TEMA 4. FUNCIONES.

1. INTRODUCCIÓN.

Una función es la definición de un conjunto de acciones encapsuladas que se ejecutan todas juntas.

Las funciones son subprogramas, que podremos reutilizar cuantas veces necesitemos, y que se utilizan para realizar tareas concretas, una función debe realizar una sola tarea.

La creación de funciones nos va a permitir crear bloques de código más pequeños, legibles y reutilizables. También nos van a permitir corregir errores y ampliar nuestros programas de una forma más rápida y eficiente.

Vamos a crear funciones atendiendo a dos criterios, el primero es la **modularización** de nuestro código y el segundo es evitar la **repetición de código**.

En las funciones diferenciamos dos partes el cuerpo de la función y la "llamada", para que una función sea ejecutada tenemos que llamarla mediante su nombre.

Una función puede devolver un valor, aunque no es obligatorio, podemos tener funciones que realicen una tarea que no devuelva un valor.

Una de las sintaxis que podemos utilizar para trabajar con funciones puede ser la siguiente, este sería el cuerpo de la función.

```
function nombreFunción ( [parámetro1]....[parámetroN] ){
// instrucciones
}
```

Si nuestra función devuelve un valor, tenemos que utilizar la palabra reservada **return** y a continuación el valor que devuelve la función.

Una función solo puede devolver un valor y cuando se ejecuta la instrucción return, finaliza la ejecución de la función.

Ejemplo del cuerpo de la función:

```
function nombreFunción ( [parámetro1]....[parámetroN] ){
// instrucciones
return valor;
}
```

Los nombres que puedes asignar a una función tendrán las mismas restricciones que tienen las variables en JavaScript. Es conveniente utilizar nombres de funciones significativos, que indiquen qué acción realiza la función. Puedes usar palabras compuestas como checkMail o addUser, y fíjate que las funciones **suelen llevar un verbo**, puesto que las funciones son elementos que **realizan acciones**.

Normalmente vamos a crear funciones que realicen solo una tarea, esto hará que la función sea más reutilizable.

Para realizar una llamada a una función:

```
nombreFuncion();// Esta llamada ejecutaría la función.
```

En el caso de que la función devuelva un valor tendremos que recogerlo en la llamada o utilizarlo en una expresión.

```
variable=nombreFuncion(); // En este caso la función devolvería un valor que se asigna a
la variable.
```

Siempre que queramos llamar a una función, esta debe haber sido definida anteriormente, es decir, el código de la función debe aparecer en nuestro editor antes de la llamada.

2. PARÁMETROS Y ARGUMENTOS.

Los parámetros son la lista de variables que ponemos cuando se define una función, por ejemplo, en la siguiente función tenemos dos parámetros "a" y "b":

```
function suma(a, b) {
    return a + b;
}
```

Los argumentos son los valores que se pasan a la función cuando ésta es invocada, de esta manera, en el siguiente ejemplo tendríamos que "7", "4" son los argumentos de nuestra invocación a la función:

```
suma(7,4)
```

Los parámetros son datos que necesita la función poder ejecutarse.

Cuando se realiza una llamada a una función, tendremos que indicar (pasar) los datos que necesita la función, es decir, aquellos que hemos definido al crear el cuerpo de la función. Para pasar los parámetros a una función, tendremos que escribir dichos parámetros entre paréntesis y separados por comas, después del nombre de la función. Si la función no tiene parámetros tendremos que escribir los paréntesis vacíos.

Cuerpo de la función

```
function saludar(a,b){
alert("Hola " + a + " y "+ b +".");
}
```

Si llamamos a esa función desde el código:

```
saludar("Martín","Silvia"); //Mostraría una alerta con el texto: Hola Martin y Silvia.
```

Los parámetros que usamos en la definición de la función a y b, no usan la palabra reservada **var o let** para declarar dichas variables. Esos *parámetros a y b serán variables locales a la función*, y se inicializarán automáticamente al llamar a la función, con los valores que le pasemos en la llamada. En el siguiente apartado entraremos más en profundidad en lo que son las variables locales y globales.

Otro ejemplo de función que devuelve un valor:

```
function devolverMayor(a,b){
if (a > b) then
return a;
else
return b;
}
```

Ejemplo de utilización de la función anterior:

```
document.write ("El número mayor entre 35 y 21 es el: " + devolverMayor(35,21) + ".");
```

3. ÁMBITO DE LAS VARIABLES.

Ha llegado la hora de distinguir entre las variables que se definen fuera de una función, y las que se definen dentro de las funciones.

Las variables que se definen fuera de las funciones se llaman **variables globales,** y las que se definen dentro de una función son **variables locales.**

Aunque el uso de la palabra reservada **var o let**, para declarar variables es opcional, es muy recomendable que lo utilicemos siempre, también es muy conveniente utilizar **let** para la declarar las variables, en lugar de **var**.

El alcance de una **variable global** es todo el documento actual que está cargado en la ventana del navegador. Cuando inicializas una variable como variable global, quiere decir que todas las instrucciones de un script (incluidas las instrucciones que están dentro de las funciones), tendrán acceso directo al valor de esa variable. Todas las instrucciones podrán leer y modificar el valor de esa variable global.

En el momento que una página se cierra, todas las variables definidas en esa página se eliminarán de la memoria para siempre. Si necesitas que el valor de una variable persista de una página a otra, tendrás que utilizar almacenamiento en el navegador que te permitan almacenar esa variable en el tiempo.

En contraste a las variables globales, una **variable local será definida dentro de una función**. El alcance de una variable local está solamente dentro del ámbito de la función. Ninguna otra función o instrucciones fuera de la función podrán acceder al valor de esa variable.

Ejemplo de variables locales y globales:

```
/// Uso de variables locales y globales no muy recomendable, ya que estamos empleando el mismo
nombre de variable en global y en local.
let chica = "Aurora"; // variable global
let perros = "Lacky, Samba y Ronda"; // variable global
function demo(){
let chica = "Raquel"; // variable local
document.write( "<br/>"+ perros +" no pertenecen a "+ chica +".");
}
// Llamamos a la función para que use las variables locales.
demo();
// Utilizamos las variables globales definidas al comienzo.
document.write( "<br/>"+ perros +" pertenecen a "+ chica +".");
Como resultado obtenemos:
Lacky, Samba y Ronda no pertenecen a Raquel.
Lacky, Samba y Ronda pertenecen a Aurora.
```

3.1. FUNCIONES ANIDADAS.

Los navegadores más modernos nos proporcionan la opción de anidar unas funciones dentro de otras. Es decir, podemos programar una función dentro de otra función.

La estructura de las funciones anidadas será algo así:

```
function principalA(){
// instrucciones
function internaA1(){
// instrucciones
}
// instrucciones
}
```

```
function principalB(){
// instrucciones
function internaB1(){
// instrucciones
}
function internaB2(){
// instrucciones
}
// instrucciones
}
```

Una buena opción para aplicar las funciones anidadas es cuando tenemos una secuencia de instrucciones que necesitan ser llamadas desde múltiples sitios dentro de una función, y esas instrucciones sólo tienen significado dentro del contexto de esa función principal. En otras palabras, en lugar de romper la secuencia de una función muy larga en varias funciones globales, haremos lo mismo, pero utilizando funciones locales.

Ejemplo de una función anidada:

```
function hipotenusa(a, b){
function cuadrado(x){
return x*x;
}
return Math.sqrt(cuadrado(a) + cuadrado(b));
}
document.write("<br/>La hipotenusa de 1 y 2 es: "+hipotenusa(1,2));
// Imprimirá: La hipotenusa de 1 y 2 es: 2.23606797749979
```

3.2. FUNCIONES PREDEFINIDAS DEL LENGUAJE.

JavaScript disponemos de algunos elementos que necesitan ser tratados de forma global y que no pertenecen a ningún objeto en particular (o que se pueden aplicar a cualquier objeto).

Propiedades globales en JavaScript:

Propiedad	Descripción
Infinity	Un valor numérico que representa el infinito positivo/negativo.
NaN	Valor que no es numérico "Not a Number".
undefined	Indica que a esa variable no le ha sido asignado un valor.

Vamos a ver algunas de las funciones que tiene predefinidas JavaScript, que se pueden utilizar a nivel global en cualquier parte de tu código. Estas funciones no están asociadas a ningún objeto en particular.

Función	Descripción
isNaN()	Determina cuando un valor no es un número.
parseFloat()	Convierte una cadena a un número real.
parseInt()	Convierte una cadena a un entero.

Estas son algunas de las funciones predefinidas, aunque existen muchos más.

4. SINTAXIS DE FUNCIONES.

JavaScript tiene diferentes sintaxis para trabajar con funciones. El resultado de una función no cambia por utilizar una sintaxis u otra, es decir, si escribimos la misma función con diferentes sintaxis el resultado va a ser siempre el mismo independientemente de la sintaxis que utilicemos.

4.1. SINTAXIS "TRADICIONAL"

Es la sintaxis que hemos estado viendo a lo largo del tema, esta sintaxis ha sido utilizado durante mucho tiempo en JavaScript, pero actualmente está en desuso, aunque se siga viendo.

Sintaxis:

```
function nombreFunción ( [parámetro1]....[parámetroN] ){
    // instrucciones
    [return valor;]
}
```

Código de la función:

```
function sumar(a, b){
    return a + b;
}
```

Ejecución:

```
console.log(sumar(3,5)) //El resultado será 8
```

4.2. FUNCIONES ANÓNIMAS.

Una función anónima es aquella que no tiene nombre, es decir, en su definición no le damos nombre.

Para poder utilizarlas tenemos que asignarlas a una variable o una constante que será el nombre de la función. Y que nos permitirá llamar (ejecutarla) y pasarle parámetros.

Sintaxis:

```
const variable = function ( [parámetro1]....[parámetroN] ){
// instrucciones
[return valor;]
}
```

Código de la función:

```
const sumar = function(a, b){
    return a + b;
}
```

Ejecución:

```
console.log(sumar(3,5)) //El resultado será 8
```

4.3. FUNCIONES DE FLECHA

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. La nueva sintaxis omite la palabra reservada function, es anónima y añade una flecha después de los parámetros para indicar que es una función

Sintaxis:

```
const variable = ( [parámetro1]....[parámetroN] )=>{
// instrucciones
[return valor;]
}
```

Código de la función:

```
const sumar = (a, b)=>{
    return a + b;
}
```

Ejecución:

```
console.log(sumar(3,5)) //El resultado será 8
```

La sintaxis de las funciones de flecha puede simplificarse, todavía más:

- ✓ Si solo pasamos un parámetro nos podemos ahorrar los paréntesis

```
const mostrar = a=>{
    return a; // puede que editor te los añada automáticamente
}
```

- ✓ Si no pasamos ningún parámetro es obligatorio poner los paréntesis en los parámetros.

```
const saludo = ()=>{
    console.log("Hola que tal);
}
```

- ✓ Si tenemos una función de una sola línea que en la que devolvemos un valor, nos podemos ahorrar la palabra return y las llaves

  Código de la función:

  ```
  const saludo = ()=>"Hola que tal";
  ```

  Ejecución:

  ```
  console.log(saludo);//mostrará por consola Hola Mundo
  ```

  Código de la función:

  ```
  const sumar = (a, b)=>  a + b;
  ```

  Ejecución:

  ```
  console.log(sumar(3,5)) //El resultado será 8
  ```

5. FUNCIONES DE CALLBACK.

Vamos a hacer una introducción al concepto de funciones de callback. Una función de callback es una función que se pasa como parámetro a otra función.

Es importante tener en cuenta que cuando pasamos un callback solo pasamos la definición de la función y no la ejecutamos en el parámetro. Así, la función contenedora elige cuándo ejecutar el callback.

Veamos un ejemplo:

Cuerpo de la función:

```
const mostrar = (cb)=> {
  console.log("Hola")
  cb();
}
```

Ejecución

```
mostrar(()=>{console.log("Acabo de decirte Hola")});
```

Otro Ejemplo

Cuerpo de la función:

```
const sumatorio = (num1, cb) => {
 console.log(num1 + cb(num1, 5));
};
```

Ejecución

```
sumatorio(10, (num2, num3) => {
  return num2 + num3;
});
```