Javascript

Semana 3

Índice

Resultado de aprendizaje, contenido y criterios de evaluación

Resultado de aprendizaje
RA2. Escribe sentencias simples, aplicando la sintaxis del lenguaje y verificando su ejecución sobre navegadores web.
Contenido
1 - Variables. 2 - Tipos de datos. Conversiones. 3 - Asignaciones. 4 - Operadores 5 - Comentarios al código. 6 - Sentencias. 7 - Decisiones. 8 - Bucles. 9 - Prueba y documentación del código.
Criterios de evaluación
e) Se han utilizado mecanismos de decisión en la creación de bloques de sentencias. f) Se han utilizado bucles y se ha verificado su funcionamiento g) Se han añadido comentarios al código h) Se han utilizado herramientas y entornos para facilitar la programación, prueba y documentación del código

Resultado de aprendizaje
RA3. Escribe código, identificando y aplicando las funcionalidades aportadas por los objetos predefinidos del lenguaje.
Contenido
6 - Interacción con el usuario.
Criterios de evaluación
e) Se han escrito sentencias que utilicen los objetos predefinidos del lenguaje para interactuar con el usuario

Resultado de aprendizaje
R4. Programa código para clientes web analizando y utilizando estructuras definidas por el usuario
Contenido
1 - Funciones predefinidas del lenguaje 2 - Llamadas a funciones. Definición de funciones 3 - Matrices (arrays). 4 - Operaciones agregadas: filtrado, reducción y recolección 5 - Creación de objetos. 6 - Definición de métodos y propiedades 8 - Depuración y documentación del código.
Criterios de evaluación
a) Se han clasificado y utilizado las funciones predefinidas del lenguaje b) Se han creado y utilizado funciones definidas por el usuario. c) Se han reconocido las características del lenguaje relativas a la creación y uso de matrices (arrays). d) Se han creado y utilizado matrices (arrays). e) Se han utilizado operaciones agregadas para el manejo de información almacenada en colecciones f) Se han reconocido las características de orientación a objetos del lenguaje g) Se ha creado código para definir la estructura de objetos. h) Se han creado métodos y propiedades i) Se ha creado código que haga uso de objetos definidos por el usuario k) Se ha depurado y documentado el código

Resultado de aprendizaje
RA6. Desarrolla aplicaciones web analizando y aplicando las características del modelo de objetos del documento
Contenido
1 - El modelo de objetos del documento (DOM). 2 - Objetos del modelo. Propiedades y métodos 3 - Acceso al documento desde código
Criterios de evaluación
a) Se ha reconocido el modelo de objetos del documento de una página web h) Se han independizado las tres capas de implementación (contenido, aspecto y comportamiento), en aplicaciones web

Semana 3. Funciones

Muchas veces nos encontramos con código que se repite o cálculos específicos que podemos aprovechar en otras aplicaciones. JavaScript ofrece las funciones a tal efecto y podemos profundizar en el anexo ANEXO IV. Funciones (función flecha, callbacks, scope, clousures)

Sesión 1

3.1. Ejemplo guiado

3.1.1. Combate de personajes

Ejemplo guiado

Crear un simulador de combate entre personajes usando callbacks y funciones flecha.

1. *Primeramente definimos un array con los jugadores, tendrán los atributos nombre y vida:*

   Sería interesante visitar el anexo sobre arrays AIII.1. Arrays.

```
const jugadores = [
   { nombre: "Ana", puntos: 100 },
   { nombre: "Luis", puntos: 100 },
   { nombre: "Marta", puntos: 100 }
];
```

2. *Definimos los tipos de ataque, por ejemplo de espada y magia. Estos serán métodos, recibirán como valores de referencia el jugador atacante y el defensor. El método calculará los daños de forma aleatoria que realizará el atacante al defensor y mostrará un mensaje por consola informando del enfrentamiento.*

```
function ataqueEspada(atacante, defensor) {
  const daño = Math.floor(Math.random() * 20) + 5;
  defensor.vida -= daño;
  console.log(`${atacante.nombre} ataca con espada y hace ${daño} de daño.`);
};

function ataqueMagia(atacante, defensor) {
  const daño = Math.floor(Math.random() * 30) + 10;
  defensor.vida -= daño;
  console.log(`${atacante.nombre} lanza un hechizo e inflige ${daño} de daño.`);
};
```

***NOTA**: Fíjate cómo estamos mostrando los mensajes por consola. Esta forma usa la "interpolación" de variables a través haciendo uso de ${…} y todo el texto a mostrar va entre comillas inclinada a la izquierda ` ….` . Es una forma cómoda de mostrar texto por consola de forma más legible y cómoda.*

3. *Ahora definiremos el turno de juego. Será otra función que recibirá al atacante, defensor y el tipo de ataque. Resolverá el ataque y mostrará los mensajes necesarios.*

```
function turno(atacante, defensor, ataque) {
  if (defensor.vida <= 0) {
    console.log(`${defensor.nombre} ya ha sido derrotado.`);
    return;
  }
  ataque(atacante, defensor);
  if (defensor.vida <= 0) {
    console.log(`${defensor.nombre} ha caído en combate.`);
  } else {
    console.log(`${defensor.nombre} tiene ${defensor.vida} de vida restante.`);
  }
}
```

4. *Finalmente simularemos los turnos:*

```
// Ejemplo de uso:
const jugador1 = jugadores[0];
const jugador2 = jugadores[1];

turno(jugador1, jugador2, ataqueEspada);
turno(jugador2, jugador1, ataqueMagia);
turno(jugador1, jugador2, ataqueEspada);
turno(jugador2, jugador1, ataqueMagia);
```

Más adelante volveremos sobre este juego para darle una apariencia gráfica más amable.

3.1.2. Conversión de Fahrenheit a Celcious.

Ejemplo guiado

Conversión de Fahrenheit a Celcious. La fórmula de conversión es: C = (F – 32) * (5/9).
Crea fichero javascript con una función que al pasarle por parámetro la temperatura en grados Fahrenheit retorne la conversión a grados Celcius.
Al ejecutar el fichero por consola se le pasará como argumento la temperatura a convertir e imprimirá por consola la temperatura convertida.

1.- Primeramente validamos los datos de entrada

```
// Verificar si se proporcionó un argumento por consola
if (process.argv.length < 3) {
    console.log("Por favor, proporciona la temperatura en Fahrenheit como argumento.");
    console.log("Ejemplo: node fahrenheitToCelsius.js 75");
    process.exit(1);
}

// Obtener el argumento de la línea de comandos
const fahrenheitInput = process.argv[2];
const fahrenheit = parseFloat(fahrenheitInput);

// Validar que el input sea un número
if (isNaN(fahrenheit)) {
    console.log("Error: Por favor ingresa un número válido para la temperatura.");
    process.exit(1);
}
```

Aquí hemos usado pareFloat como alternativa a parseInt y Number.

También hemos usado la función isNaN(), que devuelve true si no e un número.

2.- Creamos la función para realizar el cálculos de lo grados

```
function fahrenheitToCelsius(fahrenheit) {
    return (fahrenheit - 32) * (5/9);
}
```

3.- Finalmente hacemos la llamada a la función y mostramos el resultado:

```
// Realizar la conversión
const celsius = fahrenheitToCelsius(fahrenheit);

// Mostrar el resultado formateado
console.log(`${fahrenheit}°F = ${celsius.toFixed(2)}°C`);
```

3.1.3. Ejercicio propuesto: calculo de edad

Haz un script de javascript para calcular la edad de una persona. La edad se calcula como la diferencia entre el año actual y el año de nacimiento de la persona.

Sesión 2

3.2. Ejemplo guiado

3.2.1. Receta de tortilla con callbacks

Ejemplo guiado

Vamos a trabajar asincronía en JavaScript usando callbacks y setTimeout sobre un flujo de pasos de cocina.

Crear la función prepararTortilla con los ingredientes: papas, cebolla, huevos, aceite, sal. Esta función Hace la llamada a los pasos para hacerla en forma de callback. Los pasos son:
- pelarPapas, cortarPapas, cortarCebolla, freirIngredientes, batirHuevos, mezclarIngredientes CocinarTortilla, servirTortilla

Cada función realiza un setTimeout de 2 segundos o el tiempo que estime.

Ante de comenzar lee el AIV.3. Callback para comprender el formato.

1.- Creamos primeramente

```
function prepararTortilla() {
    console.log("🏁 INICIANDO PREPARACIÓN DE TORTILLA DE PAPAS 🏁");
    console.log("    Ingredientes: papas, cebolla, huevos, aceite, sal");
    console.log("==============================================");

    // Iniciar la secuencia de pasos
    pelarPapas(() => {
        cortarPapas(() => {
            cortarCebolla(() => {
                freirIngredientes(() => {
                    batirHuevos(() => {
                        mezclarIngredientes(() => {
                            cocinarTortilla(() => {
                                servirTortilla();
                            });
                        });
                    });
                });
            });
        });
    });
}
```

Nota que cortarPapas es el callback de pelarPapas, cortarCebolla es el callback de cortarPapas así suceivamente.

2.- Ahora creamos cada función: pelarPapas

```
function pelarPapas(callback) {
    const ingrediente = "papas";
    console.log(`🥔 Paso 1: Pelando ${ingrediente}...`);

    setTimeout(() => {
        console.log(`✔ ${ingrediente} peladas correctamente`);
        callback();
    }, 2000);
}
```

Aquí lo destacable es la función setTimeout(función, tiempo). Donde función es algo a realizar y tiempo (en milisegudos, recuerda 1000 ms es 1s.) es lo que espero antes de ejecutar la función,

En nuestro caso esperamos 2s ante de ejecutar la función flecha:

```
console.log(`✔${ingrediente} peladas correctamente`);
callback();
```

Y como habrás apreciado, la función callback es cortarPapas.

3.- Ahora creamos cada función: cortarPapas

```
function cortarPapas(callback) {
    const ingrediente = "papas";
    console.log(`🔪 Paso 2: Cortando ${ingrediente} en rodajas...`);

    setTimeout(() => {
        console.log(`✔ ${ingrediente} cortadas en rodajas finas`);
        callback();
    }, 2500);
}
```

4.- Y así con las sucesivas:

Antes pasa por [AIV.5. Clousures]() para ver el tema de Clousure

```
// Paso 3: Cortar cebolla
function cortarCebolla(callback) {
    const ingrediente = "cebolla";
    console.log(`     Paso 3: Picando ${ingrediente}...`);

    // Uso de función flecha como callback
    setTimeout(() => {
        console.log(`✔ ${ingrediente} picada finamente (¡y llorando un poco!)`);
        callback();
    }, 1800);
}

// Paso 4: Freír ingredientes
function freirIngredientes(callback) {
    const ingredientes = ["papas", "cebolla"];
    console.log(`🍳 Paso 4: Friendiendo ${ingredientes.join(" y ")}...`);

    setTimeout(() => {
        console.log(`✔ ${ingredientes.join(" y ")} fritas y doraditas`);
        callback();
    }, 3000);
}

// Paso 5: Batir huevos
function batirHuevos(callback) {
    const ingrediente = "huevos";
    console.log(`◯ Paso 5: Batiendo ${ingrediente}...`);

    // Closure: la función interna mantiene acceso a 'ingrediente'
    const terminarDeBatir = () => {
        console.log(`✔ ${ingrediente} bien batidos con sal`);
        callback();
    };

    setTimeout(terminarDeBatir, 1500);
}

// Paso 6: Mezclar ingredientes
function mezclarIngredientes(callback) {
    const ingredientes = ["papas fritas", "cebolla", "huevos batidos"];
    console.log(`🥣 Paso 6: Mezclando ${ingredientes.join(", ")}...`);

    setTimeout(() => {
        console.log(`✔ Mezcla uniforme lista para cocinar`);
        callback();
    }, 2000);
}
```

Continúa….

```
// Paso 7: Cocinar tortilla
function cocinarTortilla(callback) {
    console.log("🔥 Paso 7: Cocinando la tortilla...");

    setTimeout(() => {
        console.log("✔ Tortilla dorada por ambos lados");
        console.log("🔃 Dando la vuelta a la tortilla...");

        // Segundo setTimeout anidado
        setTimeout(() => {
            console.log("✔ Tortilla volteada perfectamente");
            callback();
        }, 1500);
    }, 2500);
}

// Paso final: Servir
function servirTortilla() {
    console.log("🍽  Paso 8: Sirviendo la tortilla...");

    setTimeout(() => {
        console.log("================================================");
        console.log("🎉 ¡TORTILLA DE PAPAS LISTA PARA SERVIR! 🎉");
        console.log("😋 ¡Buen provecho!");
        console.log("================================================");
    }, 1000);
}

// Ejecutar la receta
console.log("👨‍🍳 RECETA DE TORTILLA DE PAPAS CON CALLBACKS 👨‍🍳");
console.log("================================================");

// Ejecutar la versión normal
prepararTortilla();
```

3.2.2. Calculadora realizada con funciones

Ejemplo guiado

Crear una calculadora genérica que reciba por consola dos números y la operación.
Al inicio se llamará a la funcion procesarComando() que comprobará el número de parámetros
Y llamará a una función como operar (num1, num2, operación), que devolverá el resultado.
La función “operar” validará: num1, num2 y operación. Si todo es correcto e llamará a la función,
por callback para realizar el cálculo.

1.- Definimos la función procesarComandos().

```
const procesarComandos = () => {
  const args = process.argv.slice(2); // Ignorar node y script path

  if (args.length !== 3) {
    console.log("Uso: node calculadora.js <numero1> <numero2> <operacion>");
    console.log("Operaciones disponibles: sumar, restar, multiplicar, dividir");
    return;
  }

  const [num1, num2, operacion] = args;

  try {
    const resultado = operar(num1, num2, operacion);
    console.log(`Resultado de ${operacion} ${num1} y ${num2}: ${resultado}`);
  } catch (error) {
    console.error("Error:", error.message);
  }
};
```

Nos aseguramos que tenemos 3 parámetro if (args.length !== 3) .

Al asegurarnos podemos extraer los parámetros de la forma:

const [num1, num2, operacion] = args;

Con todos los datos llamamos const resultado = operar(num1, num2, operacion);

Envuelto en un try catch para recoger cualquier error.

2.- Definimos la función operar(num1, num2, operacion)

```
// Función principal que recibe callback
const operar = (num1, num2, operacion) => {
  // Validar entradas
  const numero1 = validarNumero(num1);
  const numero2 = validarNumero(num2);
  validarOperacion(operacion);

  // Mapeo de operaciones a funciones
  const operaciones = {
    sumar: sumar,
    restar: restar,
    multiplicar: multiplicar,
    dividir: dividir,
  };

  // Ejecutar la operación (callback)
  return operaciones[operacion](numero1, numero2);
};
```

Como dijimos en el enunciado "operar" valida los números y la operación llamando a las funciones "validarNumero" y "validaOperacion" repectivamente. La vemos a continuación.

Ahora quiero que te fijes en el callback. En el return:

return operaciones[operacion](numero1, numero2);

operaciones[operacion] es un objeto al que se le pasa la key "operación".

Y el objeto "operaciones" es:

```
const operaciones = {
  sumar: sumar,
  restar: restar,
  multiplicar: multiplicar,
  dividir: dividir,
};
```

Es decir, si yo hago operaciones[sumar] me devuelve "sumar"

¿Y qué tengo justo después de return operaciones[sumar]? Pues (numero1, numero2)

Pero como operaciones[sumar] devolvió sumar tendré: return sumar(numero1, numero2);

¡Vaya! Esto es una llamada a la función sumar al que le paso los parámetros numero1 y numero2.

3.- Veamos lo métodos de las validaciones

```
// Funciones de validación
const validarNumero = (valor) => {
  const numero = Number(valor);
  if (isNaN(numero)) {
    throw new Error(`"${valor}" no es un número válido`);
  }
  return numero;
};

const validarOperacion = (operacion) => {
  const operacionesValidas = ["sumar", "restar", "multiplicar", "dividir"];
  if (!operacionesValidas.includes(operacion)) {
    throw new Error(
      `Operación "${operacion}" no válida. Use: ${operacionesValidas.join(
        ", "
      )}`
    );
  }
};
```

4.- Funcione de las operaciones: sumar, retar, etc

```
// Funciones de operación (usando arrow functions)
const sumar = (a, b) => a + b;
const restar = (a, b) => a - b;
const multiplicar = (a, b) => a * b;
const dividir = (a, b) => {
  if (b === 0) {
    throw new Error("No se puede dividir por cero");
  }
  return a / b;
};
```

Estas son las funcione llamadas con return operaciones[operacion](numero1, numero2); de la función “operar”.

3.2.3. Ejercicio propuesto: piedra, papel y tijera

Implementar el clásico piedra-papel-tijera en consola o usando prompt(), organizando toda la lógica con funciones y usando Math.random() para el turno del PC.

Pistas de uso
Funciones → Se define jugar() con lógica pura
Función flecha → Puede usarse para definir la lógica interna (const elegir = () => ...)
Scope → El juego tiene variables locales para usuario y pc
Clousure → Añadir contador de partidas jugadas

Sesión 3

Tarea a desarrollar en clase

ANEXO IV. Funciones (función flecha, callbacks, scope, clousures)

AIV.1. Funciones

Una función es un bloque que contiene un código a ejecutar que puede recibir parámetros. Las funciones pueden o no devolver parámetros.

La definición se hace con la palabra reservada **function**:

```
function miFuncion([parámetros de entrada){
        (código a ejecutar)
        [return parámetros de salida;]
}
```

La invocación de la función es llamándola por su nombre:

```
miFuncion([parámetros de entrada);
```

En caso de devolver algo se lo asignamos a una variable:

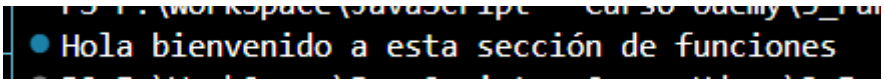
```
var valores = miFuncion([parámetros de entrada);
```

Veamos un ejemplo de una función que no recibe parámetros ni devuelve nada:

```
function bienvenida(){
    console.log("Hola bienvenido a esta sección de funciones");
}

bienvenida();
```

Si desde una terminal, dentro del proyecto, ejecuto el fichero con la sentencia "node app.js" tendré como resultado:



El mismo ejemplo, pero haciendo que devuelva el mensaje sería:

```
function bienvenida(){
    console.log(“Hola bienvenido a esta sección de funciones”);
}

var mensaje = bienvenida();
console.log(mensaje);
```

Obteniendo el mismo resultado.

AIV.1.a. Argumentos de entrada de una funciones

Son los parámetros que se le pasa a una función. Veamos un ejemplo:

```
function cuadradoNumero(numero){
   return numero*numero;
}

var num = 3;
var result = cuadradoNumero(num);
console.log(result);
```

Hemos creado la función y una variable, num, con el valor 3.
Llamamos a la función pasando la variable num: cuadradoNumero(num)

La respuesta de la función se guarda en la variable result, 9.

También podría haber hecho directamente:

```
console.log(cuadradoNumero(num));
```

AIV.1.b. Uso de funciones como expresiones

Es una forma de exprear la sintaxis de una función de manera distinta. Su sintaxis, haciendo uso de un ejemplo, es:

```
var prueba = function(){
        console.log(“Esto es una prueba.”);
}
```

Como puedes ver hay tres grandes diferencias:

- La función no tiene nombre
- La función se asigna a una variable
- La llamada se hace a la variable, pero con los ().

Estamos habituados a mostrar mensajes compuestos concatenando cadenas de la siguiente forma:

```
console.log("Tienes " + currentAge(yearBirthday) +  " años.");
```

Hay otra forma de mostrar la cadena de salida, haciendo uso de las bondades de javascript, el Template Screen sería:

```
console.log(`Tienes ${currentAge(yearBirthday)} años.`);
```

Nótese que las comillas son las acostada a la derecha y que la llamada a la función va entre llaves, precedido del símbolo $, es la manera de insertar variables o resultado de funciones de javascript en esta notación.

AIV.1.c. Argumentos undefine

Es cuando yo la creo una variable pero no le asigno valor y esta es pasada a una función.

```
var nombre;
```

Veamos el ejemplo:

```
var nombre;
var prueba = function(n){
   return 'Hola ' + n;
}
console.log(prueba(nombre));
```

Devuelve:

```
Hola undefined                                                    app.js:76
```

Javascript no devuelve error. Solo indica que no tiene un valor, undefine.

En este caso hay un puntero con una dirección de memoria pero que no contiene valor.

AIV.1.d. Argumentos nulos

En este caso la varible pasada es de valor nulo. Veamos que sucede:

```
var a;
a = null;

var valorNulo = function(n){
    return n
}
console.log(valoNulo(a));
```

Esto devuelve:

```
null                                                              app.js:87
```

En este caso hay un puntero con una dirección de memoria pero que contiene valor null. Es decir null se considera un valor y es legítimo usarlo en comparaciones.

AIV.1.e. Argumentos por defecto

Imaginemos que tenemos:

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

console.log(sumar(10, 4));
```

Esto da como resultado:

```
NaN                                                               app.js:96
```

¿Por qué sucede esto? Porque c tiene un valor null y su tipo no es igual a y b.

Lo que puedo hacer es dar un valor por defecto de la siguiente manera:

```
var sumar = function(a, b, c=3){
   return a +b + c;
}
console.log(sumar(10, 4));
```

En la función yo doy un valor a la variable c. Si javascript que c no se le está pasando como parámetro le dará este valor de 3 y ejecutará el código. En este caso da como resultado:

```
17                                                    app.js:96
```

Pero si yo hiciese:

```
var sumar = function(a=5, b=3, c=3){
   return a +b + c;
}

console.log(sumar(10, null, 7));
```

Da como resultado:

```
17                                                    app.js:96
```

¿Por qué 17 y no 11? Porque null es considerado como valor 0.

AIV.1.f. Función anónima

Las funciones anónimas son funciones que no tienen un nombre definido. Se usan comúnmente en callbacks, IIFE (Immediately Invoked Function Expressions), closures y como argumentos de otras funciones.

Un ejemplo sencillo sería:

```
// Función anónima asignada a una variable
const saludar = function() {
  console.log("Hola mundo");
};
saludar(); // "Hola mundo"
```

Más adelante veremos el uso en callbacks e IIFE.

AIV.2. Función flecha

La función flecha es artificio que nos permite crear funciones un poco más dinámicas y que nos permiten ahorrar cierto código. Sus características son:

- No tienen su propio this (toman el this del contexto superior)
- No tienen arguments propio
- Muy útiles como callbacks

Veamos un ejemplo. Dado el año de nacimiento calcula la edad.

A lo que estamos habituado es:

```
Const years = [200, 2005, 2008, 2012];

var edad = years.map(function(el){
        return 2019 – el;
});

console.log(edad);
```

Lo anterior devuelve: 19, 14, 11, 7.

Haciendo uso de la función flecha:

```
const years = [200, 2005, 2008, 2012];

let edad = years.map(el => 2019 – el);

console.log(edad);
```

Devolviendo lo mismo.

En el caso de tener vario parámetros lo podremos entre paréntesis separados por comas:

```
const sumar = (a, b) => a + b;
```

En el caso de tener que ejecutar varias líneas las pondremos entre llaves:

```
const saludo = (nombre) => {
  let mensaje = "Hola " + nombre;
  return mensaje;
};
```

AIV.3. Callback

Un callback es una función que se pasa como argumento a otra función, y se ejecuta después.

Veámoslo mejor con un ejemplo:

```
function hacerAlgoDespues(nombre, callback) {
  console.log("Preparando saludo...");
  callback(nombre);
}

hacerAlgoDespues("Carlos", function(nombre) {
  console.log("Hola, " + nombre + " :)");
});
```

Vemos que declaramos una función llamada hacerAlgoDespues() al que se le pasa como segundo parámetro una función que en el cuerpo de la función es llamada pasándole “nombre”.

Más abajo invocamos a la función pasándole un nombre y una función anónima que muestra por consola “Hola Carlos”.

Se puede usar la función flecha cuando invocamos a hacerAlgoDespues(), sería:

```
hacerAlgoDespues("Carlos", nombre => console.log("Hola, " + nombre));
```

AIV.4. Scope

Hace referencia al ámbito de la variable. Esta puede ser:

- **Global**: fuera de cualquier función o bloque.
- **De función**: dentro de una función.
- **De bloque**: dentro de `{}` con `let` o `const`.

Un ejemplo donde podemos ver los tres tipos sería el siguiente:

```
let x = 10; // global

function demo() {
  let y = 20; // local
  if (true) {
    let z = 30; // de bloque
    console.log(x, y, z); // Esto funcionará
  }
    console.log(z); //  Esto dará Error
}
```

AIV.5. Clousures

Un closure es cuando una función interna que recuerda el contexto (variables) en el que fue creada.

Un ejemplo típico es el siguiente:

```
function crearContador() {
  let contador = 0;

  return function() {
    contador++;
    console.log(contador);
  };
}

const contar = crearContador();
contar(); // 1
contar(); // 2
```

Aunque crearContador() terminó su ejecución, la función interna sigue teniendo acceso a contador, sigue manteniendo el último valor.