TEMA 4. Funciones

Desarrollo Web en Entorno Cliente.

Profesor: Juan José Gallego García

Índice :

Funciones definidas por el usuario.

Definición de una función.

Las funciones son bloques de código que al ejecutarse, realizan una determinada tarea. Una función ejecuta su código definido cuando es invocada o llamada, ya sea por algún evento, por código o autoinvocada.

En JS existen varias formas de definir una función pero la sintaxis básica es la siguiente :

```
function nombreFuncion (parametro1, parametro2 , ...)
{
  // código a ejecutar
}
```

```
// Llamada a la función …
 nombreFuncion (valor1 , valor2 , ... ) ;
```

Ejemplo :

La primera llamada a la función se da cuando se ejecuta el script, posteriormente podemos invocarla de nuevo pulsando el botón.
Si invocamos una función sin los paréntesis nos devuelve la expresión de la función.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <script>
        function nombreFuncion (parametro1, parametro2) {
          // código a ejecutar;
          var suma = parametro1 + parametro2;
          alert (" La suma es = " + suma);
        }
        nombreFuncion(5,10);
        alert(nombreFuncion); // Devuelve expresión
        </script>
</head>
<body>
<button onclick="nombreFuncion(3,4)">Click</button>

</body>
</html>
```

Podemos hacer que la función retorne un valor para asignarlo a una variable (p.e.) mediante la palabra clave **return .** Al usar **return** se saldrá de dicha función.

```
<!DOCTYPE html>
<html>
<head>
    <script>
        function nombreFuncion (parametro1, parametro2) {
          // código a ejecutar;
          var suma = parametro1 + parametro2;
          return suma;
        }
        var x = nombreFuncion(5,10);
        alert ("Suma =" + x);
        </script>
</head>
<body>
</body>
</html>
```

Definición de función como una expresión.

Se puede definir una función e igualarla a una variable.

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

// Si imprimimos “producto” devuelve la definición de la función no el
resultado.
// Si queremos usar la función …

alert ("El producto es  =" + producto (2,4) );
```

Función autoinvocada.

Podemos hacer que una función sea autoinvocada.

```
(function () {
            var x = "Hola";
            alert (x);
})();
```

Definición de función con constructor. Function

Usando la palabra clave **Function** podemos definir una función a través de un constructor.

```
var producto = new Function("a", "b", "return a * b");
var x = producto(4, 3);
```

Realmente las funciones son objetos y por tanto como ya veremos más adelante poseen propiedades y métodos.

Funciones flecha (arrow).

Con ES6 se puede definir una función usando los operadores "**=>**" como muestra el ejemplo:

```
const a = (x, y) => { return x * y };
alert ("Producto = " + a (2,4));
```

Incluso podemos omitir la palabra clave **return**

```
const a = (x, y) => x * y ;
alert ("Producto = " + a (2,4));
```

```
const a = (x, y) => {
   var b = 4;
   return x * y * b ;
}
alert ("Producto =" + a (2,4));
```

Parámetros de una función.

- En la declaración de parámetros de una función no se usa **var** para declararlos.
- Las funciones de JavaScript no realizan comprobaciones en los tipos pasados como argumentos.
- Tampoco se comprueban el número de argumentos pasados.
- A partir de ES5 se permite dar valores por defecto a los parámetros.

```
function  suma (a=1, b=1) {
// Código ….
}
```

Paso por valor y paso por referencia.

En una función los argumentos de tipo primitivo (number, string, boolean, etc…) se pasan **por valor** (copia) , es decir, la variable original pasada no cambia si se realiza un cambio interno en la función a través de su parámetro.

Por el contrario, si pasamos un **objeto** como argumento a una función se pasará **por referencia** (dirección), es decir, si internamente se hace un cambio a alguna propiedad del objeto a través del parámetro , éste modificará la propiedad del objeto original. (Ver ejemplos)

Paso por valor (se crea una copia)

```
<!DOCTYPE html>
<html>
<head>
    <script>
      var a = 1;
        function nombreFuncion (x) {
          // código a ejecutar;
          x = 3 ;
          return "Valor de x = " + x;
        }
        alert (nombreFuncion(a));
        alert ("Valor de a = " + a );
        </script>
</head>
<body>
</body>
</html>
```

Paso por referencia

```
<!DOCTYPE html>
<html>
<head>
    <script>
      var objetoA = {a:1 , b : "Hola"}
        function nombreFuncion (x) {
          // código a ejecutar;
          x.a = 3 ;
          return "Valor de x = " + x.a;
        }
        alert ("Valor de a = " + objetoA.a );
        alert (nombreFuncion(objetoA));
        alert ("Valor de a = " + objetoA.a );
        </script>
</head>
<body>
</body>
</html>
```

Ámbito o alcance de las variables.

- En JS las variables pueden ser **locales** o **globales.** Las variables **locales** sólo se conocen dentro de la función definida. Pueden ser usada hasta que se cierra dicha función.
- Las variables **globales** pueden ser usadas por cualquier función u objeto del documento o ventana. Permanece activa mientras no se cierre la ventana del navegador.
- Las variables creadas sin “**var**” son definidas como globales incluso dentro de una función.
- Una función puede hacer uso de cualquier variable que esté definida por encima de ella jerárquicamente.
- Si declaramos con **let** una variable su alcance será sólo el del bloque donde se ha declarado mediante **{ }**. Ver ejemplo a continuación.

```
<!DOCTYPE html>
<html>
<head>
 <script>
    var x = 0 ; // Alcance global
    function cambia () {
        var x = 1; // Alcance local a función
    }
    cambia ();
    alert ("Valor de x : " + x );
    {
        let x = 3 ; // Alcance de bloque
    }
    alert ("Valor de x : " + x );
  </script>
</head>
<body>
</body>
</html>
```

Comprobar qué valores de x muestran los mensajes.

Según las buenas prácticas de programación es aconsejable (si es posible) no usar variables globales para evitar cambios por código no deseados. De esta forma nos encontramos con un pequeño problema para dar solución a lo siguiente :

Queremos usar una variable que actúe como un contador definiendo una función. Para solucionarlo usamos una variable global.

```
<script>
   var contador = 0;
   function incrementa () {
       // Código ....
       contador++;
   }
   incrementa ();
   incrementa ();
   alert ("Contador = " + contador); // Imprime 2
 </script>
```

Funciona pero no es lo más correcto porque cualquier código puede cambiar el valor de **contador** sin llamar a la función. Probamos con variable local :

```
<script>
   var contador = 0;
   function incrementa () {
       // Código ....
       var contador = 0
       contador++;
       // Otro código ...
   }
   incrementa ();
   incrementa ();
   alert ("Contador = " + contador); // Imprime 0 ...
 </script>
```

Debería de imprimir 2, pero se está haciendo uso de la variable global. Podríamos definir sólo “**contador**” local pero se iniciaría a “0” con cada llamada a la función.

Función cierre (closure).

Para dar solución al caso del contador (hasta que estudiemos el uso de variables privadas, encapsulamiento, etc..), JS cuenta con las funciones cierres, una función cierre es aquella que estando anidada dentro de otra, puede alcanzar a las variables definidas en su función superior.

La función cierre es privada a la función superior o externa, sólo se puede acceder a ella desde la función externa.

Por tanto combinando función de autoinvocación, y función cierre podemos obtener una solución al contador consiguiendo que una **variable** sea **privada** , es decir, sólo se puede modificar a través de algún método o función definida y no directamente.

```
<!DOCTYPE html>
<html>
<head>
 <script>
    var incrementa = (function () {
        // Código ....
        var contador = 0 ;
        var fun= function (){
            contador++;
            return contador;
        }
    return fun;
    })();
    incrementa();
    incrementa();
    alert ("Contador = " + incrementa()); // Imprime 3 ...
  </script>
</head>
<body>
</body>
</html>
```

Explicación :

1. La variable **incrementa** apunta a las variables declaradas en la función principal o externa y a la definición de la función interna retornada por la autoinvocación.
2. Al finalizar la autoinvocación, **incrementa,** sólo almacena la expresión definida para la función **fun**, pero ha creado un espacio en memoria para **contador (**que permanece**)** y al que puede acceder **fun**.
3. Si ejecutamos **incrementa( )** , aumentamos en una unidad el contador . Podemos comprobar que sin los paréntesis **incrementa** nos devuelve la función en sí (expresión).

Bibliografía

- LibrosWeb
- W3Schools
- Developer Mozilla Docs