Modulo 5. Funciones

¿Qué es una función? Es una pieza de código separada, que constituye un cierto todo lógico cerrado, destinado a realizar una tarea específica. Por lo general, asignamos un nombre a esta pieza de código separada. Con este nombre podemos llamarlo (ejecutarlo) muchas veces en diferentes lugares del programa.

Declaración de funciones

Al igual que con las variables, las funciones deben declararse antes de que podamos usarlas. La sintaxis para la declaración de funciones se ve así:

```
function functionName() {
   código
}
```

Variables locales

Las variables locales son aquellas que se declaran dentro de una función. El alcance de las mismas se limitan solo a la función.

Return

Las funciones que han sido invocadas ejecutan una secuencia de instrucciones contenidas en su cuerpo. El resultado de esta ejecución puede ser un valor determinado que tal vez queramos almacenar en alguna variable. La palabra clave return viene a ayudarnos en este caso. ¿Para qué sirve exactamente `return`?

- Primero, hace que la función termine exactamente donde aparece esta palabra, incluso si hay más instrucciones.
- Segundo, nos permite devolver un valor dado desde dentro de la función al lugar donde fue invocada.

Parametros

En primer lugar, el uso de parámetros en funciones es opcional. Puede haber funciones que no tengan parámetros, como hemos visto en nuestros ejemplos anteriores, al igual que puede haber funciones que no devuelvan valores. Sin

embargo, la mayoría de las veces creamos funciones que tienen parámetros definidos y valores devueltos.

En JavaScript, los parámetros de una función aparecen en su declaración. Estos son nombres separados por comas, colocados entre paréntesis después del nombre de la función. Cada parámetro dentro de una función será tratado como una variable local. Una función cuya definición especifica los parámetros debe invocarse de forma adecuada. Cuando se llama a una función de este tipo, los valores (literales, variables, llamadas de función) deben colocarse entre paréntesis después de su nombre y se tratan como parámetros dentro de la función. Los valores dados durante una llamada se llaman argumentos. Los argumentos, si hay más de uno, se separan por comas y deben pasarse en el mismo orden que los parámetros que definimos en la declaración de la función.

```
function add(first, second) {
  return first + second;
}
```

Sombreado

Los parámetros se tratan dentro de la función como variables locales. Y al igual que las variables locales explícitamente declaradas dentro de una función, sombrean las variables globales del mismo nombre (o más generalmente, las variables del ámbito externo).

Recursividad

La recursividad se refiere a una función a sí misma (pero con un argumento diferente). Una recursividad es un mecanismo que permite simplificar la notación formal de muchas funciones matemáticas y presentarlas de forma elegante. También podemos usar con éxito la recursividad en la programación.

```
function factorial (n) {
    return n > 1 ? n * factorial(n - 1) : 1;
}

console.log(factorial(6)); // -> 720
```

La recursividad se usa comúnmente en la programación. Sin embargo, como con cualquier solución, la recursividad debe manejarse con cuidado. No deberíamos

usarlo donde no podamos estimar la cantidad de llamadas incrustadas. También debemos tener mucho cuidado al formular la condición que interrumpirá las llamadas a la secuencia de funciones: los errores pueden hacer que el programa se suspenda.

Funciones como miembro de primer clase

En JavaScript, las funciones son miembros de primera clase. Este término significa que las funciones pueden tratarse como cualquier dato, que puede almacenarse en variables o pasarse como argumentos a otras funciones. Por ejemplo, podemos declarar la función showMessage y luego almacenarla en la variable `sm`.

```
function showMessage(message) {
    console.log(`Mensaje: ${message}`);
}

let sm = showMessage;
```

Podemos almacenar cualquier función que sea accesible en este ámbito en una variable y usar un operador de llamada de función () para ejecutarla. Podemos verificar que la variable sm ahora es una función usando el operador typeof.

```
sm("¡Esto funciona!"); // -> Mensaje: ¡Esto funciona!
console.log(typeof sm); // -> function
```

Pero es importante recordar que cuando asignamos una función a una variable, no usamos un operador de llamada de función, ya que ejecutaría la función y asignaría el resultado de la función a una variable, y no a la función misma.

```
function doNothing() {
    return undefined;
}

let a = doNothing(); // asignar el resultado de la llamada de función
let b = doNothing;   // asignar una función

console.log(typeof a); // -> undefined
console.log(typeof b); // -> function
```

Expresiones de funciones

Una variable se puede declarar como una función.

```
let myAdd = function add(a, b) {
    return a + b;
}

console.log(myAdd(10, 20)); // -> 30
console.log(add(10, 20)); // -> 30
```

Esta forma de definir una función se llama expresión de función. En este caso, es específicamente una expresión de función con nombre, porque la función tiene un nombre (add). Si hay una expresión de función con nombre, probablemente también habrá una expresión de función sin nombre, o precisamente, anónima. De hecho, simplemente elimina el nombre que sigue a la palabra clave de función para cambiar la función de nombrada a anónima.

```
let myAdd = function(a, b) {
    return a + b;
}

console.log(myAdd(10, 20)); // -> 30
```

Callbacks sincronas

Las funciones que se pasan como argumentos a otras funciones pueden parecer bastante exóticas y no muy útiles, pero de hecho, son una parte muy importante de la programación. Tan importante que incluso tienen su propio nombre.
Son **funciones callback**. Como hemos visto en ejemplos anteriores, una función que recibe una callback como argumento puede llamarla en cualquier momento. Es importante destacar que, en nuestros ejemplos, la callback se ejecuta de forma síncrona, es decir, se ejecuta en un orden estrictamente definido que resulta de dónde se coloca entre las otras instrucciones.

La ejecución **síncrona** es la forma más natural de ver cómo funciona el programa. Las instrucciones posteriores se ejecutan en el orden en que se colocan en el código. Si llama a una función, las instrucciones que contiene se ejecutarán en el momento de la llamada. Si le pasamos otra función a esta función como argumento, y también la llamamos dentro de una función externa, entonces todas las instrucciones mantendrán su orden natural.

Callbacks asincronas

El funcionamiento **asíncrono** de programas es un tema bastante complejo, que depende en gran medida de un lenguaje de programación en particular y, a menudo, también del entorno.

En el caso de que JavaScript del lado del cliente se ejecute en un navegador, se limita a la programación basada en eventos, es decir, la respuesta asincrónica a ciertos eventos. Un evento puede ser una señal enviada por un temporizador, una acción del usuario (por ejemplo, presionar una tecla o hacer clic en un elemento de interfaz seleccionado) o información sobre la recepción de datos del servidor.

Usando las funciones apropiadas, combinamos un tipo específico de evento con una función callback seleccionada, que se llamará cuando ocurra el evento.

Uno de los casos más simples cuando existe una ejecución asíncrona de instrucciones es el uso de la función `setTimeout`. Esta función toma otra función (una callback) y el tiempo expresado en milisegundos como argumentos. La función callback se ejecuta después del tiempo especificado y, mientras tanto, se ejecutará la siguiente instrucción del programa (ubicada en el código después del `setTimeout`).

Por lo tanto, el momento en que se llama a la función callback no está determinado por su orden, sino por un retraso impuesto arbitrariamente. El retraso solo se aplica a la función callback dado a `setTimeout`, mientras que el resto del código aún se ejecuta de forma síncrona.

Funciones setTimeout y setInterval

La función `setTimeout` se utiliza cuando deseas provocar una acción retrasada. Una función similar es `setInterval`. Esta vez, la acción también se realiza con retraso, pero periódicamente, por lo que se repite a intervalos fijos. Mientras tanto, se ejecuta el programa principal y, en el momento especificado, se llama a la función callback proporcionada como argumento para una llamada a setInterval.

Curiosamente, la función `setInterval` devuelve un identificador durante la llamada, que se puede usar para eliminar el temporizador utilizado en ella (y, en consecuencia, para detener la llamada cíclica de la función callback)

Si ejecutamos el código JavaScript en el lado del cliente, en el navegador, siempre está asociado con el sitio web. La ventana en la que se encuentra esta página se representa en el JavaScript del lado del cliente mediante una variable de ventana global. El objeto ventana tiene un método (o su propia función) llamado addEventListener. Esta función te permite registrar una determinada acción para que se realice en respuesta a un evento relacionado con la ventana. Dicho evento puede ser un "clic", que es un solo clic del mouse en cualquier lugar de la página

(hay un conjunto limitado de eventos con nombre asociados con un objeto en particular, al que puede reaccionar). La acción a realizar se pasa al método addEventListener como una función callback.

```
window.addEventListener("click", function() {
    console.log("¡hizo clic!");
});
```

Funciones arrow o flecha

Una **función flecha** es una forma más corta de una expresión de función. Una expresión de función flecha se compone de: paréntesis que contienen de cero a varios parámetros (si está presente exactamente un parámetro, se pueden omitir los paréntesis); una flecha que se ve así `"=>";` y el cuerpo de la función, que puede estar entre llaves {} si el cuerpo es más largo. Si una función flecha tiene solo una sentencia y devuelve su valor, podemos omitir la palabra clave `return`, ya que se agregará implícitamente. Por ejemplo, la función add, que ya conocemos:

```
let add = function(a, b) {
    return a + b;
}
console.log(add(10, 20)); // -> 30
```

Se puede reescribir con funciones arrow

```
let add = (a, b) => {
    return a + b;
}
console.log(add(10, 20)); // -> 30
```

o simplificado aún más (la función tiene solo una sentencia, cuyo valor regresa):

```
let add = (a, b) => a + b;
console.log(add(10, 20)); // -> 30
```

Un ejemplo típico del uso de funciones flecha es el método `forEach`, disponible en datos de tipo `Array`. Hay varias formas de pasar a través de los elementos del arreglo, utilizando diferentes tipos de bucles. El método `forEach` es otro, y

francamente hablando, actualmente el más utilizado. Este método toma como argumento... una función.

Esta función se llamará cada vez para cada elemento del arreglo. Podemos crear cualquier función para este propósito. Hay una condición, que debe tener al menos un parámetro, que será tratado como un elemento visitado del arreglo.