CIP FP cecheste

Desarrollo Web en Entorno Cliente

Javascript

Sintaxis [2]

Funciones

1. Una lista de argumentos para la función, encerrados entre paréntesis y separados por comas (,).
2. Las sentencias JavaScript que definen la función, encerradas por llaves, { }.
3. Los parámetros **primitivos**, como un número, se pasan por valor, los no primitivos, como un Array, por referencia.
4. El nombre de la función puede ser opcional según el contexto.
5. No es necesario respetar el número de argumentos declarados en la función. Aquellos que no se pasen a través de la llamada serán `undefined`.
6. Los argumentos de una función en Javascript se almacenan en un array denominado `arguments`.

Funciones

7. Las variables definidas dentro de una función no pueden ser accedidas desde ningún lugar fuera de la función.

8. Una función puede acceder a todas las variables y funciones definidas dentro del ámbito en el cual está definida.

9. Una función puede declararse dentro de cualquier ámbito, ya sea el programa principal, una estructura u otra función. [Mucho cuidado]

Funciones

```
function nombre(p1, p2, …, pN) {
    //Instrucciones
}
```

```
//Expresión de función

let nombre = function(p1, p2, …, pN) {
    //Instrucciones
}
```

```
//Función flecha

let nombre = (p1, p2, …, pN) => //Instrucción ;
```

Funciones flecha

1. No llevan la palabra reservada function.
2. Las sentencias de JavaScript que definen la función estarán encerradas por llaves, { }, salvo si solo tienen una instrucción.
3. Los parámetros estarán entre () excepto si solo hay uno, en cuyo caso se pueden obviar.
4. La sentencia `return` se da por hecha y se omite.

```
let variable = (p1, p2, …, pN) => //Instrucción ;

let variable = (p1, p2, …, pN) => { //Instrucciones };

let variable = p1 => //Instruccies ;
```

Continuación -> Métodos sobre arrays

array. forEach

```
let frutas = ['Manzana', 'Banana'];

frutas.forEach(function (elemento, indice, array) {
   console.log(elemento, indice);
});
```

```
// Manzana 0
// Banana 1
```

Analiza el funcionamiento del bucle

Continuación -> Métodos sobre arrays

map()

Permite recorrer el array y modificar los elementos presentes en él, retornando un nuevo array con la misma longitud que el original.

```
const array = [1, 2, 3, 4, 5, 6];
let nuevoArray = array.map((elemento) => elemento + 10);
console.log(nuevoArray);


//[11, 12, 13, 14, 15, 16]
```

Continuación -> Métodos sobre arrays

filter()

Recorre el array y retorna un nuevo array con aquellos elementos que pasen una determinada condición.

```
const array = [1, 2, 3, 4, 5, 6];
let nuevoArray = array.filter(elemento => elemento % 2 == 0);
console.log(nuevoArray);
```

```
//[2, 4, 6]
```

Continuación -> Métodos sobre arrays

findIndex()

Devuelve el índice del primer elemento de un array que cumpla con la función de prueba proporcionada. En caso contrario devuelve -1.

```
const array = [1, 2, 3, 4, 5, 6];
let indice = array.findIndex(elemento => elemento%2 == 0);
console.log(nuevoArray);


//1
```

Continuación -> Métodos sobre arrays

reduce()

Aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor.

```
const array = [1, 2, 3, 4, 5, 6];
let acumulador = array.reduce((acumulador, elemento) =>
acumulador + elemento);
console.log(acumulador);


//21
```

Continuación -> Métodos sobre arrays

some()

Itera el array y retorna un booleano si como mínimo uno de los elementos presentes en el array pasa una condición determinada.

```
const vector = [1, 2, 3, 4, 5, 6];
const alMenosUnPar = vector.some( elemento => elemento%2 == 0);
console.log(alMenosUnPar);
```

```
//true
```

Continuación -> Métodos sobre arrays

every()

Itera el array y retorna un booleano si todos los elementos presentes en el array pasan una condición determinada.

```
const vector = [1, 2, 3, 4, 5, 6];
const todosPares = vector.every( elemento => elemento%2 == 0);
console.log(todosPares); //false
```

```
//false
```

Continuación -> Métodos sobre arrays

flat()

Crea una nuevo array con todos los elementos de sub-array concatenados recursivamente hasta la profundidad especificada.

```
const vector = [1, 2, [3, 4, [5, 6]]];
const nuevoVector = vector.flat(1);
console.log(nuevoVector);
```

```
//[1,2,3,4[5,6]]
```

Continuación -> Métodos estáticos sobre arrays

isArray()

¿Qué es un método estático?

Determina si el valor pasado es un Array.

```
const vector = [1, 2, 3, 4, 5, 6];
console.log(Array.isArray(vector));
```

```
//true
```

Continuación -> Métodos estáticos sobre arrays

from()

Crea una nueva instancia de Array a partir de un objeto iterable.

```
console.log(Array.from('CIPFP CE CHESTE'));
```

```
//['C', 'I','P','F','P',' ','C','E',' ','C','H','E','S','T','E']
```

Funciones predefinidas

Funciones integradas en el lenguaje que podemos utilizar directamente.

- eval
- isNaN
- parseInt
- parseFloat

```
const variable = "10";
console.log(isNaN(variable));
```

¿Qué devolverá `isNaN()`?

Importar y exportar

Exports con nombre

archivo.js

```
// exporta la función declarada
export { myFunction };

// exporta una constante
export const foo = Math.sqrt(2);

// directamente sobre la función
export function myFunction2 () {

}
```

Importar y exportar

Imports

archivo2.js

```
// importa las funciones, constantes, … indicadas
import { myFunction, myFunction2, foo } from “archivo”;
```

¿ Funciona? ¿Por qué?

```
<script type="module" src="movidas.js"></script>
```

Importar y exportar

Exports por defecto

```
export default function () {
    //instrucciones
}
```

¿Cómo crees que funcionará?

Solo puede haber un export por defecto

Importar y exportar

Imports para función por defecto

archivo2.js

```
// importa las funciones, constantes, … indicadas
import nombreQueQuiera from “archivo.js”;
```

Excepciones

Control de posibles excepciones en fragmentos de código.

```
try {
   monthName = getMonthName(myMonth);
} catch (e) {
   monthName = "unknown";
   logMyErrors(e);
} finally {
   console.log("fin");
}
```

Código a controlar

Lo hago ante una excepción

Lo hago siempre

Excepciones

Generar excepciones

```
throw "Error2";    // Tipo string
throw 42;          // Tipo número
throw true;        // Tipo booleano
```

Ejercicios

1. Realiza un programa en el que nos muestre un menú con los ejercicios del bloque anterior y nos permita acceder a cada uno ellos seleccionando una opción. Utiliza funciones para organizar los ejercicios.

2. Crea una calculadora en la que primero se nos pidan dos números y después qué operación deseamos realizar: suma, resta, división entera (esta debe devolver también el resto) o multiplicación. Utiliza funciones para organizar mejor el ejercicio.

3. Crea un programa que gestione un inventario de productos. El programa debe permitir agregar, eliminar y actualizar productos, así como mostrar el inventario completo con su respectivo stock y precio unitario. Además, debe calcular el valor total del inventario. Para la creación del programa, usa 2 o más archivos .js

- `mostrar_menu()`: Muestra las opciones disponibles para el usuario.
- `agregar_producto()`: Permite al usuario agregar un nuevo producto al inventario.
- `eliminar_producto()`: Permite al usuario eliminar un producto existente del inventario.
- `actualizar_producto()`: Permite al usuario actualizar la cantidad o el precio de un producto.
- `mostrar_inventario()`: Muestra todos los productos en el inventario junto con sus cantidades y precios.
- `calcular_valor_total()`: Calcula el valor total del inventario.

NOTA: Vosotros sois los que diseñáis las funciones, y, por lo tanto, los qué decidís que argumentos les pasamos y que nos devuelven o no.