Unidad 4
Funciones

CESUR
Tu Centro Oficial de FP

1. Introducción

FUNCIÓN – grupo de instrucciones que tienen un nombre identificativo. Pueden recibir valores de entrada (argumentos) y devuelven un valor de salida.

Ventajas de la utilización de funciones:

1. Permiten dividir un programa complejo en distintas partes, esto facilita tanto la programación como el depurado.
2. Permiten la reutilización de código, ya que la misma función se puede utilizar en distintos programas.

Las funciones permiten reutilizar el código, mejorar la eficiencia, aumentar la legibilidad y reducir los costes de mantenimiento de los programas.

1. Introducción

Ejemplo 1: FUNCIONES

Piensa en el proceso de inicio de sesión de una aplicación web cualquiera y crea una descomposición del problema en problemas más pequeños que podrías implementar con funciones.

SOLUCIÓN:

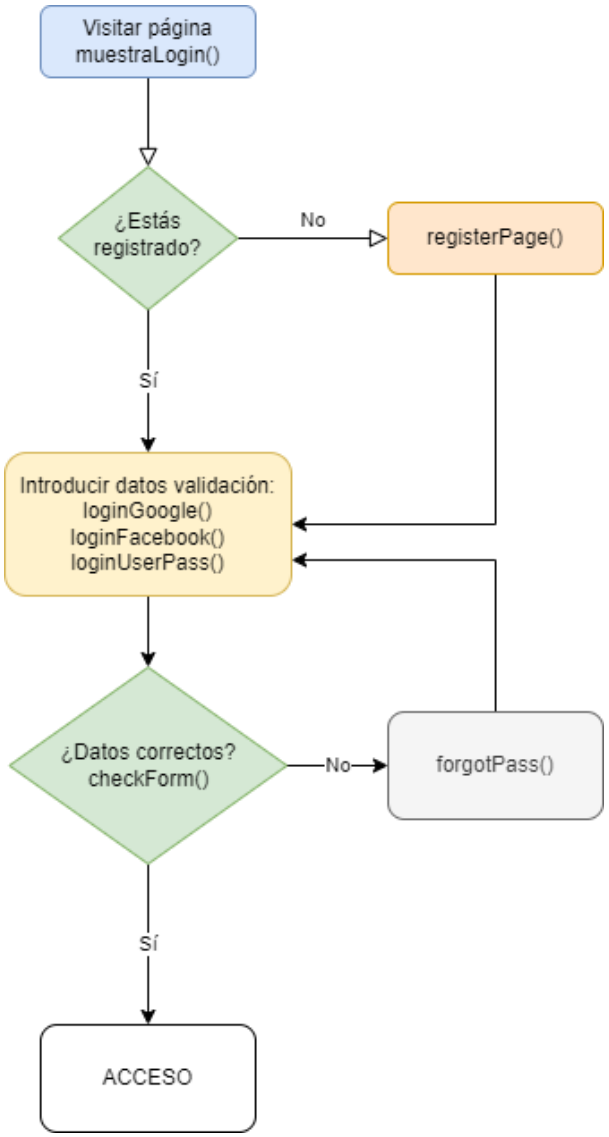
1) Ir a la página y mostrar el Inicio de sesión – función **muestraLogin()**

2) Permitir registrarte si aún no lo estás – función **registerPage()**

3) Utilizar procesos de login de servicios externos (Google, Facebook, etc.) – funciones **loginGoogle()**, **loginFacebook()**, **loginUserPass()** – en las dos primeras funciones se implementaría el acceso a la API de dichos servicios y en la tercera nuestro propio proceso de autenticación

4) Comprobar que se han introducido correctamente los datos – función **checkForm()**

5) Permitir recuperar la contraseña – función **forgotPass()**



1. Introducción

En JS tenemos dos tipos de funciones:

1) **FUNCIONES PREDEFINIDAS** – funciones que se incluyen por defecto en JS. Por ejemplo: alert(); write(); Number(); prompt();

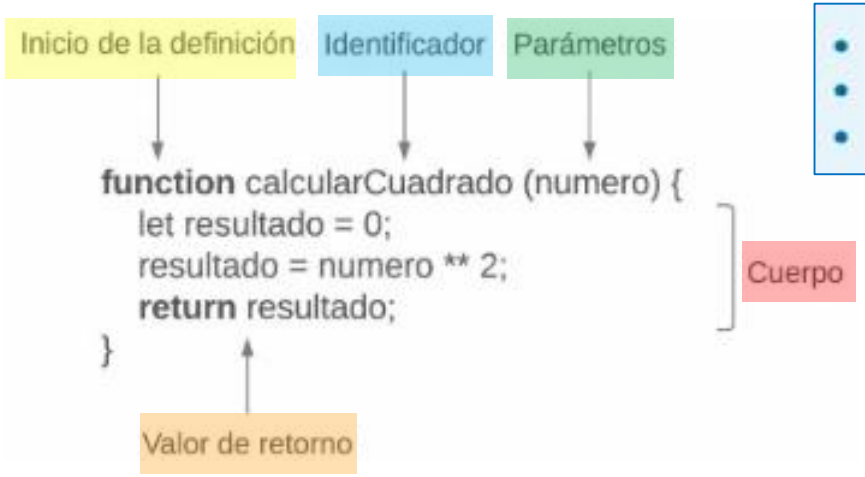
Funciones predefinidas en JS
https://www.tutorialspoint.com/javascript/javascript_builtin_functions.htm

2) **FUNCIONES DEFINIDAS POR EL USUARIO** – funciones que crea el programador para adaptarlas a sus programas y que se pueden reutilizar.

Funciones JS
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions

2. Definición de una función

La estructura de una función definida por el usuario, en general, suele ser:



- Una función puede tener cero, uno o varios parámetros.
- El código de la función estará dentro de las llaves { y }.
- La función devolverá el resultado con la sentencia return.

Donde:

1) **Inicio de la definición** – una función siempre comienza con la palabra clave **function**.
2) **Identificador** – es el nombre que se le asigna a la función y que se utiliza para ejecutarla.
3) **Parámetros** – datos exteriores que se pasan a la función para que trabaje con ellos.
4) **Cuerpo** – bloque de instrucciones que se ejecutan cuando se llama a la función.
5) **Valor de retorno** – es el valor opcional que devuelve la función al exterior (al punto del programa desde el que se la invocó).

2. Definición de una función

Para poder llamar (invocar) a una función es necesario que se haya definido previamente.

```
function mensajeAlerta() {
        alert("Cuidado, errores en el formulario");
}
```

La definición se inicia con la palabra reservada **function**.

Después le sigue el identificador de la función - **mensajeAlerta**.

Sigue una lista de parámetros – en este caso vacía.

Y continúa con el cuerpo de la función donde se incluyen las instrucciones que la componen – en este caso el alert.

No se ha indicado un valor de retorno porque en este caso no tiene sentido, ya que simplemente su cometido es mostrar una ventana del navegador con el mensaje correspondiente.

Si ejecutamos este código, no se ve nada, ya que las funciones, además de **definirlas**, hay que **invocarlas**.

3. Invocación de una función

Para invocar a una función (llamarla) se utiliza su identificador junto con los parámetros que se desea pasarle.

```
mensajeAlerta();
```

Ahora si ejecutamos el código, si se muestra el mensaje:

NOTA: aunque una función se haya definido sin parámetros, hay que escribir los paréntesis igualmente.

```
function mensajeAlerta(mensajeExterior) {
        alert(mensajeExterior);
}
let mensajePersonalizado = "Error definido por el usuario";
mensajeAlerta(mensajePersonalizado);
```

Esta función recibe un parámetro desde el exterior, el mensaje que va a mostrar.

3. Invocación de una función

Una función con valor de retorno sería:

```
function raizCuadrada(numero) {
        return (Math.sqrt(numero));
}
console.log(raizCuadrada(4)); // muestra 2
```

Define la **función raizCuadrada** que recibe un parámetro llamado **numero** y devuelve la raíz cuadrada de ese número con la ayuda de la **librería Math** (integrada en JS).

Para ejecutar la función se la invoca incluyendo entre paréntesis el número del que queremos hacer el cálculo (número 4).

Como la función tiene definido un valor de retorno, se puede tratar como una variable que contiene un valor y se puede utilizar con expresiones:

```
console.log(raizCuadrada(4+5)); // muestra 3
console.log(raizCuadrada(4)+raizCuadrada(9)); // muestra 5
console.log(raizCuadrada(raizCuadrada(16))); // muestra 2
console.log(raizCuadrada(11-3)); // muestra 2.8284271247461903
```

3. Invocación de una función

Por otra parte, el cuerpo de una función puede llamar a otras funciones, ya sean predefinidas o creadas por el programador.

```
function raizCuadrada(numero) {
        return (Math.sqrt(numero));
}
function calcularMayor(vector) {
        let mayor = raizCuadrada(vector[0]);
        for (let i=0; i<vector.length; i++) {
                if (raizCuadrada(vector[i]) > mayor)
                        mayor = raizCuadrada(vector[i]);
        }
        return mayor;
}
console.log(calcularMayor([64,128,4,1024,16]));
// muestra 32
```

La **función raizCuadrada** se utiliza dentro de otra **función calcularMayor** cuyo cometido es calcular la mayor raíz cuadrada de los elementos de un array.

4. Valor de retorno de una función

El valor de retorno de una función es el dato más importante de la misma, puesto que es el resultado que la función devuelve a quien la invocó.

NOTA: debemos tener cuidado con las expresiones que se indican en el return de una función, ya que, en ocasiones, la instrucción que invocó a la función está esperando un tipo de dato en concreto que no coincide con el tipo de dato que está devolviendo la función.

```
function esPar(numero) {
        if (numero%2 == 0)
                return true;
        else
                return false;
}
```

Muestra una función que devuelve true si el número que recibe es par, o false si es impar.

```
if (esPar(4))
        console.log("El número es par");
else
        console.log("El número es impar");
```

Invocación correcta. El resultado de evaluar la condición de if debe ser un valor booleano.

```
console.log(esPar(4)+5);
```

Invocación incorrecta. La operación que se realiza es una suma entre un booleano (resultado de la función (true) y un número (5), lo que no tiene sentido

4. Valor de retorno de una función

Si tenemos una función con cientos de líneas de código con múltiples **return** en distintas partes del mismo, la legibilidad y el seguimiento del flujo de ejecución del programa se convierten en tareas extenuantes.

Se recomienda que se incluya un solo **return** en cada función

La forma más adecuada de escribir la función anterior sería:

```
function esPar(numero) {
        if (numero%2 == 0)
                return true;
        else
                return false;
}
```

Función con varios **return**.

```
function esPar(numero) {
        let resultado = false;
        if (numero%2 == 0)
                resultado = true;
        return resultado;

}
```

Función con un único **return**.

4. Valor de retorno de una función

Ejemplo 2: ECUACIÓN SEGUNDO GRADO I

Crea un programa que incluya una función que reciba tres parámetros (a, b, c) y devuelva un array con las soluciones de una ecuación de segundo grado.

NOTA: hay que tener en cuenta que **parcial = $b^2 - 4*a*c$**

- Si **parcial > 0** existen dos soluciones reales y distintas (fórmula x)

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Si **parcial = 0** existen una única solución real (los resultados de la fórmula x son iguales)
- Si **parcial < 0** no existen soluciones reales

4. Valor de retorno de una función

Ejemplo 2: ECUACIÓN SEGUNDO GRADO II (con if anidados)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            var soluciones = new Array();
            //Definimos la función
            function ecuacio2Grado(a,b,c){
                let parcial = (b*b) - (4*a*c);
                if (parcial < 0){
                    document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' no tiene solución real');
                } else {
                    soluciones[0]=(-b + Math.sqrt(parcial))/(2*a);
                    soluciones[1]=(-b - Math.sqrt(parcial))/(2*a);
                    if (soluciones[0]==soluciones[1]){
                        document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' tiene una única solución:<br>');
                        document.write(soluciones[0]);
                    } else{
                        document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' tiene las soluciones:<br>');
                        document.write(soluciones); }
                    }
                }
                //Introduce 3 valores
                A = parseInt(prompt('Ingresar el primer valor:'));
                B = parseInt(prompt('Ingresar el segundo valor:'));
                C = parseInt(prompt('Ingresar el tercer valor:'));
                ecuacio2Grado(A,B,C); //Llamada a la función
        </script>
    </head>
    <body>
    </body>
</html>
```

2, 3, 4
1, 5, 6
4, 4, 1

4. Valor de retorno de una función

Ejemplo 2: ECUACIÓN SEGUNDO GRADO III (con switch)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            var soluciones = new Array();
            //Definimos la función
            function ecuacio2Grado(a,b,c){
                let parcial = (b*b) - (4*a*c);
                soluciones[0]=(-b + Math.sqrt(parcial))/(2*a);
                soluciones[1]=(-b - Math.sqrt(parcial))/(2*a);
                switch (true){
                case (parcial < 0):
                    document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' no tiene solución real');
                    break;
                case (parcial > 0):
                    document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' tiene las soluciones:<br>');
                    document.write(soluciones);
                    break;
                default:
                    document.write('La ecuacion de segundo grado: '+ A +'x2 + '+ B +'x + '+ C +' tiene una única solución:<br>');
                    document.write(soluciones[0]);
                    break;
                }
            }
            //Introduce 3 valores
            A = parseInt(prompt('Ingresar el primer valor:'));
            B = parseInt(prompt('Ingresar el segundo valor:'));
            C = parseInt(prompt('Ingresar el tercer valor:'));
            ecuacio2Grado(A,B,C);    //Llamada a la función
        </script>
    </head>
    <body>
    </body>
</html>
```

2, 3, 4
1, 5, 6
4, 4, 1

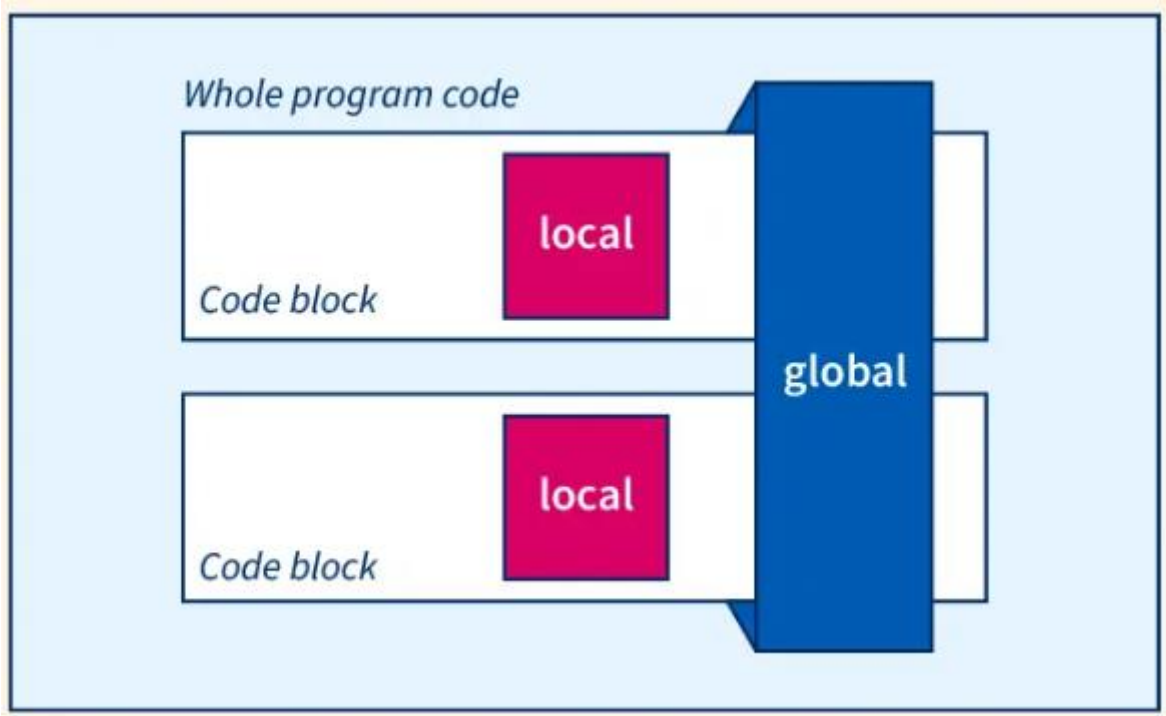
5. Ámbito de las variables

ÁMBITO DE LA VARIABLE – hace referencia a las zonas del programa donde una variable es accesible (visible).

Tipos de ámbitos de las variables:

1) **VARIABLE LOCAL** – la variable sólo es visible desde el interior de una función.

2) **VARIABLE GLOBAL** – la variable es visible desde cualquier parte del programa.



5. Ámbito de las variables

```
var mensaje = "Fuera de la función";
function mostrarAnuncio() {
        var mensaje = "Dentro de la función";
        console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```

```
Dentro de la función
Fuera de la función
>
```

Tenemos dos variables que se llaman igual (**mensaje**).

La primera variable es **global**, por tanto, se puede acceder desde cualquier parte del programa.

La segunda variable es **local**, solo es accesible desde el interior de la función.

La variable mensaje del interior de la función no machaca el valor de la variable mensaje del exterior de la función, esto es debido a los ámbitos.

```
var mensaje = "Fuera de la función";
function mostrarAnuncio() {
        console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```

```
Fuera de la función
Fuera de la función
>
```

Si eliminamos la variable mensaje del interior de la función, vemos que a la variable **global** mensaje se puede acceder desde el interior de la función.

```
function mostrarAnuncio() {
        var mensaje = "Dentro de la función";
        console.log(mensaje);
}
mostrarAnuncio();
console.log(mensaje);
```

```
Dentro de la función
▸Uncaught ReferenceError: mensaje is not defined
    at funciones.js:7:13
>
```

Si eliminamos la variable mensaje del exterior y se deja la del interior, variable **local**, el intérprete lanza un error, que la variable mensaje no está definida en el ámbito global del programa, aunque si lo esté en el ámbito local de la función.

Recuerda

No se recomienda la utilización de variables globales en los programas. A menudo son muy problemáticas a la hora de depurarlos. Al ser accesibles desde cualquier punto del programa, las puede modificar cualquier función, por lo que localizar un error causado por una variable global suele convertirse en un auténtico dolor de cabeza.

6. Parámetros

PARÁMETROS – valores que permiten la comunicación entre las funciones.

El carácter de los parámetros depende de si su modificación en el interior de la función afecta a su valor en el exterior de la función.

```
var numero1 = 7;
var numero2 = 8;
function menor(primero,segundo) {
        var elmenor = primero;
        if (segundo < primero)
                elmenor = segundo
        return elmenor;
}
console.log(menor(numero1,numero2));
```

Aquí se crean dos variables en el programa principal (**numero1** y **numero2**).

Se define una función que calcula el menor de los dos números que recibe en los parámetros **primero** y **segundo**.

Se llama a la función con los parámetros **numero1** y **numero2,** estos valores se copian en **primero** y **segundo**.

La función devuelve mediante la variable **elmenor**, el valor más pequeño de las dos variables iniciales.

Si en el interior de la función se modificara el valor de **primero** o **segundo**, las variables globales **numero1** y **numero2** seguirían teniendo sus propios valores iniciales.

6. Parámetros

```
var numero1 = 7;
var numero2 = 8;
function menor(primero,segundo) {
        primero = 10;
        segundo = 21;
        var elmenor = primero;
        if (segundo < primero)
                elmenor = segundo
        return elmenor;
}
console.log(menor(numero1,numero2)); // muestra 10
console.log(numero1); // muestra 7
console.log(numero2); // muestra 8
```

Aquí se crean dos variables en el programa principal (**numero1** y **numero2**).

Se define una función que calcula el menor de los dos números.

Se crean dos variables **primero** y **segundo** dentro de la función, con sus correspondientes valores.

Se llama a la función con los parámetros **numero1** y **numero2**.

Devuelve el resultado de la función con **elmenor,** haciendo los cálculos con los valores de **primero** y **segundo** que se han definido dentro de ella.

Después se muestran los valores de **numero1** y **numero2**, que son los globales que se han establecido al principio del programa.

6. Parámetros

Todos los parámetros que hemos trabajado hasta ahora son parámetros por valor.

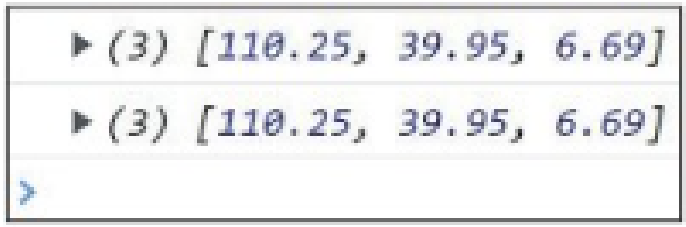
JS no permite definir parámetros por referencia, como ocurre en otros lenguajes.

Lo que permite JS es que ciertas variables, cuando hacen referencia a un objeto, su propio identificador es la referencia.

RECORDATORIO: ASIGNACIÓN DE ARRAYS

El identificador de un array no es donde se almacena la estructura de datos, sino que es una referencia que apunta a la estructura de datos.

```
let sinIVA = [20.45,39.95,6.69];
let conIVA = sinIVA;
conIVA[0] = 110.25;
console.log(sinIVA);
console.log(conIVA);
```



Cuando se asigna identificadores a los arrays, lo que ocurre es que dos identificadores apuntan a la misma estructura de datos, por lo que cualquier modificación que se haga utilizando cualquiera de los dos identificadores estará afectando a la misma estructura de datos.

6. Parámetros

PARÁMETROS POR DEFECTO

En JS podemos definir funciones con parámetros que tengan un valor predeterminado.

Si en la invocación no se especifican valores para esos parámetros, tomaron los valores que se les ha asignado por defecto.

Si en la invocación, se especifican valores para esos parámetros, dichos valores sustituirán a los valores predeterminados.

```
function dividir(numerador, denominador=1) {
        return (numerador/denominador);
}
console.log(dividir(4)); // muestra 4
console.log(dividir(4,2)); // muestra 2
console.log(dividir()); // muestra NaN
```

La función dividir, tiene un parámetro predeterminado denominador.

En la primera llamada a la función se divide 4 por 1 (predeterminado).

En la segunda llamada a la función se divide 4 por 2 (el valor predeterminado es sustituido por el valor de llamada).

En la tercera llamada a la función, como no se especifica un valor para numerador se sustituye por **undefined**. **En JS todos los parámetros tienen un valor predeterminado: undefined**

6. Parámetros

PARÁMETROS VARIABLES

En JS hay otra forma de utilizar parámetros que es utilizar un número variable de ellos.

No es necesario especificar en la definición de la función cuántos son ni como se llaman los parámetros que utiliza. Lo que se utiliza es un objeto de JS llamado **arguments**.

arguments – es un array que contiene todos los argumentos que se han indicado en la invocación de la función.

arguments.length nos dice el número de parámetros que se han incluido.

NOTA: este método es propenso a errores si no se conoce bien el programa, por lo que se debe utilizar con precaución.

```
function sumaTodo() {
        let sum = 0;
        for (let i = 0; i < arguments.length; i++)
                sum += arguments[i];
        return sum;
}
x = sumaTodo(11, 22, 33, 44, 55);
console.log(x);
```

En la definición de la función sumaTodo() no se han especificado parámetros; sin embargo, es capaz de gestionar una invocación con cinco parámetros: sumaTodo(11,22,33,44,55).

6. Parámetros

La otra forma de trabajar con parámetros variables es utilizar el **operador de propagación o spread (…)**

```
function sumaTodo(...parametros) {
        let sum = 0;
        for (let i = 0; i < parametros.length; i++)
                sum += parametros[i];
        return sum;
}
x = sumaTodo(11, 22, 33, 44, 55);
console.log(x);
```

En este caso, el operador de propagación convierte la lista de parámetros usados en la invocación de la función sumaTodo(11,22,33,44,55), en un array llamado parametros donde cada posición almacena un parámetro.

Ejemplo 3: SUMA DE ENTEROS

Crea un programa que te pida dos números e incluya una función que retome la suma de dos enteros.

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>Actividad 1</title>
6           <script>
7               function sumar(x, y) {
8                   let s = x + y;
9                   return s;
10              }
11              x = parseInt(prompt('Ingresar el primer valor:'));
12              y = parseInt(prompt('Ingresar el primer valor:'));
13              document.write("La suma de "+ x + " + " + y + " = "+ sumar(x, y));
14          </script>
15      </head>
16      <body>
17      </body>
18  </html>
```

Ejemplo 3: SUMA DE ENTEROS 2

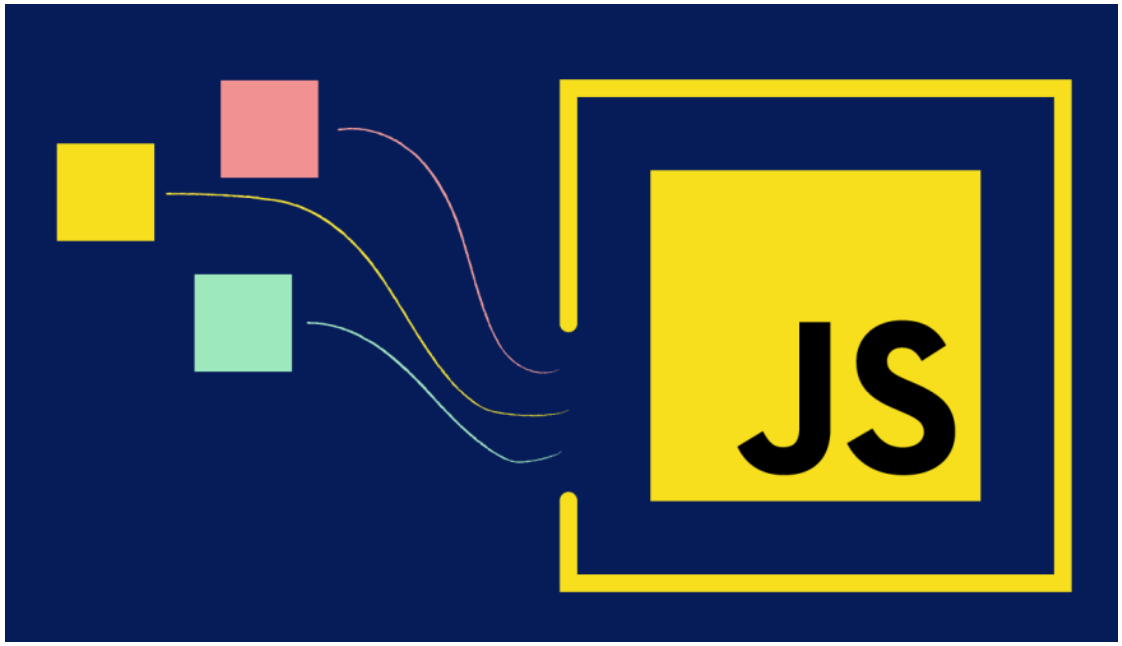
Modifica la función del programa anterior utilizando el objeto arguments que se crea cada vez que llamamos a la función.

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>Actividad 1</title>
6           <script>
7               function sumar(x, y) {
8                   let s = arguments[0] + arguments[1];
9                   return s;
10              }
11              x = parseInt(prompt('Ingresar el primer valor:'));
12              y = parseInt(prompt('Ingresar el primer valor:'));
13              document.write("La suma de "+ x + " + " + y + " = "+ sumar(x, y));
14          </script>
15      </head>
16      <body>
17      </body>
18  </html>
```

7. Tipos de funciones

En JS tenemos diferentes formas de definir y utilizar funciones:

1) Funciones por declaración
2) Funciones por expresión
3) Funciones como objetos
4) Funciones anónimas
5) Callbacks
6) Funciones autoejecutables
7) Funciones flecha



7. Tipos de funciones

1) FUNCIONES POR DECLARACIÓN

Es la forma más común de definir funciones, la más sencilla de entender y la que más se parece a la forma de definir funciones en otros lenguajes de programación.

Las funciones definidas por declaración existen y están disponibles a lo largo de todo el código del programa.

Además, pueden invocarse incluso antes de ser definidas, puesto que JS recorre el código para buscar sus definiciones y luego ejecuta todo el código secuencialmente.

```
let resultado = multiplicar(7,5);
function multiplicar(a,b) {
        return a*b;
}
console.log(resultado); // muestra 35
```

2) FUNCIONES POR EXPRESIÓN

En este tipo de funciones lo que se pretende es relacionar la definición de una función con el identificador de una variable.

Es como, almacenar una función en una variable, de manera que al utilizar esa variable lo que se hace es invocar a su función almacenada.

```
const bienvenido = function sesionIniciada() {
        console.log("Bienvenido de nuevo");
};
bienvenido();
sesionIniciada();
```

Al ejecutar este código, vemos que el identificador de la función (**sesionIniciada**) carece de utilidad, puesto que para acceder a la función hay que hacerlo con el identificador de la variable **bienvenido**.

La consola nos da un error donde se evidencia que el nombre de la función (**sesionIniciada**) no se puede referenciar directamente.

7. Tipos de funciones

3) FUNCIONES COMO OBJETOS

En JS todo es un objeto, e incluso las funciones se pueden definir como un objeto.

NOTA: esta es una variante muy poco utilizada.

```
const bienvenido = new Function("console.log('Bienvenido de nuevo');");
bienvenido();
```

4) FUNCIONES ANÓNIMAS

Son funciones que no tienen un identificador de función asociado.

```
const bienvenido = function () {
        console.log("Bienvenido de nuevo");
};
bienvenido();
```

En las funciones anónimas, como se ha ligado su uso a una variable, hasta que la variable no esté inicializada no se pueden utilizar. Esto no pasa con las funciones declarativas, que pueden invocarse en el código antes de que estén definidas.

7. Tipos de funciones

Ejemplo 4: CADENAS INVERTIDAS

Crea un programa que incluya una función anónima vinculada a una variable llamada invertida que reciba una cadena de texto y la devuelva invertida (transformada de derecha a izquierda).

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            //Defnimos la función anonimamente con un identificar asociado
            let invertida = function(cadena) {
                let resultado = "";
                for (let i = cadena.length -1; i >= 0; i--){
                    resultado += cadena [i];
                }
                return resultado;
            }
            let frase = prompt('Introduce una frase para invertirla:');
            document.write('La frase inicial es: '+ frase);
            document.write('<br><br>La frase invertida es: '+ invertida(frase));
        </script>
    </head>
    <body>
    </body>
</html>
```

La frase inicial es: Hola ¿qué tal?

La frase invertida es: ?lat éuq¿ aloH

7. Tipos de funciones

5) CALLBACKS

CALLABACK – función anónima que puede pasarse como parámetro a otra función.

La estrategia es definir una función que, al ser anónima, se vincula a una variable, y luego se utiliza dicha variable como parámetro de otra función, de manera que esta última función pueda ejecutar el contenido de la primera.

```
const bienvenido = function () {
        return "Bienvenido de nuevo, ";
};
const usuario = function (callback) {
        console.log(callback() + "Javier");
};
usuario(bienvenido); // muestra "Bienvenido de nuevo, Javier"
```

```
const usuario = function (callback) {
        console.log(callback() + "Javier");
};
usuario(
        function () {
                return "Bienvenido de nuevo, ";
        }
);
```

En muchos casos, ni siquiera sale a cuenta vincular una función anónima a una variable, sino directamente en la invocación a la función se utilizan los parámetros para definir la función anónima.

Si embargo, para funciones más extensas siempre es recomendable utilizar la primera forma, ya que mejora la legibilidad y el seguimiento del código.

7. Tipos de funciones

6) FUNCIONES AUTOAJUSTABLES

En alguna situación, puede tener sentido ejecutar una función inmediatamente después de crearla, por ejemplo, en cuanto el intérprete de JS esté disponible tras la carga de una web.

Las **funciones autoejecutables**, incluyen entre paréntesis una función sin identificador que será invocada tan pronto como el intérprete pase por su definición:

```
(function () {
        console.log("Bienvenido de nuevo");
})();
```

También es posible, pasar parámetros a la función:

```
(function (usuario) {
        console.log("Bienvenido de nuevo, "+usuario);
})("Javier");
```

NOTA: Hay que tener cuidado al asignar una función autoejecutable a una variable. La función sí se va a ejecutar de forma automática se utilice o no la variable. Eso no ocurre con las funciones anónimas, que sólo se ejecutan cuando se usa la variable.

```
const variable = (function (usuario) {
        return ("Bienvenido de nuevo, "+usuario);
})("Javier");
console.log(variable);
```

7. Tipos de funciones

Ejemplo 5: TIPOS DE FUNCIONES

Escribe un programa que incluya una función autoejecutable que informe por consola de la fecha y la hora a la que se inició la ejecución de la función.

NOTA: investiga el uso del objeto **Date** para facilitar el formato de los datos de salida

Date
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            //Definición función autojecutable
            (function() {
                let fechaInicio = new Date();
                document.write ("La función se inció el "+ fechaInicio.toLocaleDateString() + " a las " + fechaInicio.toLocaleTimeString());
            })();
        </script>
    </head>
    <body>
    </body>
</html>
```

7) FUNCIONES FLECHA

Forma simplificada de definir funciones, de manera que puede eliminarse la palabra function y usar => en su lugar.

```
const mensaje = () => { console.log ("Hola de nuevo."); };
mensaje();
```

Las ventajas de las funciones flecha son:

- Las llaves se pueden omitir si el cuerpo de la función tiene una sola línea

```
const mensaje = () => console.log ("Hola de nuevo.");
```

- Si la función tiene un **return**, también se puede obviar porque se haría de forma automática.
- Si la función tuviera un solo parámetro, también se podría prescindir de los paréntesis iniciales

```
const mensaje = nombre => console.log("Hola de nuevo, " + nombre);
mensaje("Javier");
```

- Aumentan la legibilidad el código
- Mejora la productividad de los programadores

7. Tipos de funciones

Ejemplo 6: FUNCIONES FLECHA I

Escribe un programa que incluya una función flecha que reciba dos cadenas de caracteres e informe de aquella que contiene más vocales.

```
CONTAR VOCALES

La primera cadena: hola, tiene 2 vocales
La segunda cadena: como estas , tiene 4 vocales
La cadena que más vocales tiene es la segunda: como estas
```

7. Tipos de funciones

Ejemplo 6: FUNCIONES FLECHA II

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            //Función flecha llamada contarVocales que tiene un parámetro de entrada (cadena). Cuenta las vocales de una cadena
            let contarVocales = (cadena) =>{
                let numVocales = 0;
                const vocales = ["a", "e", "i", "o", "u"];
                for (let i = 0; i < cadena.length; i++){
                    if (vocales.includes(cadena[i].toLowerCase())){
                        numVocales++;
                    }
                }
                return numVocales;
            };
            //Función flecha llamada masVocales que tiene dos parámetros de entrada (cadena1, cadena2) . Compara el número de
            vocales de cada cadena y nos dice cuál es la mayor
            let masVocales = (cadena1, cadena2)=>{
                let numVocalesCadena1 = contarVocales(cadena1);
                let numVocalesCadena2 = contarVocales(cadena2);
                document.write('CONTAR VOCALES <br><br>');
                document.write('La primera cadena: ' + cadena1 + ', tiene ' + numVocalesCadena1 + ' vocales');
                document.write('<br>La segunda cadena: ' + cadena2 + ', tiene ' + numVocalesCadena2 + ' vocales');
                document.write('<br>La cadena que más vocales tiene es ')
                if (numVocalesCadena1 > numVocalesCadena2){
                    document.write('la primera: ' + cadena1);
                } else if (numVocalesCadena2 > numVocalesCadena1){
                    document.write ('la segunda: ' + cadena2);
                } else {
                    console.log("Ambas cadenas tienen la misma cantidad de vocales");
                }
            }
            cadena1 = prompt('Ingresar la primera cadena:');
            cadena2 = prompt('Ingresar la segunda cadena:');
            masVocales(cadena1, cadena2);
        </script>
    </head>
    <body>
    </body>
</html>
```

CONTAR VOCALES

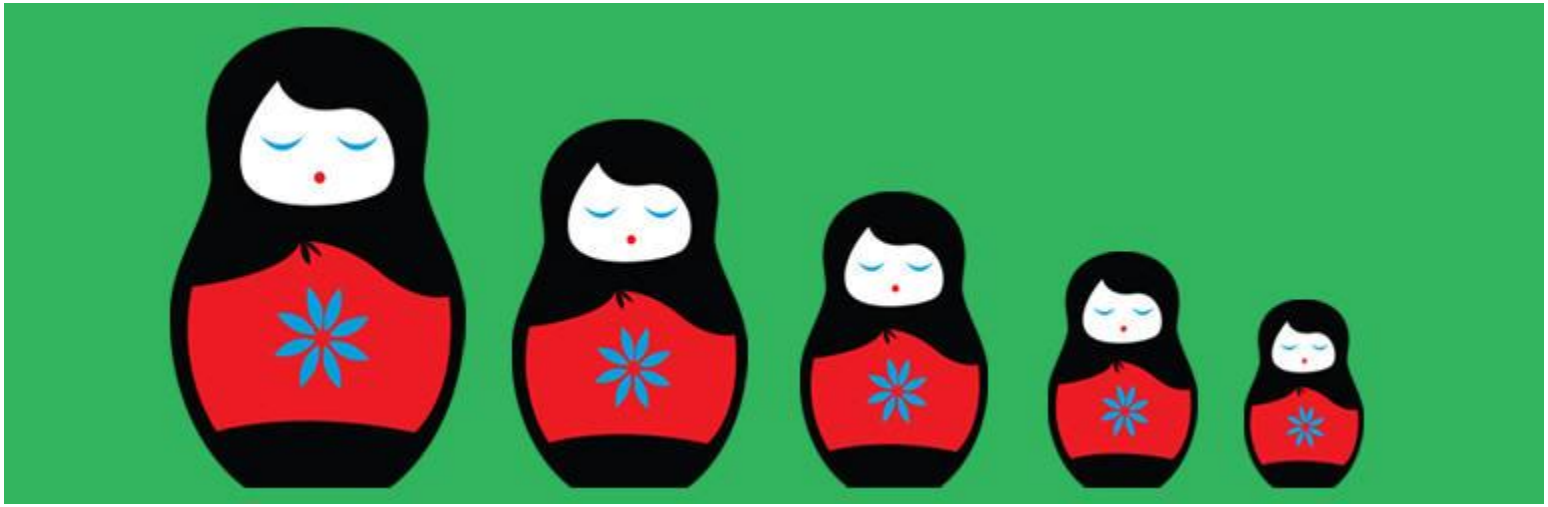
La primera cadena: hola, tiene 2 vocales
La segunda cadena: como estas , tiene 4 vocales
La cadena que más vocales tiene es la segunda: como estas

8. Recursividad

RECURSIVIDAD – capacidad que tienen las funciones de llamarse a sí mismas.

NOTA: La recursividad es una técnica de programación peligrosa (puede generar llamadas infinitas y desbordar la pila del sistema).

La recursividad es una forma elegante de resolver problemas y en muchos casos sirve para que problemas complejos tengan una solución más simple.



Ejemplo 7: FACTORIAL I

Escribir un programa que calcule el factorial de un número n.

FACTORIAL DE UN NÚMERO – Es el producto del número por todos los números anteriores. Por ejemplo: 5! = 5 * 4 * 3 * 2 * 1

CÁLCULO DEL FACTORIAL:

5! = 1 * 2 * 3 * 4 * 5 = 120

Ejemplo 7: FACTORIAL II

Escribir un programa que calcule el factorial de un número n.

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>Actividad 1</title>
6           <script>
7               function factorial(n){
8                   if (n <=1)
9                       return 1;
10                  else
11                      return n * factorial(n-1);
12              }
13              let num = parseInt(prompt('Introduce el número del que quieres calcular
                el factorial '));
14              document.write('CÁLCULO DEL FACTORIAL: <br><br>');
15              document.write(num + '! = ');
16              for (let i=1; i<num; i++){
17                  document.write(i + ' * ');
18              }
19              document.write(num + ' = ')
20              document.write(factorial(num));
21          </script>
22      </head>
23      <body>
24      </body>
25  </html>
```

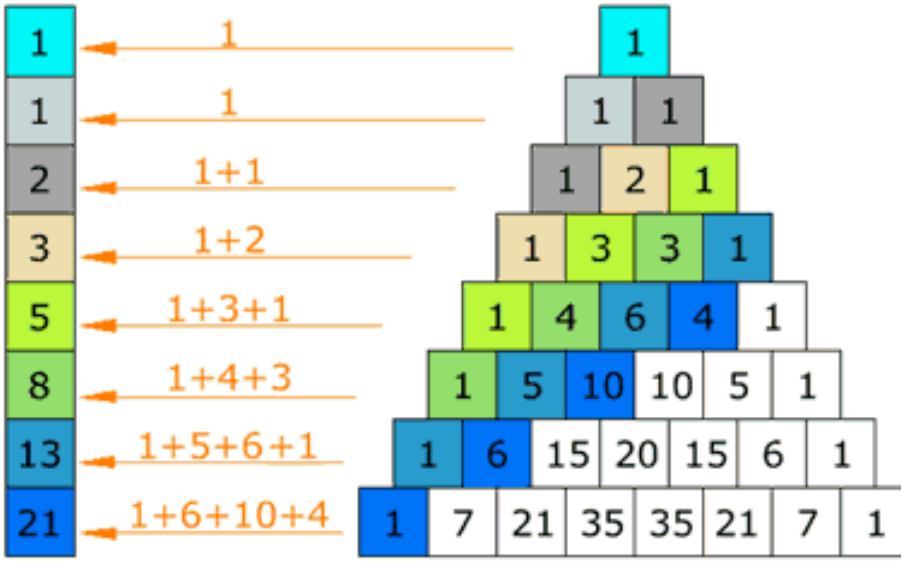
CÁLCULO DEL FACTORIAL:

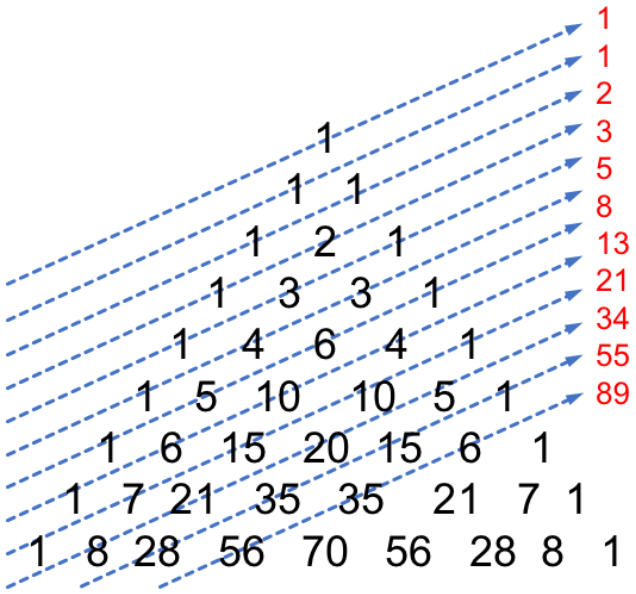
5! = 1 * 2 * 3 * 4 * 5 = 120

Ejemplo 8: FIBONACCI

Escribir un programa que pida un valor y muestre tantos elementos de la sucesión de Fibonacci como el valor introducido.

SERIE DE FIBONACCI - serie numérica en la que cada elemento es la suma de los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, etc.






```
SERIE DE FIBONACCI DE 10 ELEMENTOS:
1 1 2 3 5 8 13 21 34 55
```

Ejemplo 8: FIBONACCI

Escribir un programa que pida un valor y muestre tantos elementos de la sucesión de Fibonacci como el valor introducido.

SERIE DE FIBONACCI - serie numérica en la que cada elemento es la suma de los dos anteriores: 1, 1, 2, 3, 5, 8, 13, 21, etc.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            function fibo(numero){
                if (numero <2)
                    return 1;
                else
                    return fibo(numero-1) + fibo(numero-2);
            }
            let elementos = parseInt(prompt('Introduce cuántos elementos de la serie de Fibonacci quieres calcular '));
            document.write('SERIE DE FIBONACCI DE '+ elementos + ' ELEMENTOS: <br>')
            for (let i=0; i < elementos; i++)
                document.write(fibo(i) + ' ');
        </script>
    </head>
    <body>
    </body>
</html>
```

SERIE DE FIBONACCI DE 10 ELEMENTOS:
1 1 2 3 5 8 13 21 34 55

8. Recursividad

La estrategia básica de la recursividad que incorporan todos los algoritmos recursivos es:

Caso base + caso recursivo

Siempre que se plantee un problema recursivo debe buscarse el caso base, es decir, el caso en el que no es necesario seguir invocando a la función. Este es el caso en el que la recursividad se detiene.

En el ejemplo, la recursividad debe parar al estar el elemento 0 o 1, es decir, **numero < 2**, ya que es el inicio de la sucesión.

El resto de casos, necesitarán invocar a la función puesto que cada elemento de Fibonacci es la suma de los dos anteriores **fibo(numero1 + fibo(numero2)**

```
function fibo(numero){
    if (numero <2)
        return 1;
    else
        return fibo(numero-1) + fibo(numero-2);
}
```

Importante

En general, las soluciones recursivas son mucho más costosas computacionalmente que las soluciones iterativas, por lo que se recomienda que se recurra a la recursividad solo en aquellos casos en los que la solución iterativa no sea posible.

9. Otras funcionalidades

1.- ORDENACIÓN AVANZADA DE ARRAYS

```
let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort();
// ordenación esperada -> ['casa', 'Casado', 'ñam', prueba', 'zancos']
// ordenación obtenida -> ['Casado', 'casa', 'prueba', 'zancos', 'ñam']
```

En este ejemplo, la ordenación que obtenemos no es la esperada, ya que dicha ordenación se realiza considerando la posición que ocupa cada carácter en la tabla Unicode.

Pero la **función sort** deja abierta la posibilidad de incluir un callback que permita establecer un criterio de ordenación personalizado.

La función sort debe incluir dos parámetros de manera que la función devuelva:

- **< 0** - se sitúa el primero en un índice menor que el segundo, es decir, aparece antes. El primero se pone al principio.
- **= 0** - no se realizan cambios entre ellos
- **> 0** - se sitúa el segundo en un índice menor que el primero, es decir, aparece antes. El segundo se pone al principio.

9. Otras funcionalidades

La primera mejora en la búsqueda que se puede añadir es situar las palabras más cortas al principio del array:

```
let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort((primera,segunda)=>primera.length-segunda.length);
// Ordenación obtenida -> ['ñam', 'casa', 'Casado', 'prueba', 'zancos']
```

Ahora vamos a utilizar una función flecha (por su simplicidad) para aplicar una ordenación estricta del español, es decir, situar la "ñ" entre la "n" y la "o", y también no tener en cuenta las mayúsculas y minúsculas.

La función destinada a la comparación de cadenas de caracteres en un idioma determinado que se pasa como parámetro es **localCompare()**.

```
let vector = ["Casado","casa","prueba","zancos","ñam"];
vector.sort((primera,segunda)=>primera.length-segunda.length);
vector.sort((primera,segunda)=>primera.localeCompare(segunda,"es"));
// Ordenación obtenida -> ['casa', 'Casado', 'ñam', 'prueba', 'zancos']
```

Ejemplo 9: PARES I

Escribir un programa que incluya una función que dado un array de enteros positivos lo ordene, pero además sitúe a los pares al principio del array. Ambos grupos deben estar bien ordenados (pares e impares).

```
El array original es:
5,7,3,1,4,9,2,6,8

El array ordenado de forma creciente es:
1,2,3,4,5,6,7,8,9

El array separando pares e impares:
8,6,4,2,1,3,5,7,9

El array definitivo (separando pares e impares ordenados de forma creciente) es:
2,4,6,8,1,3,5,7,9
```

PISTAS:

1) Ordenar el array de menor a mayor
2) Ordenar el array de nuevo separando pares al principio e impares al final
3) Ordenar el array ordenando pares e impares de forma creciente

CESUR
Tu Centro Oficial de FP

Ejemplo 9: PARES II

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Actividad 1</title>
        <script>
            let vector = [5,7,3,1,4,9,2,6,8];
            document.write("El array original es:<br>" + vector)
            //Primero: hacermos una ordenación por defecto para colocar todos los elementos de menor a mayor
            vector.sort();
            document.write("<br><br>El array ordenado de forma creciente es:<br>" + vector);
            //Segundo: ordenar el array separando pares al principio e impares al final
            vector.sort(
                (primero,segundo)=>{
                    if (primero % 2 == 0)
                        return -1;
                    else
                        return 0;
                }
            );
            document.write("<br><br>El array separando pares e impares:<br>" + vector);

            //Tercero: hacemos una nueva ordenación ordenando pares e impares de forma creciente
            vector.sort(
                (primero,segundo)=>{
                    if ((primero % 2 == 0) && (segundo > primero))
                        return -1;
                    else
                        return 0;
                }
            )
            document.write("<br><br>El array definitivo (separando pares e impares ordenados de forma creciente) es:<br>" + vector);
        </script>
    </head>
    <body>
    </body>
</html>
```

El array original es:
5,7,3,1,4,9,2,6,8

El array ordenado de forma creciente es:
1,2,3,4,5,6,7,8,9

El array separando pares e impares:
8,6,4,2,1,3,5,7,9

El array definitivo (separando pares e impares ordenados de forma creciente) es:
2,4,6,8,1,3,5,7,9

9. Otras funcionalidades

2.- RECORRIDOS CON FOREACH

ForEach permite configurar un recorrido usando una función que recibe dos parámetros:

1) El primero almacena automáticamente cada elemento
2) El segundo (opcional) su índice.

Además, los elementos **undefined**, igual que pasaba con for..in, no los tiene en cuenta.

2.1.- ARRAYS

```
let vector = [12,334,111,52,98];
vector.forEach(function (elemento, posicion){
        console.log(`Posición ${posicion}: ${elemento}`);
});
```

```
Posición 0: 12
Posición 1: 334
Posición 2: 111
Posición 3: 52
Posición 4: 98
>
```

9. Otras funcionalidades

2.2.- CONJUNTOS

```
let conjunto = new Set();
conjunto.add(12).add(334).add(111).add(52).add(98);
conjunto.forEach(function (elemento){
        console.log(`Elemento: ${elemento}`);
});
```

```
Elemento: 12
Elemento: 334
Elemento: 111
Elemento: 52
Elemento: 98
>
```

2.3.- MAPAS

```
let mapa = new Map();
mapa.set('a',12).set('b',334).set('c',111).set('d',52).set('e',98);
mapa.forEach(function (valor,clave){
        console.log(`Clave: ${clave} / Valor: ${valor}`);
});
```

```
Clave: a / Valor: 12
Clave: b / Valor: 334
Clave: c / Valor: 111
Clave: d / Valor: 52
Clave: e / Valor: 98
>
```

3.- RECORRIDOS AVANZADOS DE ARRAYS

3.1.- MAP

Es una función establecida mediante una callback con un parámetro, que no modifica el contenido del array y que devuelve una copia con los cambios aplicados por la función.

Ejemplo: Obtener un array de precios con IVA a partir de otro sin IVA

```
let sinIVA = [12.45,34.42,99.90,49.95];
let conIVA = sinIVA.map(x=>(x*1.21).toFixed(2));
```

```
▶ (4) [12.45, 34.42, 99.9, 49.95]
▶ (4) ['15.06', '41.65', '120.88', '60.44']
>
```

NOTA: La función **toFixed()** se ha usado para formatear el resultado de las operaciones a dos decimales.

3.2.- FILTER

Es una función que recibe una callback con un parámetro que va recogiendo el valor de cada elemento del array. En cada iteración comprueba si el elemento cumple con una condición específica. Finalmente devuelve un array con aquellos elementos que han cumplido la condición.

```
let playas = ["Hierbabuena","Caños","Zahara","Carmen","Palmar"];
let filtradas = playas.filter(elemento=>elemento.length!=6);
```

Se crea un array con aquellos elementos cuya longitud es mayor o menor a seis caracteres, es decir, distintos a 6.

```
▶ (5) ['Hierbabuena', 'Caños', 'Zahara', 'Carmen', 'Palmar']
▶ (2) ['Hierbabuena', 'Caños']
>
```

Gracias