Clase 14

JavaScript Parte 3

JavaScript

Temas:Funciones, Arrow Functions, Callbacks y Closures - Scope

Funciones

¿Qué es una función?

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.

Funciones

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

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

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

Funciones

Generalmente hay 2 razones para declarar funciones:

- Cuando un conjunto de instrucciones se va a usar muchas veces, se declara una sola vez la función con esas instrucciones y se llama muchas veces. por ejemplo parseFloat() y parseInt(), alert( ) y prompt(...) (son funciones de JS no escritas por el programador)
- Por claridad. Para que programa no tenga muchas líneas y sea difícil seguirlo, lo que se hace es separar fracciones de código en una función con un nombre que las identifique y luego llamarlas. Entonces el programa queda mucho más claro. (aunque esa función solo se llame una vez)

Funciones

¿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

Ejemplo

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

        // Ejecución
        tablaMultiplicar(1, 10); // Tabla del 1, calcula desde el 1 hasta el 10
        tablaMultiplicar(5, 10); // Tabla del 5, calcula desde el 1 hasta el 10
```

Funciones

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).

```
// Declaración
function sumar(a, b) {
  return a + b;    // Devolvemos la suma de a y b al exterior de la función
  }

// Ejecución
var resultado = sumar(5, 5); // Se guarda 10 en la variable resultado
console.log(“La suma entre “+ a+” y “+ b+” es:”+ suma;
```

Función Flecha (Arrow Functions)

Una expresión de función flecha es una alternativa compacta a una expresión de función tradicional, pero es limitada y no se puede utilizar en todas las situaciones.

Nota: Cada paso a lo largo del camino es una "función flecha" válida

```
// Función tradicional
function (a){
    return a + 100;
}

// Desglose de la función flecha

// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y las llaves de apertura.
(a) => { return a + 100; }

// 2. Quita los llaves{} del cuerpo y la palabra "return" — el return está implícito.
(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos
a => a + 100;
```

Arrow Functions

Sintaxis

Sintaxis básica
Un parámetro. Con una expresión simple no se necesita return:
param => expression

Varios parámetros requieren paréntesis. Con una expresión simple no se necesita return:
(param1, paramN) => expression

Las declaraciones de varias líneas requieren corchetes y return:

```
param => { let a = 1;
                return a + b; }
```

Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requieren corchetes y return:

```
(param1, paramN) => {
    let a = 1;
    return a + b;   }
```

Arrow Functions

Sintaxis avanzada (todavia no vimos objetos, ni array, asi que solo lo nombramos)

Para devolver una expresión de objeto literal, se requieren paréntesis alrededor de la expresión:
params => ({foo: "a"}) // devuelve el objeto {foo: "a"}

Los parámetros rest son compatibles:
(a, b, ...r) => expression

Se admiten los parámetros predeterminados:
(a=400, b=20, c) => expression

Desestructuración dentro de los parámetros admitidos:
([a, b] = [10, 20]) => a + b; // el resultado es 30

({ a, b } = { a: 10, b: 20 }) => a + b; // resultado es 30

https://www.w3schools.com/js/js_arrow_function.asp

Callback(devolución de llamada)

Las funciones en JavaScript son objetos. Como cualquier otro objeto, puede pasarlos como parámetro. Por lo tanto, en JavaScript podemos pasar una función como argumento de otra función. Esto se llama función de devolución de llamada. Las funciones también se pueden devolver como resultado de otra función.

```
function saludar(nombre) {
  alert('Hola ' + nombre);
}

function procesarEntradaUsuario(callback) {
  var nombre = prompt('Por favor ingresa tu nombre.');
  callback(nombre);
}

procesarEntradaUsuario(saludar);
```

Clousure(Cierre)

Un cierre es una variable o función local que usa otra función y las referencias a la función se devuelven a la función. Es decir, devolvemos una función en una función externa que hace referencia a las variables locales de la función externa. Esto es posible si tenemos funciones anidadas en otra función y devueltas como referencia. En la función interna, podemos usar las variables de la función externa. Debido al alcance de las variables locales, las funciones internas pueden acceder a las variables de la función externa. Cuando devolvemos la función interna en la función externa, las referencias a las variables locales de la función externa todavía están referenciadas en la función interna.

```
const hello = (name)=>{
   const greeting = “Hello”+name;
   return alert(greeting);
}
const helloJane = hello('Jane');
helloJane();
```

helloJane es una función que devuelve hello.

Scope (alcance)

El scope(alcance) determina la accesibilidad (visibilidad) de las variables.

En JavaScript hay dos tipos de alcance:

- Alcance local
- Alcance global

JavaScript tiene un alcance de función: cada función crea un nuevo alcance.

El alcance determina la accesibilidad (visibilidad) de estas variables.

Las variables definidas dentro de una función no son accesibles (visibles) desde fuera de la función.

Scope (alcance)

El scope(alcance) determina la accesibilidad (visibilidad) de las variables.

En JavaScript hay dos tipos de alcance:

- Alcance local
- Alcance global

JavaScript tiene un alcance de función: cada función crea un nuevo alcance.

Las variables definidas dentro de una función no son accesibles (visibles) desde fuera de la función.

Ejemplo

```
// aca no puedo usar la variable carName
function myFunction() {
  var carName = "Volvo";
  // aca si puedo usar la variable carName
}
// aca no puedo usar la variable carName
```

Scope (alcance)

Variables globales de JavaScript

Una variable declarada fuera de una función se convierte en GLOBAL .

Una variable global tiene alcance global : todos los scripts y funciones de una página web pueden acceder a ella.

Ejemplo

```
var carName = "Volvo";
    // aqui si puedo usar carName
function myFunction() {
    // aqui tambien puedo usar la variable carName
}
```

Variables de JavaScript

En JavaScript, los objetos y las funciones también son variables.

El alcance determina la accesibilidad de variables, objetos y funciones de diferentes partes del código.

Scope (alcance)

Automáticamente global

Si asigna un valor a una variable que no ha sido declarada, automáticamente se convertirá en una variable GLOBAL .
Este ejemplo de código declarará una variable global carName, incluso si el valor se asigna dentro de una función.

Ejemplo

```
myFunction();
// aquí  se puede usar carName
function myFunction() {
  carName = "Volvo";
}
```

https://www.w3schools.com/js/js_scope.asp

Argumentos de función

Los argumentos de la función (parámetros) funcionan como variables locales dentro de las funciones.

La instrucción **let** declara una variable de alcance local con ámbito de bloque(block scope), la cual, opcionalmente, puede ser inicializada con algún valor.

Sintaxis

```
let var1 [= valor1] [, var2 [= valor2]] [, ..., varN [= valorN]];
```

Scope (alcance)

let te permite declarar variables limitando su alcance (scope) al bloque, declaración, o expresión donde se está usando. Lo anterior diferencia let de la palabra reservada var , la cual define una variable global o local en una función sin importar el ámbito del bloque.

Alcance (scope) a nivel de bloque con let
Usar la palabra reservada **let** para definir variables dentro de un bloque.

```
for( let i=0 ; i<10; i++){
        console.log(i);
}
console.log(i) ;// aca da error
```