencillos, legibles y mantenibles. Principio KISS (Keep It Simple Stupid!).
- Permiten extraer código que se repite y encapsularlo en un lugar común, la propia función. Principio DRY (Don't Repeat Yourself).
- Fomentan la estructuración del código en bloques que hace una cosa y una sola cosa, y la hacen correctamente, enfocándonos en una tarea cada vez, sin mezclarlas. Principio de Única Responsabilidad (Single Responsability) y Principio de Separación de Intereses.

Declarando Funciones

Vamos a declarar o definir nuestra primera función. Es muy sencillo, necesitamos los siguientes ingredientes:

- La palabra clave **`function`** .
- Un **nombre** para nuestra función, será el "alias" con el que la llamemos posteriormente.
- **Paréntesis**, encierran la lista de parámetros de entrada.
- Operador de **bloque (llaves)**, donde se define el cuerpo de la función, es decir, donde escribiremos el código de nuestra función.

Ejemplos:

```
function sayHello() {
  console.log("Hola Mundo!");
}

function greetSomeone() {
  console.log("¡Buenos días querido Antonio!");
}

function sumNumbers() {
  var result = 3 + 2;
  console.log("El resultado es:", result);
}
```

Sin embargo, ésta no es la única forma de definir funciones. Javascript es un lenguaje bastante flexible y ofrece variantes:

Funciones anónimas

También podemos declarar una **función sin nombre**, de forma anónima. Por ejemplo:

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

Pero necesitamos referirnos de algún modo a una función para poder ejecutarla posteriormente, de lo contrario, ¿como llamo a una función que no tiene nombre?

La solución es asignarla a una variable, almacenarla directamente en dicha variable. Javascript permite almacenar funciones en variables:

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

Arrow functions

En Javascript existe una sintaxis corta para declarar funciones llamada `arrow functions` o `funciones flecha`. También lo podréis escuchar bajo los términos `fat arrow` o `lambdas` (muy utilizado en otros lenguajes).

Se trata de declarar una función anónima, por tanto necesitaremos una variable para almacenarla. Además, supirmimos la palabra clave `function` y añadimos una flecha que precede al cuerpo de la función.

Ejemplos:

```
var sayHello = () => {
  console.log("Hola Mundo!");
};

var greetSomeone = () => {
  console.log("¡Buenos días querido Antonio!");
};

var sumNumbers = () => {
  var result = 3 + 2;
  console.log("El resultado es:", result);
};
```

IMPORTANTE: Esta variante acortada de declarar funciones no existe por capricho, tiene un motivo, una finalidad, pero lo descubriremos un poco más adelante en el bootcamp. Ahora mismo solo hace falta practicar con su sintaxis.

LLamando a una función

LLamar a una función consiste en ejecutarla, o lo que es lo mismo, ejecutar el código que hemos definido en el cuerpo de la función (todo lo que va entre llaves). Para eso necesitamos referirnos al **nombre de la función** o el **nombre de la variable** donde la hemos almacenado en caso de ser función anónima o `arrow function`.

LLamamos a una función **invocando su nombre + paréntesis**:

```
// Declaración de una función regular
function sayHello() {
  console.log("Hola Mundo!");
}

// Declaración de una arrow función
var greetSomeone = () => {
  console.log("¡Buenos días querido Antonio!");
};

// Las ejecutamos!
sayHello();
greetSomeone();
```

Entrada de una función: Parámetros

Las funciones son la herramientas indicada para reutilizar código y ejecutarlo cuando queramos. Pero sería muy útil poder ejecutar pequeñas variaciones de ese código cada vez, es decir, hacer ejecuciones dinámicas.

Por ejemplo, pensemos en la función `greetSomeone()` ¿no sería mucho más útil poder cambiar el nombre "Antonio" por el que yo quiera en cada momento? De este modo podré saludar a quien quiera, haciendo una función mucho más flexible. Esto se consigue del siguiente modo:

```
function greetSomeone(name) {
  console.log("¡Buenos días querido " + name + "!");
}

greetSomeone("Antonio"); // "Buenos días querido Antonio"
greetSomeone("Javier"); // "Buenos días querido Javier"
```

Por supuesto es también aplicable a su equivalente en `arrow function` :

```
// Versión arrow function
var greetSomeone = name => {
  console.log("¡Buenos días querido " + name + "!");
};

greetSomeone("Antonio"); // "¡Buenos días querido Antonio!"
greetSomeone("Javier"); // "¡Buenos días querido Javier!"
```

Es decir, **desde fuera podemos pasar parámetros de entrada a cualquier función**. Una función admitirá tantos parámetros de entrada como queramos, tan sólo tenemos que separarlos por comas si es más de uno. La misión de los paréntesis en las funciones es precisamente la de contener la lista de parámetros de entrada. Por ejemplo, una versión más elaborada de la función de saludo anterior podría ser:

```
var greetSomeone = (greeting, name) => {
  console.log("¡" + greeting + " querido " + name + "!");
};

greetSomeone("Buenos días", "Antonio"); // "¡Buenos días querido Antonio!"
greetSomeone("Buenas noches", "Javier"); // "¡Buenas noches querido Javier!"
greetSomeone("Hola", "Jaime"); // "¡Hola querido Jaime!"
```

Por supuesto, no todas las funciones que hagamos o nos encontremos necesitarán parámetros de entrada, dependerá de cada caso y si realmente es útil añadirle parámetros de entrada o no.

Salida de una función: Retorno

De igual forma que una función admite parámetros de entrada, también es muy útil poder ofrecer un resultado de salida. Es decir, cuando la función termine, poder devolver un valor de retorno que simboliza el resultado del código que la función ha ejecutado.

Pensemos por ejemplo una función que suma números, que obtendremos como parámetros de entrada, y devuelve el resultado de la suma como valor de retorno:

```
function sum(numberA, numberB) {
  var result = numberA + numberB;
  return result;
}

var sumResult = sum(3, 2);
console.log(sumResult);
// Equivalente a console.log(sum(3, 2));
```

Para devolver un valor en una función, utilizamos la palabra clave `return` . IMPORTANTE: Cuando una sentencia con `return` se ejecuta, la función finaliza. Por eso solemos encontrarla al final de la función.

Al igual que antes, no todas las funciones que hagamos o nos encontremos devolverán obligatoriamente un valor, dependerá de cada caso.

Ámbito de una función

Una función se comporta como un compartimento separado del resto del código. Las variables y objetos que definimos **dentro de una función** permanecen **aisladas** del resto. No podremos acceder a dichas variables desde fuera. Cuando el código de una función se ejecuta no se mezcla, es como una caja negra, puedo pasarle datos de entrada (parámetros de entrada) y me devuelve datos de salida (valor de retorno) pero no tengo acceso a lo que sucede por dentro de la función.

Es por eso que decimos que las funciones tienen su propio **ámbito o contexto**, separado del resto.

```
function greetSomeone() {
  var name = "Antonio";
  console.log("¡Buenos días querido " + name + "!");
}

greetSomeone(); // "¡Buenos días querido Antonio!"
console.log(name); // Uncaught ReferenceError: name is not defined
```

Sin embargo, el ámbito o contexto superior, el que está fuera de todas nuestras funciones, se llama **ámbito o contexto global**. Actúa como el compartimento o contenedor de todo nuestro programa, y se le llama global porque los valores definidos en el ámbito global si son accesibles desde cualquier parte del código.

En la práctica esto se traduce en que, dentro de una función, si puedo tener acceso a su ámbito superior y por tanto si puedo acceder a variables u otras funciones de dicho ámbito.

```
var name = "Antonio";

function greetSomeone() {
  var greeting = "Buenos días";
  console.log("¡" + greeting + " querido " + name + "!");
}

greetSomeone(); // "¡Buenos días querido Antonio!"
console.log(name); // "Antonio"
console.log(greeting); // Uncaught ReferenceError: greeting is not defined
```

Anidando Llamadas de Funciones

Las funciones pueden llamar a su vez a otras funciones y servirse de ellas:

```
var greetSomeone = (greeting, name) => {
  console.log("¡" + greeting + " querido " + name + "!");
};

function startDay(name) {
  greetSomeone("Buenos días", name);
  console.log("¿Qué tal se encuentra hoy?");
}

startDay("Antonio"); // "¡Buenos días querido Antonio!" "¿Qué tal se encuentra hoy?"
```

Métodos vs Funciones

Un método no es más que una función que se ha almacenado en un objeto. Es una simple cuestión de nomenclatura, cuando las funciones quedan almacenadas en propiedades de objetos. Por ejemplo:

```
var person = {
  name: "Javier",
  age: 36,
  saySomething: () => console.log("Soy el profe de hoy");
}
```

`saySomething` en este caso es un método del objeto `person`.

Existen también muchos métodos que ya incorpora el propio lenguaje para manipular ciertos tipos de datos como por ejemplo los strings o cadenas de texto. Por ejemplo:

```
function toUpper(name) {
  console.log(name.toUpperCase());
}
```

Fijaos en la línea `name.toUpperCase()` donde `toUpperCase()` es un método, porque es una función que aparece como propiedad de todos los objetos de tipo string.

Sintaxis abreviadas

Existe una forma peculiar de abreviar funciones en formato `arrow function` cuando el cuerpo consiste en una única línea. En tal caso podemos ahorrarnos el operador de bloque (las llaves) y hacer algo tan expresivo como lo siguiente:

```
// var sayHello = () => {
//   console.log("Hola Mundo!");
// }
var sayHello = () => console.log("Hola Mundo!");
```

Además, si tiene un único parámetro de entrada, también puedo ahorrarme los paréntesis:

```
// var greetSomeone = (name) => {
//   console.log("¡Buenos días querido " + name + "!");
// }
var greetSomeone = name => console.log("¡Buenos días querido " + name + "!");
```

Y cuando lo que hago es devolver un valor de retorno en una sola línea, me puedo ahorrar también el `return`:

```
// function sum(numberA, numberB) {
//  var result = numberA, numberB;
//  return result;
//}
var sum = (numberA, numberB) => numberA + numberB;
```

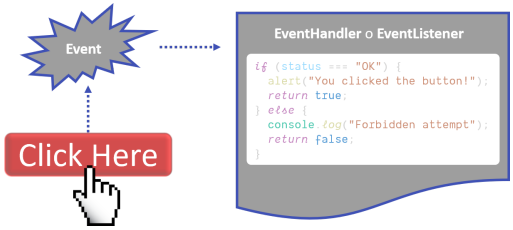
Esta manera de escribir funciones refuerza la idea de que una función consiste en transformar una entrada en una salida.

Eventos

Javascript se ideó para añadir interactividad a nuestras páginas web y convertirlas así en aplicaciones completas. De este modo, a medida que un usuario interactua con una interfaz web, esperamos que dicha aplicación reaccione o responda a dichas interacciones. Por ejemplo, si un usuario hace click en un botón de "Reservar", esperamos que se complete la reserva de su habitación, o si el mismo usuario selecciona un idioma distinto esperamos que la pagina se traduzca, o si busca una palabra que aparecezca un resultado, etc.

La naturaleza de estos eventos puede ser variada en una aplicación. Hemos empezado hablando de interacciones entre el usuario y la interfaz, pero no son el único tipo de eventos que puede darse. Una aplicación también podría necesitar interactuar con un servidor web, para recuperar los datos de un producto almacenado en una base de datos online, por ejemplo. O también podríamos querer que nuestra aplicación interactue con otras aplicaciónes, dispositivos, etc.

En definitiva, como usuarios esperamos que la aplicación responda o reaccione, pero como programadores necesitamos un **mecanismo con el que podamos decirle a nuestra aplicación como debe reaccionar (o qué hacer) ante determinados eventos**.



Por tanto, necesitamos un *mecanismo de aviso*, en lenguaje llano algo tan sencillo como "*hey, acaba de pasar esto*". Pues bien, será **el sistema donde se ejecuta nuestra aplicación quien va a proveernos de estos avisos**, es decir, **el navegador**, y en última instancia **el motor Javascript** que subyace en él. Es por esto que Javascript se fundamenta en un modelo basado en eventos, **toda ejecución comienza a partir de un evento**.

Este mecanismo que nos da el sistema tiene dos partes:

- El **evento** propiamente dicho, el sistema se encarga de dispararlo.
- El código que queremos que se ejecute asociado al evento. A esto se le conoce como **event handler** o **event listener**.

Un *event handler* o *event listener* (acortado *handler* o *listener*) no es más que una **función que asociaremos a un evento** para que sea ejecutada cuando dicho evento tenga lugar. A este proceso se le llama **registrar un event handler/listener**.

Ejemplo

Uno de los eventos más típicos es el click de un botón. Supongamos que en nuestro HTML tenemos un botón:

```
<button id="myButton">Click me</button>
```

Lo primero es crear nuestro event handler, la función que queremos que se ejecute cuando el botón sea *clickado*:

```
var handleButtonClick = () => alert("Button Clicked!");
```

Solo nos queda registrar este event handler:

```
document.getElementById("myButton").addEventListener("click", handleButtonClick);
```

A través de `addEventListener` podemos registrar tantos `handlers` como queramos para un mismo evento. Todos serán ejecutados en caso de que el evento suceda.

Parámetros en event handlers

Supongamos ahora que tenemos un input. Uno de los eventos que nos ofrece y al cual podemos suscribirnos es el evento `onChange`. Este evento es disparado cada vez que el texto del input cambie y el input haya perdido el foco. Si quisieramos registrar un event handler que tuviese acceso al texto, ¿como lo hacemos?

```
<input id="myInput" />


var handleInputChange = () => {
  // ¿Cómo accedo al texto actual del input?
};
```

Recurrimos a los parámetros de los event handlers. Puesto que un handler no es más que una función, admite parámetros de entrada. Es el propio sistema que dispara el evento quien me informa de dichos parámetros y me proporciona información útil que podría ser de ayuda para el código de mi event handler. En palabras más sencillas, cuando el sistema ejecute ese handler, pasará cierta información como parámetros de entrada de la función para que la tengamos disponible dentro de la función:

```
var handleInputChange = event => {
  alert(event.target.value): // Muestra el valor actual del input.
}

document.getElementById("myInput").addEventListener("change", handleInputChange);
```

Otro ejemplo interesante sería transformar el texto que el usuario escribe en el input y ponerlo siempre en mayúsculas. Para ello podemos servirnos del evento `onKeyUp`, que sucede una vez que el usuario ha terminado de presionar una tecla del teclado en el contexto del input:

```
var handleInputKeyUp = event => {
  event.target.value = event.target.value.toUpperCase();
};

document.getElementById("myInput").addEventListener("keyup", handleInputKeyUp);
```

Con los parámetros en los event handlers, el sistema no sólo nos informa de cuando sucede un evento sino que nos aporta todo tipo de información útil acerca de él.

Conclusiones

- Una función es un bloque de código reutilizable destinado a realizar una tarea.
- Es altamente recomendable que las funciones realicen tareas sencillas y concisas, donde su responsabilidad esté muy acotada.
- Las funciones representan piezas pequeñas de un programa.
- Nos ayudan a organizar el código mejor, a estructurarlo y a restarle complejidad puesto que separamos distintos niveles de abstracción en funciones diferentes. Hemos de procurar el mismo nivel de abstracción dentro de una función.
- Las funciones se pueden llamar (ejecutar), tantas veces como queramos. Pueden admitir argumentos de entrada y devolver un valor de retorno.
- Los usuarios interactuan con nuestra aplicación web a través del navegador, quien nos informa sobre eventos que suceden.
- Podemos reaccionar a dichos eventos registrando una función (o tantas como queramos) para que realice una tarea concreta cada vez que dicho evento suceda.

Recursos

- MDN - Tutorial Básico sobre Funciones
- MDN - Tutorial Avanzado sobre Funciones
- MDN - Crea tu propia función
- MDN - Una función devuelve valores
- MDN - Introducción a eventos