Funciones y modularización de código

Si ponemos cualquier fragmento de código JavaScript de los que hemos visto hasta ahora en una página, se ejecutará nada más cargar la página, y eso no es lo habitual. Lo normal es que el código JavaScript se vaya activando o ejecutando a medida que el usuario vaya haciendo cosas en la página. Por ejemplo, cuando el usuario pulse el botón de enviar un formulario, debería activarse el código que comprueba que es correcto.

Para evitar la ejecución inmediata del código JavaScript, y de paso poderlo estructurar mejor y dejarlo preparado para que cada trozo de código se active en un momento determinado, usamos funciones. Las funciones son fragmentos de código a los que les ponemos un nombre, de forma que podemos elegir el momento en que ejecutar esas instrucciones simplemente utilizando el nombre que hayamos puesto a la función.

1. Definición de funciones

Estas funciones se crean con la palabra `function` seguida del nombre que queramos y unos paréntesis. El nombre que pongamos a la función debe ser representativo de lo que hace, y normalmente sólo usaremos letras, números y subrayados(_), como si fueran nombres de variables. Después, entre llaves, pondremos todas las instrucciones que queremos que se ejecuten en bloque. Por ejemplo, esta función crea dos variables numéricas, las suma y muestra el resultado:

```
function sumaVariables()
{
    var num1 = 2;
    var num2 = 6;
    var suma = num1 + num2;
    alert("El resultado de la suma es " + suma);
}
```

Si luego queremos hacer que este código se active al hacer clic en un párrafo, por ejemplo, pondríamos esto:

```
<p onclick="sumaVariables()">Bla bla bla bla...</p>
```

Como hemos comentado, el orden en que definamos las funciones en el código de la página (o del archivo JavaScript) es irrelevante. Podemos ponerlas antes o después del bloque de código que las invoca.

1.1. Uso de parámetros

Entre los paréntesis de las funciones podemos introducir, separados por comas, unos datos llamados *parámetros*. Básicamente son variables, a los que se les da valor desde otro lugar (por ejemplo, el contenido principal de la página) y se pueden utilizar dentro de la función. Esta función, por ejemplo, suma los dos datos que recibe como parámetros y muestra el resultado.

```
function sumaDatos(num1, num2)
{
    var suma = num1 + num2;
    alert("El resultado de la suma es " + suma);
}
```

Podríamos llamarla desde otra función, o desde el contenido de la página, con algo como esto:

```
<p onclick="sumaDatos(3, 5)">Bla bla bla bla...</p>
```

> **NOTA**: es habitual en JavaScript que tanto los nombres de funciones como los nombres de parámetros (y variables) sigan la nomenclatura *camel case*: comenzar en minúscula y luego iniciar cada palabra del nombre de la variable en mayúscula: *sumaDatos*, *nombreUsuario*, etc.

> **Ejercicio 1**:
>
> Crea un documento llamado **operaciones_funcion_js.html**. En él, define cuatro párrafos: uno que ponga "Suma", otro que ponga "Resta", otro que ponga "Multiplicación" y otro que ponga "División". Añade un código JavaScript que defina cuatro funciones llamadas *sumar*, *restar*, *multiplicar* y *dividir* que reciban dos números como parámetro y hagan cada una de estas cuatro operaciones con los números que reciben, y muestren su resultado, como en el ejemplo anterior. Estas funciones se deben activar respectivamente cuando hagamos clic sobre el párrafo correspondiente, es decir, al hacer clic en el párrafo que pone "Suma", deberemos activar la función *sumar*, y así sucesivamente. Pasa como parámetro a las funciones los números que quieras.

1.1.1. Número de parámetros y parámetros por defecto

JavaScript es algo laxo en cuanto al paso de parámetros a funciones, de forma que no "se queja" si pasamos parámetros de más o de menos. Si una función necesita dos parámetros y le pasamos sólo el primero, el segundo tomará el valor *undefined*. Si enviamos parámetros de más, el resto se descarta.

Además, podemos dar valores por defecto a los parámetros que lo necesiten, asignándoselo en la declaración de la función:

```
function sumaDatos(num1, num2 = 0)
{
    var suma = num1 + num2;
    alert("El resultado de la suma es " + suma);
}
```

Según la declaración del ejemplo anterior, si llamamos a la función con *sumaDatos(3, 2)* obtendrá un resultado de 5, y si la llamamos con *sumaDatos(3)* obtendrá un resultado de 3, porque al segundo parámetro se le asigna el valor por defecto de 0.

Es importante que los parámetros que tengan valor por defecto se coloquen al final de la secuencia de parámetros, para así poner primero todos los que tienen que tener valor, y después los que pueden no tenerlo, y no alterar el orden inicial de los parámetros en la declaración.

1.2. Tipo de retorno

En ocasiones nos interesa que una función produzca o *devuelva* un resultado, para poderlo utilizar en alguna otra expresión, o almacenarlo en una variable. Por ejemplo, la función de suma anterior podría devolver el resultado de la suma en lugar de mostrarlo con un *alert*, y así nosotros poder decidir qué hacer con ese valor (sacarlo por pantalla, guardarlo en un campo de formulario...).

Para indicar qué valor devuelve una función, empleamos la instrucción `return` seguida del valor que se quiere devolver. Nuestra función de suma anterior podríamos transformarla en esta otra:

```
function sumaDatos(num1, num2)
{
    var suma = num1 + num2;
    return suma;
}
```

> **NOTA**: también podríamos haber puesto directamente `return num1 + num2` y ahorrarnos la variable adicional *suma*.

En este caso, la función no produce ningún resultado directamente visible. Si hacemos algo como esto, no veremos nada en ninguna parte al hacer clic en el párrafo:

```
<p onclick="sumaDatos(3, 5)">Bla bla bla bla...</p>
```

Sin embargo, esto nos puede venir bien para, en cualquier parte del código, utilizar la función y almacenar el resultado producido, o hacer con él lo que queramos. En este ejemplo almacenamos el resultado en una variable, y luego la mostramos por la consola.

```
let resultado = sumaDatos(3, 5);
console.log(resultado);
```

> **Ejercicio 2**:
>
> Crea una página llamada **calcular_precio_js.html** que defina una función llamada `calcularPrecio` con 3 parámetros: el nombre de un producto, su precio y su porcentaje de IVA.
>
> - Los tres parámetros deberán tener un valor por defecto: "Producto genérico", 100 y 21.
> - Internamente, se debe convertir el nombre del producto a string y los otros dos datos a números
> - Si el precio o el porcentaje no son numéricos, se mostrará el mensaje de "Datos incorrectos"
> - Si son válidos, mostraremos por consola el nombre del producto y el precio final incluyendo el IVA
> - Llama a la función varias veces con varios datos de prueba para comprobar su funcionamiento

> **Ejercicio 3**:
>
> Crea una página llamada **tabla_multiplicar_js.html**. Deberá pedirle al usuario un número del 1 al 10 (repetidamente hasta que sea válido) y después, usando una función previamente definida, mostrar la tabla de multiplicar de ese número en el contenido de la página (`document.write`), en forma de lista no ordenada.

1.3. Variables locales y globales

Una variable definida dentro de una función es *local* a la función (no existe fuera de ella), independientemente de si se ha declarado con *var* o con *let*. Una variable declarada fuera de cualquier función es *global* a todas ellas. Cuando existe una variable local con el mismo nombre que una global, prevalece la local sobre la global. Dicho de otro modo, JavaScript comienza a buscar las variables en el ámbito local, y si no las encuentra, pasa al ámbito global.

Modo estricto

Si en algún momento usamos una variable que no hemos declarado, se considera una variable global a toda la aplicación. Esto puede provocar importantes efectos colaterales en un programa, y no es algo deseable.

Para evitar que JavaScript sea tan permisivo, podemos habilitar su modo estricto, incluyendo esta instrucción al principio del fichero o código JavaScript:

```
'use strict';
```

2. Algunas funciones útiles

Además de poder definir nuestras propias funciones, JavaScript incorpora una serie de funciones que nos pueden resultar útiles para ciertos ámbitos, como por ejemplo el manejo de cadenas de texto, o algunos

cálculos matemáticos, entre otras categorías.

2.1. Funciones para manejo de textos

Ya vimos anteriormente que uno de los tipos básicos que maneja JavaScript son los textos, que se podían representar indistintamente entre comillas dobles o simples.

```
let texto = "Hola, buenas";
let otroTexto = 'Otro texto distinto';
```

Además, el propio objeto o variable que almacena el texto dispone de una serie de instrucciones útiles que nos permiten manipularlo o transformarlo. Veremos aquí algunas de las instrucciones más útiles.

- El operador `+` se emplea para concatenar cadenas de texto, enlazando una a continuación de otra.
- Para acceder a un carácter determinado de la cadena, se utilizan los **corchetes** con su posición, empezando en 0. También se puede utilizar la instrucción `charAt` indicando entre paréntesis la posición a obtener (también empezando por la 0).
- La propiedad `length` devuelve el tamaño o número de caracteres de la cadena en cuestión
- La función `indexOf` sirve para buscar un texto dentro de la cadena, devolviendo la posición donde se encuentra, o -1 si no hay coincidencias. También se puede emplear `includes` para ver si un texto está contenido dentro de otro.
- En el caso de querer buscar patrones, podemos emplear la función `search`, que busca la primera ocurrencia que encaje con un patrón.
- Las funciones `endsWith` y `startsWith` determinan si un texto termina o empieza por otro, respectivamente.
- Las funciones `toUpperCase` y `toLowerCase` convierten a mayúscula y minúscula, respectivamente, la cadena entera.
- La función `replace` se emplea para reemplazar un texto con otro. Recibe como parámetros el texto viejo y el nuevo, y reemplaza sólo la primera ocurrencia encontrada.
- Para obtener una subcadena a partir de otra, empleamos la instrucción `substring`. Como parámetros indicamos desde qué posición empezar a obtener la subcadena, y hasta qué posición llegar. Si no se indica este segundo dato, se obtiene hasta el final de la cadena. Alternativamente, se puede usar la función `slice`, con los mismos parámetros y utilidad, y también se puede emplear la función `substr`, que se diferencia de las anteriores en que el segundo parámetro no indica la posición hasta la que llegar, sino cuántos caracteres queremos obtener desde la posición inicial.
- La función `split` sirve para partir una cadena por un delimitador indicado, obteniendo un array con todas las partes encontradas.
- La función `trim` (sin parámetros) elimina posibles espacios adicionales que pueda haber al principio o al final de la cadena. Es especialmente útil cuando se recoge información de un formulario, donde el usuario puede haber introducido espacios accidentalmente.
- Para comparar dos cadenas podemos emplear los operadores de comparación `==`, `!=`, `>`, `<`. Hay que tener en cuenta que esta comparación es alfabética.

Veamos un ejemplo de todas estas funciones. En comentarios se indica el resultado que se produce en cada paso.

```
let texto = "Hola";
texto = texto + " buenas";                    // texto = "Hola buenas"
console.log(texto[0]);                        // H
console.log(texto.charAt(0));                 // H
let tamano = texto.length;                    // 11
let posicion = texto.indexOf("buenas");       // 5
let contiene = texto.includes("bue");         // true
let posicion2 = texto.search(/[aeiou]/);      // 1 (posición de la primera vocal r
let mayus = texto.toUpperCase();              // mayus = "HOLA BUENAS"
let texto2 = texto.replace("bu", "BU");       // texto2 = "Hola BUenas"
let subcad = texto.substring(5, 7);           // subcad = "bu"
let subcad2 = texto.substr(5, 3);             // subcad2 = "bue"
let partes = texto.split(" ");                // partes = ["Hola", "buenas"]
let menor = texto < "Adiós";                  // false
let empieza = texto.startsWith("Hola");       // true
```

Ejercicio 4:

Crea una página llamada **ejercicio_strings_js.html** con las siguientes funciones:

- Función que reciba como parámetro dos textos *t1* y *t2* y devuelva el texto resultado de quitar de *t1* todas las apariciones de *t2*
- Función que reciba como parámetro un nombre de fichero (texto) y determine si es una imagen JPG o PNG.
- Función que reciba como parámetro un texto y determine si es una matrícula válida. Una matrícula se considera válida si está compuesta por 4 dígitos y 3 letras mayúsculas.

Comprueba el funcionamiento de todas estas funciones desde el propio código JavaScript, con los datos de entrada que quieras.

2.2. Funciones matemáticas

A través del elemento `Math` de JavaScript podemos acceder a una serie de funciones matemáticas, y también a algunas constantes numéricas que pueden resultar útiles, como el número PI o el número E.

- `Math.PI` y `Math.E` representan estos dos valores numéricos (*pi* y el número *e*)
- `Math.round(x)` redondea *x* al entero más cercano (por arriba o por abajo)
- `Math.floor(x)` y `Math.ceil(x)` aproximan al entero más cercano por abajo o por arriba, respectivamente.
- `Math.min(n1, n2, n3...)` y `Math.max(n1, n2, n3...)` obtienen el menor/mayor valor de los que se le pasan como parámetro, respectivamente.
- `Math.pow(b, e)` obtiene la potencia de elevar una base *b* a un exponente *e*.
- `Math.abs(x)` obtiene el valor absoluto de *x*

- `Math.random()` genera un número real aleatorio entre 0 y 1 (sin contar el 1)
- `Math.sqrt(x)` calcula la raíz cuadrada de *x*

Veamos algunos ejemplos de uso:

```
let n1 = Math.sqrd(16);             // 4
let n2 = Math.min(2, 6, 1, 7);      // 1
let n3 = Math.pow(3, 2);            // 9
let n4 = Math.ceil(3.12);           // 4
let n5 = Math.abs(-3);              // 3
```

2.3. Trabajando con fechas

Para gestionar fechas JavaScript proporciona el elemento `Date`. Podemos crear fechas de distintas formas:

```
let fecha1 = new Date();                    // Fecha y hora actuales
let fecha2 = new Date(2021, 3, 4);          // 4 de abril de 2021
let fecha3 = new Date("2021-04-04");        // 4 de abril de 2021
let fecha4 = Date.now();                    // Fecha actual en milisegundos desde el

let texto1 = fecha2.toLocaleDateString();   // 4/4/2021
```

3. Opciones avanzadas con funciones

Para terminar este documento, veamos algunas opciones algo más avanzadas que se pueden llevar a cabo con las funciones.

3.1. Funciones anónimas

Las funciones anónimas, como su propio nombre indica, son funciones que no tienen nombre. Al declararlas con la palabra *function* no se les asigna ningún nombre, y lo normal es asignar su contenido a una variable para poderlas usar más adelante, o bien definirlas en el momento en que se van a usar.

Por ejemplo, la siguiente función devuelve la suma de sus dos parámetros, pero esta vez la hemos definido como una función anónima asignada a la variable *suma*:

```
let suma = function(n1, n2) {
    return n1 + n2;
}

...
console.log(suma(4, 3));        // 7
```

A priori, usar una función anónima no aporta muchas ventajas respecto a una función tradicional. Se suelen emplear en el caso de funciones de un solo uso (cuando definimos una función que no se va a usar en ninguna otra parte del código), para poder definir el código de la función justo en el punto donde se va a usar.

3.2. Funciones lambda o *arrow functions*

Las funciones lambda son una forma abreviada de definir funciones, disponible desde ES2015. La idea es definir una función anónima pero con otra sintaxis:

- Primero indicamos los parámetros de la función, entre paréntesis y separados por comas. En el caso de que sólo haya un parámetro, se pueden omitir los paréntesis
- Después añadimos una flecha `=>` (por eso se llaman *arrow functions* o *funciones flecha*)
- Después colocamos, entre llaves, el código de la función.

La función anónima de suma anterior podríamos expresarla así:

```
let suma = (n1, n2) => {
    return n1 + n2;
}
```

Como particularidad de las funciones lambda, si la función que implementamos se limita a devolver un valor, podemos omitir las llaves y la palabra *return*. El ejemplo anterior quedaría así de corto:

```
let suma = (n1, n2) => n1 + n2;
```

Entre las ventajas del uso de las funciones lambda, conseguimos un código más compacto y resumido. Además, son ampliamente usadas en algunos ámbitos, como en la programación en servidor usando el framework Node.js. Igual que ocurre con las funciones anónimas, pueden utilizarse en el mismo lugar en que se definen.

> **Ejercicio 5**:
>
> Repite el ejercicio anterior definiendo las funciones requeridas como funciones lambda.

3.3. Cargar diversos archivos fuente

Además de definir funciones para modularizar nuestro código JavaScript, cuando la aplicación es lo suficientemente compleja, o cuando utilizamos librerías externas como las que veremos más adelante, puede ser necesario dividir el código en varios archivos fuente, de forma que cada uno de ellos incorpora un conjunto de funciones o utilidades. Podemos añadir así diversas etiquetas *script* en nuestro código HTML para incorporar cada uno de estos ficheros:

```
<!DOCTYPE html>
<html lang="es">
    ...
    <script src="fichero1.js"></script>
    <script src="fichero2.js"></script>
    ...
</html>
```

En esta situación, pueden darse dos casos:

- Las funcionalidades incorporadas por todos los archivos son independientes o, dicho de otro modo, no hay dos funciones que se llamen igual en ningún archivo. En este caso no hay ningún problema. Dependiendo de la función que utilicemos en un punto determinado del HTML, se invocará desde un archivo o desde otro.
- Las funcionalidades de los archivos se solapan (hay funciones con el mismo nombre en algunos archivos). En este caso, prevalecerá la función del último archivo incorporado. Por ejemplo, si en el caso anterior ambos archivos *fichero1.js* y *fichero2.js* tienen una función llamada *prueba* entonces se aplicará la de *fichero2.js* (aunque tengan distinto número de parámetros una y otra)