FUNCIONES EN JAVASCRIPT

FUNCIONES

1. **Definición**:

   Una función en JavaScript es un bloque de código o un conjunto de instrucciones que realiza una tarea específica y que puede reutilizarse a voluntad. Por lo tanto, una función es uno de los bloques de construcción fundamentales de JavaScript.

2. Tipos de funciones

Existen diversas formas de crear una función en JavaScript:

- **Por declaración:** Es probablemente la más utilizada y la más fácil de recordar, sobre todo si ya conoces algún otro lenguaje de programación. Consiste en declarar la función con un nombre y sus parámetros de entrada entre paréntesis.

  Esta forma permite declarar una función que existirá a lo largo de todo el código. De hecho, podríamos ejecutar la función sumar() de haberla creado y funcionaría correctamente, ya que Javascript primero busca las declaraciones de funciones y luego procesa el resto del código.

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

  Ejecutar: sumar(5,6); Resultado: //11

- **Por expresión:** Este tipo ha tomado popularidad y consiste básicamente en guardar una función en una variable, para así ejecutar la variable como si fuera una función.

  Con este nuevo enfoque, estamos creando una función **en el interior de una variable**, lo que nos permitirá posteriormente ejecutar la variable (*como si fuera una función*).

  Observa que el nombre de la función (*en este ejemplo: saludar*) pasa a ser inútil, ya que si intentamos ejecutar `sumar()` nos dirá que no existe y si intentamos ejecutar plus() funciona correctamente.

  ```
  const plus = function sumar(a, b){ return a+b;};plus(5,6)
  ```

  ¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las **funciones anónimas** (*o funciones lambda*).

  ```
  const plus = function (a, b){ return a+b;};plus(5,6)
  ```

3. DECLARACIÓN DE FUNCIONES

Una definición de función (también denominada declaración de función o expresión de función) consta de la palabra clave function, seguida de:

- El nombre de la función.
- Una lista de parámetros de la función, entre paréntesis y separados por comas.
- Las declaraciones de JavaScript que definen la función, encerradas entre llaves, { ... }.

```
function square(number) {
  return number * number;
}
```

La función square toma un parámetro, llamado number. La función consta de una declaración que dice devuelva el parámetro de la función (es decir, number) multiplicado por sí mismo. La instrucción return especifica el valor devuelto por la función

Los parámetros primitivos (como un number) se pasan a las funciones por valor; el valor se pasa a la función, pero si la función cambia el valor del parámetro, este cambio no se refleja globalmente ni en la función que llama.

Si pasas un objeto (es decir, un valor no primitivo, como Array o un objeto definido por el usuario) como parámetro y la función cambia las propiedades del objeto, ese cambio es visible fuera de la función

4. EXPRESIONES FUNCTION

Si bien la declaración de función anterior sintácticamente es una declaración, las funciones también se pueden crear mediante una expresión function.

Esta función puede ser anónima; no tiene por qué tener un nombre. Por ejemplo, la función square se podría haber definido como:

```
const square = function (number) {
  return number * number;
};
var x = square(4); // x obtiene el valor 16
```

Sin embargo, puedes proporcionar un nombre con una expresión function. Proporcionar un nombre permite que la función se refiera a sí misma y también facilita la identificación de la función en el seguimiento de la pila de un depurador:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
};


console.log(factorial(3));
```

Las expresiones function son convenientes cuando se pasa una ***función como argumento a otra función***. El siguiente ejemplo muestra una función map que debería recibir una función como primer argumento y un array como segundo argumento.

```
function map(f, a) {
  let result = []; // Crea un nuevo array
  let i; // Declara una variable
  for (i = 0; i != a.length; i++)
            result[i] = f(a[i]);
  return result;
}
```

En el siguiente código, la función recibe una función definida por una expresión de función y la ejecuta por cada elemento del array recibido como segundo argumento.

```
function map(f, a) {
```

```
  let result = []; // Crea un nuevo array
  let i; // Declara una variable
  for (i = 0; i != a.length; i++) result[i] = f(a[i]);
  return result;
}

const f = function (x) {
  return x * x * x;
};

let numbers = [0, 1, 2, 5, 10];
let cube = map(f, numbers);
console.log(cube);
```

La función devuelve: [0, 1, 8, 125, 1000].

En JavaScript, una función se puede definir en función de una condición. Por ejemplo, la siguiente definición de función define myFunc solo si num es igual a 0:

```
var myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = "Toyota";
  };
}
```

5. LLAMAR FUNCIONES:

Definir una función no la ejecuta. Definirla simplemente nombra la función y especifica qué hacer cuando se llama a la función. Llamar a la función en realidad lleva a cabo las acciones especificadas con los parámetros indicados. Por ejemplo, si defines la función square, podrías llamarla de la siguiente manera: ***square(5);***

Una función se puede llamar a sí misma. Por ejemplo, aquí hay una función que calcula factoriales de forma recursiva:

```
function factorial(n) {
  if (n === 0 || n === 1) return 1;
  else return n * factorial(n - 1);
}
```

6. PARÁMETROS OPCIONALES:

Son parámetros que no es obligatorio pasarle un valor ya que se lo puedo asignar en la cabecera de la función

Ejemplo:

```
function suma_y_muestra(numero1,numero2,mostrar=false) {

        resultado = numero1 + numero2;
        if (mostrar==false){
                console.log("El resultado es " + resultado); }
```

```
        else { document.write("El resultado es " + resultado); }
}

---------- Para invocarla ----------
for (var i=1;i<10;i++){
 console.log(i);
 suma_y_muestra(i,8); // Muestra el resultado por la consola
}
```

7. FUNCIONES ANÓNIMAS

Las funciones anónimas o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
    return "Hola";
};

Ejecución:
saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

Al escribir saludo(), estamos ejecutando la variable, es decir, ejecutando la función que contiene la variable. Sin embargo, en la línea anterior al escribir saludo, hacemos referencia a la variable (sin ejecutarla, no hay paréntesis) y nos devuelve la función en sí.

La diferencia fundamental entre las funciones por declaración y las funciones por expresión es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si «ejecutamos la variable» antes de declararla, nos dará un error.

8. FUNCIONES CALLBACKS

A grandes rasgos, un callback (llamada hacia atrás) es pasar una función B por parámetro a una función A, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función:

```
// fB = Función B
const fB = function () {
  console.log("Función B ejecutada.");
};

// fA = Función A
const fA = function (callback) {
  callback();
};

fA(fB); Ejecutar
```

Los callbacks aseguran que una función no se va a ejecutar antes de que se complete una tarea, sino que se ejecutará justo después de que la tarea se haya completado. Nos ayuda a desarrollar código JavaScript asíncrono y nos mantiene a salvo de problemas y errores

Ejemplo: (*No es necesario nombrar a una función callback con el nombre callback)*

```
function crearCita(cita, callback){
  var miCita = "Como yo siempre digo, " + cita;
  callback(miCita); // 2
```

```
}

function logCita(cita){
  console.log(cita);
}

crearCita("come tus vegetales!", logCita); // 1

// Resultado en la consola:
// Como yo siempre digo, come tus vegetales!
```

En el ejemplo anterior, crearCita es la función de orden-superior, la cual acepta dos argumentos, el segundo es el callback. La función logCita se está pasando como nuestra función callback.
Cuando ejecutamos la función crearCita:
(1), observa que no estamos agregando paréntesis a logCita al pasarla como argumento. Esto se debe a que no queremos ejecutar nuestra función callback de inmediato, simplemente queremos pasar la definición de la función a la función de orden-superior para que pueda ejecutarse más tarde.

Además, debemos asegurarnos que si la función callback que pasamos espera argumentos, debemos proporcionarlos al ejecutarla

9. FUNCIONES 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.

Diferencias y limitaciones:

- No tiene sus propios enlaces a this o super y no se debe usar como métodos.
- No se puede utilizar como constructor.

Observa, paso a paso, la descomposición de una "función tradicional" hasta la "función flecha" más simple:

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

// Desglose de la función flecha
```

- Paso 1. Elimina la palabra "function" y coloca la flecha entre el argumento y el corchete de apertura.

```
(a) => {
  return a + 100;
}
```

- Paso 2 Quita los corchetes del cuerpo y la palabra "return" — el return está implícito.

```
(a) => a + 100;
```

- Paso 3 Suprime los paréntesis de los argumentos

```
a => a + 100;
```

Nota: Como se muestra arriba, los { corchetes }, (paréntesis) y "return" son opcionales, pero pueden ser obligatorios.

Por ejemplo, si tienes varios argumentos o ningún argumento, deberás volver a introducir paréntesis alrededor de los argumentos:

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

// Función flecha
(a, b) => a + b + 100;

// Función tradicional (sin argumentos)
let a = 4;
let b = 2;
function (){
  return a + b + 100;
}

// Función flecha (sin argumentos)
let a = 4;
let b = 2;
() => a + b + 100;
```

Del mismo modo, si el cuerpo requiere líneas de procesamiento adicionales, deberás volver a introducir los corchetes Más el "return" (las funciones flecha no adivinan mágicamente qué o cuándo quieres "volver"):

```
// Función tradicional
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}

// Función flecha
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

Y finalmente, en las funciones con nombre tratamos las expresiones de flecha como variables

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

// Función flecha
let bob = (a) => a + 100;
```

Sintaxis
Sintaxis básica: Un parámetro. Con una expresión simple no se necesita return:

```
param => expression;
(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;
};
```

SECUENCIA/DESARROLLO

1. ***Ejercicio1: Revisar y ejecutar ejemplo_callback.js***
2. ***Ejercicio 2: Revisar y ejecutar ejemplo_callback_esPrimo.js***
3. ***Ejercicio 3: Realizar el ejercicio de la práctica anterior utilizando funciones. Debes pedir 6 números por pantalla, almacenar en un array y realizar las siguientes funciones***.

Para cada una de las funciones que aparecen a continuación debes utilizar una función

- *Mostrar el array en el cuerpo de la página y en la consola:* ***mostrar***
- *Ordenarlos y mostrarlos:* ***orden***
- *Invertir su orden y mostrarlos:* ***ordenInv***

  En estos dos casos debes utilizar la función mostrar, la cual se recibe como argumento en la función ***orden*** *y* ***ordenInv***
- *Mostrar el número de elementos que tiene el array:* ***tamanno***
- *Búsqueda de un valor introducido por el usuario que nos diga si lo encuentra y su índice:* ***posicion***
- *Sumar todos los números pares:* ***sumarPar****. Para realizar esta acción debes utilizar la funcion* ***esPar*** *la cual se envía como argumento*