Lenguaje JavaScript

Contenido

Introducción

JavaScript es un lenguaje de programación multiplataforma orientado a objetos que se utiliza para hacer que las páginas web sean interactivas. También hay versiones de JavaScript de lado del servidor más avanzadas, como Node.js, que te permiten agregar más funcionalidad a un sitio web.

JavaScript cuenta con una biblioteca estándar de objetos como “Array”, “Date” o “Math”, y un conjunto básico de elementos de lenguaje como operadores, estructuras de control y declaraciones. Ten en cuenta que JavaScript es extensible pudiendo operar en variedad de entornos. Por ejemplo:

- JavaScript de lado del cliente extiende el núcleo del lenguaje con el **DOM** (Document Object Model) y con el **BOM** (Browser Object Model).
- JavaScript de lado del servidor extiende mediante módulos que le permiten interactuar con bases de datos, sistemas de archivos y otros servicios.

JavaScript actualmente está estandarizado por ECMA, en realizad su nombre es **ECMAScript** o ECMA-262 que es el estándar con sus especificaciones.

Cosas que debes conocer sobre JavaScript

- **Interpretado**. Ejecuta cada instrucción de manera secuencial.
- **Basado en prototipos**. Es un tipo de POO en el que se clonan los objetos o se componen en base a plantillas con la estructura del objeto.
- **Case Sensitive**. Diferencia entre mayúsculas y minúsculas.
- **Débilmente tipado**. No hace falta especificar el tipo de dato de la variable.
- **Monohilo y asíncrono**.
- **Dinámico**. Permite cambiar la estructura de los objetos en tiempo de ejecución.

Debemos tener en cuenta que ECMAScript evoluciona año tras años, dicho lo anterior podemos distinguir varias épocas:

- JavaScript primigenio.
- ECMAScript 5th Edition (**ES5**), introduce “strict mode”, métodos adicionales en los arrays y mejoras en el manejo de objeto.
- ECMAScript 2015 (**ES6**), moderniza el lenguaje. Nuestro código debe cumplir con los estándares de esta versión.

- Ampliaciones posteriores. En este caso es posible que no todas las especificaciones estén soportadas por los navegadores, da pie a librerías externas que añaden compatibilidad (librerías "**polyfill**").

Más información en https://www.w3schools.com/js/js_versions.asp

Etiqueta script

La etiqueta script tiene una serie de peculiaridades.

Permite definir código directamente en cualquier parte del `<body>`. Se ejecuta de manera secuencial como el resto de etiquetas html.

```
<body>
    Hola clase!
    <script>
        console.log('Hola mundo!');
    </script>
</body>
```

Permite enlazar un archivo externo, es el caso común. En este caso se incluye en cabecera del documento, etiqueta `<head>`. La ventaja de esto es que se carga en paralelo a la carga del contenido html (2 conexiones simultaneas).

```
<head>
    <meta charset="UTF-8">
    <title>Demos JavaScript</title>
    <script src="app.js"></script>
</head>
```

Al cargar documentos externos podemos añadir la propiedad "`defer`", la cual garantiza que los ficheros externos se carguen en orden y que sólo se ejecuten cuando el documento se haya parseado completamente.

```
<head>
    <meta charset="UTF-8">
    <title>Demos JavaScript</title>
    <script src="app.js" defer ></script>
</head>
```

IMPORTANTE: `<script>` siempre tiene como etiqueta de cierre `</script>`. La sintaxis `<script src="app.js" />` produce un error.

Comentarios de código

Comentarios de línea

```
// Esto es un comentario de una sola línea
```

Comentarios de bloque

```
/*
    Esto es un comentario de varias líneas
*/
```

Comentarios tabulados (el disparador de VC Code es **/****).

```
/**
 * Esto es un comentario multilínea
 * con asteriscos al principio para
 * que se vea bonito
 */
```

Declaración de variables

Vamos a trabajar con `let` y `const`. Recuerda que una variable sin valor toma el valor '`undefined`'. Hay otras posibilidades que vamos a conocer pero que no son recomendadas.

Para definir un identificador podemos emplear letras "A-Z" o "a-z", el símbolo de dólar "$" y el subrayado "_". Recuerda que JavaScript diferencia entre mayúsculas y minúsculas.

NOTA: si usas el modo estricto los siguientes ejemplos no se comportan como se indican.

```
//Declaración de variables

//Variables primitivas
a = 1;
/**
 * Las variables globales se almacenan en el objeto
 * global (window en el navegador).
 * a = 1 es equivalente a window.a = 1
 */

//Forma de declarar variables anterior a ES6, NO SE RECOMIENDA.
//Global en el ámbito de la función y genera hoisting (elevación,
//se declara al inicio de la función)
var a = 1;

//Variable local
let b = 1;
```

```
//Constante
//No se puede reasignar
const c = 1;

//Valor por defecto 'undefined'
//OJO que es reasignable
let d;
console.log(d); //undefined
d = 10;
d = undefined;
console.log(d); //undefined
```

Tipado débil

Hay lenguajes de programación que nos obligan a definir el tipo de la variable al declararla, esto se conoce como tipado estático. Esto es bueno porque nos ayuda a prevenir errores de codificación a la vez que aumenta la velocidad de las aplicaciones.

JavaScript emplea tipado dinámico, lo que significa que una variable puede almacenar cualquier tipo de valor durante la ejecución de la aplicación.

Recuerda que por defecto el valor de una variable es "`undefined`" y que puede volver a asignarse.

Por ejemplo.

```
let nombre = 'Alejandro';
console.log(nombre);
nombre = 23;
console.log(nombre);
nombre = undefined
console.log(nombre);
```

Tipos de datos

Tipos primitivos

- **String**. "", '', `` .
- **Boolean**. true y false.
- **Number**. 42 o 3.14159
- **BigInt**. (rango +-(2^53-1)). A partir de 15 dígitos JS redondea. 9999999999999999n
- **Symbol**. new Symbol(). Crea un identificador único (algo similar a GUID).
- **null**. Es un tipo Object y un valor.

- **undefined**. Representa el valor de una variable que no ha sido definida. Puede asignarse a una variable ya inicializada.

Tipos no primitivos

- **Object**

Hay otros valores especiales como:

- **NaN**. Not a Number, que se produce cuando una operación aritmética es errónea.
- **Infinity**, **-Infinity**. Valores demasiado grandes o demasiado pequeños.

Más información en:

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_types

Por ejemplo.

```
let numero = 42;
let nombre = "Hola mundo";
let verdadero = true;
let noDefinida;
let nula = null;

console.log(numero, typeof numero);
console.log(nombre, typeof nombre);
console.log(verdadero, typeof verdadero);
console.log(noDefinida, typeof noDefinida);
console.log(nula, typeof nula); // este devuelve un object
```

Ejemplos problemáticos.

```
console.log('70' - 5);  //65
console.log('70' + 5); //705

console.log(0.1 + 0.2);

console.log(9999999999999999n);
console.log(9999999999999999);

console.log('1 + 1'); // '1 + 1'
console.log(new String('1 + 1')); // String('1 + 1')
```

Consideraciones finales.

Para evitar problemas con las variables vamos a emplear siempre los literales para definir variables de tipo String, Number y Boolean.

- Cadenas -> Usamos ''
- Números -> Usamos 0 o 0.0
- Booleanos -> Usamos `true` o `false`

Los tipos envoltorios disponen del método ".valueOf()" para obtener el literal.

Cadenas de texto

Las cadenas de texto son un tipo de datos especial (son un array de caracteres) que admite 3 maneras distintas de definición en JavaScript.

- **Comillas dobles ""**. No lo vamos a usar en JavaScript, lo dejamos para maquetar código HTML.
- **Comilla simple o apostrofe ''**. Opción recomendada.
- **Apostrofe inclinado ``**. Define plantillas de texto que admiten código mediante `${}`, además respetan los espacios y saltos de línea adicionales.

IMPORTANTE: El principal uso de JavaScript es manipular código HTML dinámicamente. El problema del código HTML es que también emplea las comillas dobles en su maquetado con lo que si definimos cadenas de texto con comillas dobles y además debo incluir comillas dobles tendré problemas. Por ejemplo:

```
elementoHtml.innerHTML = '<p class="destacado">Piensa en todo dobles</p>';
```

RECOMENDACIÓN: acostúmbrate a usar comillas dobles en HTML y apostrofes en JavaScript. Si sigues dudando pulsa el F12 en el navegador y mira como maqueta en la pestaña "Elementos" y como maqueta en la pestaña "Consola".

Las **plantillas de texto** `` permiten incrustar variables y expresiones dentro de una cadena de texto. Resulta útil para componer URLs, por ejemplo.

```
const hostCliente = 'www.example.com';
let productoId = 23;
let direccion = `${hostCliente}/productos?productoId=${productoId}`;
```

Un ejemplo con expresiones.

```
let edad = 19;
console.log(`El alumno es: ${edad < 18 ? 'menor' : 'mayor'} de edad`);
```

El carácter de escape es la barra inclinada (\), a recordar los siguientes caracteres.

- \n. Nueva línea.
- \t. Tabulación.
- \". Comilla doble.
- \'. Comilla simple.
- \\. Barra inversa.
- \(intro). Permite partir en el código cadenas de texto muy largas.

Operadores aritméticos

Operadores binarios (dos operandos).

- +. Suma.
- -. Resta.
- *. Multiplicación.
- /. División.
- %. Módulo o resto de división.
- **. Potencia.

Operadores unarios (un operando).

- **++variable.** Preincremento
- **variable++.** Postincremento
- **--variable.** Predecremento.
- **variable--.** Postdecremento.

Por ejemplo.

```
let a = 5;
let b= 7;
console.log(a + b, 'adición');
console.log(a - b, 'resta');
console.log(a * b, 'multiplicación');
console.log(a / b, 'división');
```

```
console.log(a % b, 'módulo');
console.log(a ** b, 'potencia');

//Operador incremento
let c = 10;
console.log(c, 'postincremento', c++);
console.log(c, 'actual');
console.log(c, 'preincremento', ++c);
```

Operadores de asignación

La asignación ser realiza con el operador "=". Recuerda que podemos desasignar el valor de una variable con "`undefined`", todas las variables no inicializadas son "`undefined`".

```
let a = 5;
a = 'hola';
a = undefined;
```

JavaScript también soporta **operaciones abreviadas**.

- **+=**. Suma.
- **-=**. Resta.
- ***=**. Multiplicación.
- **/=**. División.
- ****=**. Potencia.

Por ejemplo.

```
let a = 5;
let b = a + 5;
a += 5;
a -= 5;
a *= 5;
a /= 5;
a **= 5;
console.log(a);
```

Operadores de comparación

Operadores de comparación:

- **>**. Mayor que.
- **>=**. Mayor igual que.

- **<**. Menor que.
- **<=**. Menor igual que.

Ten cuidado con lo siguiente.

- **==**. Igual que.
- **!=**. Distinto que.
- **===**. Igual que y mismo tipo.
- **!==**. Distinto que y distinto tipo.

NOTA: es recomendable el uso de "===" y "!==" para evitar errores de codificación.

Por ejemplo.

```
let a = 5;
console.log('a > 5', a > 5);
console.log('a >= 5', a >= 5);
console.log('a < 5', a < 5);
console.log('a <= 5', a <= 5);

a = 10;
console.log('a == 10', a == 10); // true
console.log('a != 10', a != 10); // false

//Cuidado con los tipos de datos
console.log('a == "10"', a == '10'); // true comprueba sólo el valor.
console.log('a === "10"', a === '10'); // false comprueba el tipo y luego
el valor.
console.log('a !== "10"', a !== '10'); // true
```

Operadores lógicos

Disponemos de los siguientes operadores lógicos.

- **&&**. Operador "y", sólo true si ambos son true.
- **||**. Operador "o", true si al menos uno es true.
- **!**. Negación. Invierte el valor.

Por ejemplo

```
let a = true;
let b = false;

console.log('a && b:', a && b); // false
console.log('a || b:', a || b); // true
```

```
console.log('!a:', !a); // false
console.log('!b:', !b); // true

console.log('(a || b) && !b:', (a || b) && !b); // true
```

Short-circuit

Los "**corto circuitos**" son expresiones lógicas que se evalúan de izquierda a derecha. Lo interesante de este tipo de expresiones es que dejan de evaluarse cuando se cumple la condición.

Reglas.

- **expresion1 && expresion2**. Ejecuta de izquierda a derecha y devuelve la última expresión "**truthy**".
- **expresion1 || expresion2**. Ejecuta de izquierda a derecha y devuelve la primera expresión "**truthy**".
- **expresion1 ?? valorPorDefecto**.

Por ejemplo.

```
function fnCierta() {
    console.log('invoca fnCierta');
    return true;
}

function fnCierta2() {
    console.log('invoca fnCierta2');
    return 'fnCierta2';
}

let resultadoOr = fnCierta() || fnCierta2();
console.log(resultadoOr); //no se ejecuta fnCierta2
let resultadoAnd = fnCierta() && fnCierta2();
console.log(resultadoAnd);
//se ejecutan ambas funciones, devuelve el resultado de la segunda
```

Lo que acabamos de ver es un ejemplo muy sencillo. Los corto circuitos tiene sentido cuando aplican con los valores "**falsy**". Veámoslo a continuación.

Valores “falsy”

Los valores “**falsy**” son valores que se consideran falso siempre que se evalúen en un contexto booleano. Todos los valores que no son “**falsy**” devuelven “**truthy**”.

Lista de valores “**falsy**”.

- null
- undefined
- false
- NaN
- 0, 0.0, 0x0
- -0, -0.0, -0x0
- 0n. **NOTA:** no existe -0n.
- ‘’. Cadena vacía.
- document.all

Por ejemplo.

```
let usuarioActual;
//usuarioActual = 'Juanito';
let nombreMostrar = usuarioActual || 'Anónimo';
console.log(nombreMostrar);
```

Estructuras de control

Las estructuras de control nos permiten aplicar el código de manera condicional. Veamos a continuación las distintas estructuras disponibles en JavaScript.

Condicional “if”

Permite ejecutar código condicional al cumplirse la condición.

```
if (condicion) {
    // Código a ejecutar si la condición es verdadera
}
```

Podemos añadir código alternativo cuando la condición no se cumple anidando “**else**” o “**else if**”.

Por ejemplo.

```
let temperatura = 25;

if (temperatura > 20) {
    console.log('El clima es agradable');
} else if (temperatura > 10) {
    console.log('Hace un poco de frío');
} else {
    console.log('Hace mucho frío');
}
```

Condicional "switch"

Concatenar instrucciones "**if else**" puede acabar originando código poco legible. La instrucción "**switch**" permite definir múltiples opciones de una manera más sencilla.

```
switch (a) {
    case 1:
        // código para a === 1
        break;
    case 2:
        // código para a === 2
        break;
    default:
    // código si no coincide con ningún caso
}
```

RECUERDA: si omito el "**break**" se ejecuta el siguiente bloque de código.

```
let dia = 3;

switch (dia) {
  case 1:
    console.log('Hoy es lunes');
    break;
  case 2:
    console.log('Hoy es martes');
    break;
  case 3:
    console.log('Hoy es miércoles');
    break;
  case 4:
    console.log('Hoy es jueves');
    break;
  case 5:
```

```
    console.log('Hoy es viernes');
    break;
  default:
    console.log('Es fin de semana');
}
```

Operador ternario "?"

Permite definir una instrucción "**if else**" en una única instrucción, útil para situaciones simples.

```
let resultado = condicion ? expresionSiVerdadero : expresionSiFalso;
```

Por ejemplo.

```
let numero = 5;
let resultado = numero >= 0 ? 'Positivo' : 'Negativo';
console.log(resultado);  // Salida: 'Positivo'
```

Operador de coalescencia nula "??"

El operador de coalescencia nula "**??**" devuelve el lado derecho del operando cuando el lado izquierdo es "**null**" o "**undefined**".

```
let resultado = valorPropuestro ?? valorPorDefecto;
```

Por ejemplo.

```
const iva = undefined ?? 21;
console.log(iva); // Salida: 21
```

Bucle "for"

Es bucle se emplea cuando conocemos el número de veces que tenemos que iterar. Su definición se compone de 3 partes:

1. **Inicialización**: Se ejecuta una sola vez antes de que comience el bucle. Generalmente se utiliza para declarar y/o inicializar una variable de control.
2. **Condición**: Se evalúa antes de cada iteración del bucle. Si la condición es verdadera, el bucle continúa; si es falsa, el bucle se detiene.
3. **Actualización**: Se ejecuta al final de cada iteración, generalmente para incrementar o modificar la variable de control.

```
for (inicialización; condición; actualización) {
```

```
    // Código a ejecutar en cada iteración
}
```

Por ejemplo.

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
// Salida: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Bucle “for…of”

Itera sobre la colección devolviendo directamente el valor de los elementos, es decir, no necesitamos emplear los índices.

```
for (variable of iterable) {
    // Código a ejecutar con cada valor del iterable
}
```

Por ejemplo.

```
let numeros = [2, 3, 5, 7, 11];
for (let numero of numeros) {
    console.log(numero);
}
```

Bucle “for…in”

Este bucle tiene dos usos. Recorrer las propiedades de un objeto o recorrer los índices de un array.

El uso principal es el de recorrer las propiedades de un objeto.

```
let objeto = { nombre: 'Willy', edad: 23, merienda: 'bocata' };
for (let llave in objeto) {
    console.log(llave, objeto[llave]);
}
// Muestra las claves y valores: nombre Willy, edad 23, merienda bocata
```

El segundo uso es el de obtener los índices de un array.

```
let array = [10, 20, 30];
for (let index in array) {
    console.log(index, array[index]);
```

```
}
// Muestra los índices y valores: 0 10, 1 20, 2 30
```

Bucle “while”

El bucle itera mientras se cumpla la condición. Si no se cumple la condición inicialmente el bucle itera 0 veces.

```
while (condición) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

NOTA: Recuerda actualizar la condición en el interior del bucle para no iterar hasta el infinito.

Por ejemplo.

```
let i = 0;

while (i < 10) {
    console.log(i);
    i++; // Actualización de la variable para evitar un bucle infinito
}
```

Bucle “do … while”

El bucle itera mientras se cumpla la condición. La diferencia con el while es que do … while itera al menos 1 vez.

```
do {
    // Código a ejecutar
} while (condición);
```

Por ejemplo.

```
let i = 0;

do {
    console.log(i);
    i++; // Actualización de la variable
} while (i < 10);
```

Instrucciones “continue” y “break”

Las instrucciones break y continue se emplear para interrumpir el flujo normal del bucle.

- **break**, rompe el bucle, manda la ejecución a la primera instrucción fuera del bucle.
- **continue**, termina la iteración actual del bloque, salta a la siguiente iteración del bucle.

Por ejemplo.

```
for (let i = 0; i < 10; i++) {
    if (i === 5) break; // Detiene el bucle cuando i es 5
    if (i === 3) continue; //Salta a la siguiente iteración cuando i es 3
    console.log(i);
}
```

Arrays "[]" y JSON "{}"

En este apartado vamos a tratar brevemente el uso básico de los arráis [] y de los objetos JSON {} estos últimos vistos como colecciones de tipo clave-valor (diccionarios). Más adelante profundizaremos en ellos cuando veamos POO.

Array

Permiten almacenar una lista ordenada de cualquier tipo de datos. Los elementos son indexados por un número entero siendo el primer índice 0.

Inicialización.

- **[]**. Devuelve un array vacío.
- **new Array(tamaño)**. Devuelve un array vacío del tamaño indicado.
- **[lista valores]**. Devuelve un array con los elementos indicados.

Por ejemplo.

```
let numeros = [10, 20, 30];
```

Acceso a elementos.

A través de su índice. Podemos leer y escribir.

```
console.log(numeros[1]);  //devuelve 20
numeros[5] = 60; //Nuevo elemento en el índice 5
```

Tamaño (propiedad .length)

La propiedad "`.length`" devuelve el tamaño del array.

```
console.log(numeros.length); //devuelve 6
```

Objetos JSON - Diccionarios

Los objetos en JavaScript son colecciones de pares clave-valor. Cada clave (o propiedad) se asocia con un valor, y las propiedades no están ordenadas.

NOTA: existe un tipo "**Map**" para gestionar colecciones de clave-valor, pero lo habitual es emplear JSON por su facilidad de uso, por ejemplo, al serializar y deserializar.

Por ejemplo.

```
let persona = {
    nombre: "Juan",
    edad: 30
};
```

Acceso a propiedades

Podemos acceder a los valores de las propiedades usando la notación de punto (**.**) o la notación de corchetes (**[]**):

```
console.log(persona.nombre);
console.log(persona['edad']);
```

Eliminar una propiedad (operador delete)

Podemos eliminar una propiedad, o una clave con su valor, mediante el operador "**delete**".

```
delete persona.edad;
console.log(persona); // Salida: { nombre: 'Juan' }
```

Funciones

JavaScript dispone de varios tipos de funciones, vamos a desarrollarlas y a comentar las particularidades de cada tipo.

Funciones con nombre

Las funciones con nombre son aquellas que tienen un identificador que puede ser usado para invocarlas.

```
function saludar(nombre){
    return 'Hola ' + nombre;
}
```

```
console.log(saludar('Alberto')); // Salida: 'Hola Alberto'
```

Actualmente JavaScript soporta la asignación de valores por defecto en los parámetros, podemos ajustar el ejemplo anterior de la siguiente manera.

```
function saludar(nombre = 'desconocido'){
    return 'Hola ' + nombre;
}
console.log(saludar()); // Salida: 'Hola desconocido'
```

El objeto especial "**arguments**" está disponible en todas las funciones regulares (no en "arrow functions") y contiene todos los argumentos que se pasaron a la función, incluso si no se definieron explícitamente en los parámetros.

```
function sumar() {
    let total = 0;
    for (let i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
}

console.log(sumar(1, 2, 3, 4));  // Salida: 10
```

Parámetro variable "...rest"

El operador **"...rest**" permite agrupar un número indefinido de argumentos en un array. A diferencia de "**arguments**", el operador "**...rest**" es un array real, por lo que puedes utilizar métodos de array directamente sobre él.

NOTA: la sintaxis de este operador es 3 puntos y un nombre de variable, es decir, se puede cambiar la palabra "**rest**" por otra más significativa.

```
function sumar(...numeros) {
    let total = 0;
    for (let numero of numeros) {
        total += numero;
    }
    return total;
}

console.log(sumar(1, 2, 3, 4)); // Salida: 10
console.log(sumar(5, 10)); // Salida: 15
```

Funciones anónimas

Las funciones anónimas son aquellas que no tienen nombre y son usualmente asignadas a una variable o utilizadas como argumentos en otras funciones (por ejemplo, en callbacks).

Por ejemplo.

```
let saludo = function () {
    return 'Hola, Mundo';
};

console.log(saludo());
```

OJO: Fíjate que le podemos asignar la función a una variable.

Arrow functions

Las "**arrow functions**" "**()=>{}**" son una forma concisa de escribir funciones en JavaScript. Tienen una sintaxis más corta y no tienen su propio "**this**".

```
let sumarFlecha = (a, b) => a + b;
console.log(sumarFlecha(2, 3));  // Salida: 5
```

IMPORTANTE: en las "arrow functions" la variable "this" se hereda del contexto en el que se define la función, en el resto de funciones "this" hace referencia al objeto que llama la función. **ESTO ES ESPECIALMENTE RELEVANTE EN OBJETOS DONDE NO SE RECOMIENDA SU USO**.

Por ejemplo, uso correcto.

```
let persona = {
    nombre: "Juan",
    edad: 30,
    saludar: function () {
        return `Hola, soy ${this.nombre} y tengo ${this.edad} años.`;
    }
};

console.log(persona.saludar());
// Salida: "Hola, soy Juan y tengo 30 años."
```

Uso incorrecto

```
let persona = {
```

```
    nombre: "Juan",
    edad: 30,
    saludar: () => {
        return `Hola, soy ${this.nombre} y tengo ${this.edad} años.`;
    }
};

console.log(persona.saludar());
// Output: "Hola, soy undefined y tengo undefined años."
```

Closures

Es un tipo de función que recuerda el entorno en el que fue creada, incluso después de que ese entorno haya terminado su ejecución. En esencia tenemos una función dentro de otra función, la función externa inicializa el entorno y permite almacenar el contexto y la función interna es la que utilizamos.

Este tipo de función puede ser útil en la gestión de eventos.

```
function crearContador() {
    let contador = 0;
    return function () {
        contador++;
        return contador;
    };
}

let contador1 = crearContador();
console.log(contador1()); // Salida: 1
console.log(contador1()); // Salida: 2

let contador2 = crearContador();
console.log(contador2()); // Salida: 1
```

Funciones auto invocadas - IIFE

Una **función autoinvocada** (**IIFE**, “Immediately Invoked Function Expression”) es una función que se define y se ejecuta de inmediato en el momento en que se crea. Se utiliza para encapsular código y evitar que las variables definidas dentro de la función contaminen el espacio global.

```
(function () {
    // Código dentro de la función
    console.log("Esta función se autoinvoca.");
```

```
})();
```

Por ejemplo.

```
let contador = (function () {
    // Variable interna (no accesible directamente desde afuera)
    let valor = 0;

    // Retornamos un objeto con métodos para manipular la variable
interna
    return {
        incrementar: function () {
            valor++;
            console.log("Valor actual:", valor);
        },
        reiniciar: function () {
            valor = 0;
            console.log("Contador reiniciado");
        },
        obtenerValor: function () {
            return valor; // Solo devuelve el valor sin permitir
modificarlo
        }
    };
})();

// Usando los métodos del objeto devuelto
contador.incrementar(); // Salida: "Valor actual: 1"
contador.incrementar(); // Salida: "Valor actual: 2"
console.log(contador.obtenerValor());
// Salida: 2 (se puede leer, pero no modificar)
contador.reiniciar(); // Salida: "Contador reiniciado"
console.log(contador.obtenerValor());
```

Desestructuración de Arrays y Objetos

La desestructuración permite extraer valores de arrays u objetos y asignarlos directamente a variables de forma más clara y concisa.

Ten en cuenta que:

- En arrays la extracción se hace por posición.
- En objetos la extracción se hace por nombre de propiedad.
- Se pueden usar valores por defecto y el operador "**...rest**" para recoger el resto de los elementos o propiedades.

Por ejemplo.

```
// Desestructuración de arrays
const numeros = [10, 20, 30, 40, 50, 60];

[a, b, ...resto] = numeros;
console.log('a: ', a); // a: 10
console.log('b: ', b); // b: 20
console.log('resto: ', resto); // resto: Array(4) [30, 40, 50, 60]

// Desestructuración con objetos
const alumno = { nombre: "Lucía", edad: 22, curso: "DAW", nota: 8 };

let { nombre, ...otros } = alumno;
console.log(nombre); // Lucía
console.log(otros);  // Object { edad: 22, curso: "DAW", nota: 8 }

let { edad, curso, telefono='000-000 000'} = alumno;
console.log('edad: ', edad, ", curso: ", curso, ", teléfono: ",
telefono);
// edad:  22 , curso:  DAW , teléfono:  000-000 000
```

Comunicación con el usuario

Podemos generar popups para interactuar con el usuario. Disponemos de varios tipos:

- **Mensaje de alerta**: `alert('mensaje')`. Muestra un mensaje de alerta al usuario.
- **Solicitud de confirmación**: `confirm('mensaje')`. Muestra un mensaje al usuario y le pide que confirme o cancele, devuelve `true` o `false`.
- **Solicitud de texto**: `prompt('mensaje', 'valor por defecto')`. Muestra un mensaje al usuario y le permite introducir un texto de respuesta, devuelve el texto introducido.

Para la salida por consola disponemos de distintas categorías de mensaje.

- `console.log('mensaje')`. Muestra un mensaje genérico en la consola.
- `console.error('mensaje')`. Muestra un mensaje de error en la consola.
- `console.warn('mensaje')`. Muestra un mensaje de advertencia en la consola.
- `console.debug('mensaje')`. Muestra un mensaje de depuración en la consola.
- `console.info('mensaje')`. Muestra un mensaje de información en la consola.
- `console.trace()`. Muestra la pila de llamadas en la consola.

NOTA: a estas alturas ya te habrás dado cuenta de que “console.log(..data)” admite múltiples parámetros.

Modo estricto 'use strict'

A estas alturas ya hemos visto que JavaScript tiene muchas manías que invitan al error. Para tratar de limitar estos problemas en ES5 se añadió el **modo estricto** para forzar el empleo de código moderno en JavaScript.

Podemos habilitar el modo estricto la instrucción **'use strict'**. Esta instrucción admite dos ámbitos de uso:

- **A nivel de script**. Debe ser la primera instrucción del script. Aplica a todo el código del script.
- **A nivel de función.** Debe ser la primera instrucción de la función. Aplica únicamente a la función.

IMPORTANTE: Ten en cuenta que si la instrucción no es la primera instrucción del ámbito correspondiente no se activa el modo estricto.

Este modo activa un modo seguro que implica principalmente:

- No poder usar variables sin declararlas.
- No poder usar la instrucción "**delete**".
- No poder mezclar distintas notaciones matemáticas que den origen a confusión (octal + decimal).
- No poder usar palabras reservadas del lenguaje como nombres de variable.
- No poder definir dos parámetros con el mismo nombre "**fn(p1, p1)**".
- En POO no escribir en propiedades de sólo lectura ni leer en propiedades de sólo escritura (**getters** y **setters**).
- La función "**eval()**" no puede definir variables por seguridad.
- El objeto "**this**" cuando se usa en funciones ahora es "**undefined**". Normalmente hace referencia al objeto "**window**".

Por ejemplo

```
// Modo estricto a nivel de script
'use strict';
let a = 20;
delete a; //esto no le gusta
```

Por ejemplo

```
// Modo estricto a nivel de función
```

```
x = 3.14; // Esto no produce error.
miFuncion();

function miFuncion() {
    'use strict';
    y = 3.14; // Esto produce error (y no definida).
}
```

Bibliografía

Documentación MDN JavaScript

https://developer.mozilla.org/es/docs/Web/JavaScript

Guía MDN JavaScript

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

Tutorial JavaScript W3C

https://www.w3schools.com/js/

Apuntes DWEC Castillo

https://xxjcaxx.github.io/libro_dwec/

JavaScript.info

https://javascript.info/js

Guía MDN – Expresiones y operadores

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_operators