

# JavaScript

## Semana 2



# Índice

Resultado de aprendizaje, contenido y criterios de evaluación.....	4
Semana 2. Fundamentos de JavaScript (II).....	6
Sesión 1.....	6
2.1. Ejemplo guiado.....	6
2.1.1 Gestión de puntuación de jugadores.....	6
2.1.2 Gestión de películas.....	7
2.2. Ejemplo propuesto: control de alimentos en casa.....	12
Sesión 2.....	13
2.3. Ejemplo guiado.....	13
2.3.1 Gestión de estudiantes de Hogwarts.....	13
2.3.2 Inventario de la nave espacial Incom T-70.....	16
2.4. Ejemplo propuesto: gestión de comandas.....	22
Sesión 3.....	23
2.5. Ejemplo guiado.....	23
2.5.1 Imprimir información de Jedis.....	23
2.5.1 Clasificación de Jedis.....	28
2.6. Ejemplo propuesto: Gestión de tienda.....	31
ANEXO II. Variables, tipos de datos, asignaciones, operaciones, comentarios, sentencias, decisiones y bucles.....	32
AII.1. Variables.....	32
AII.2. Tipos de datos.....	32
AII.2.a Conversiones de datos.....	33
Conversión a String.....	33
Conversión a Number, parseInt, parseFloat.....	33
Conversión a Boolean.....	34
Conversión a BigInt.....	34
AII.3. Asignaciones.....	34
AII.4. Operaciones.....	34
AII.4.a. Aritméticas.....	34
AII.4.b. Desplazamiento de bits.....	35
AII.4.c. Operaciones bit a bit.....	35
AII.5.d. Operadores lógicos.....	35
AII.5.e. Type of.....	36
AII.5.f. Operadores Unarios.....	36
AII.5. Comentarios al código.....	36
AII.6. Sentencias.....	37
AII.7. Decisiones.....	37
AII.7.a. if...else.....	37
AII.7.b. Operador ternario.....	39
AII.7.c. Switch.....	40
AII.8. Bucles.....	41
AII.8.a. for.....	41
AII.8.b. for/in.....	43
AII.8.c. for/of.....	43
AII.8.d. while.....	43
AII.8.e. do...while.....	44

ANEXO III. Arrays, objetos y contexto this.....	45
AIII.1. Arrays.....	45
AIII.1.a. Métodos comunes.....	45
AIII.1.b. Modificación de los elementos.....	45
AIII.1.c. Encontrar un elemento, indexOf() y findIndex().....	46
AIII.1.d. Filtrar arreglo según una condición. Filter().....	47
AIII.1.e. Ordenación. sort().....	48
AIII.1.f. Acumulación. reduce().....	49
AIII.1.g. Acciones sobre array sin mutarlo. Map().....	51
AIII.2. Objetos.....	51
AIII.2.a. Declaración de objetos.....	51
AIII.2.b. Creación de objeto.....	52
AIII.2.c. Acceso y modificación.....	52
AIII.3. El contexto this.....	52

## 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
2 - Tipos de datos. Conversiones. 3 - Asignaciones. 5 - Comentarios al código. 6 - Sentencias. 8 - Bucles.
Criterios de evaluación
b) Se han utilizado los distintos tipos de variables y operadores disponibles en el lenguaje. c) Se han identificado los ámbitos de utilización de las variables d) Se han reconocido y comprobado las peculiaridades del lenguaje respecto a las conversiones entre distintos tipos de datos 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
1 - Utilización de objetos. Objetos nativos del lenguaje.
Criterios de evaluación
a) Se han identificado los objetos predefinidos del lenguaje d) Se han generado textos y etiquetas como resultado de la ejecución de código en el navegador. e) Se han escrito sentencias que utilicen los objetos predefinidos del lenguaje para interactuar con el usuario h) Se ha depurado y documentado el código

<b>Resultado de aprendizaje</b>
R4. Programa código para clientes web analizando y utilizando estructuras definidas por el usuario
<b>Contenido</b>
1 - Funciones predefinidas del lenguaje 3 - Matrices (arrays).
<b>Criterios de evaluación</b>
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. i) Se ha creado código que haga uso de objetos definidos por el usuario k) Se ha depurado y documentado el código

## Semana 2. Fundamentos de JavaScript (II)

Hasta ahora tan solo podemos guardar un valor en una variable. Pero si queremos guardar varios valores, por ejemplo, la temperatura de un sensor durante un año. Pues bien, el elemento que nos permite hacerlo es el Array y te invito ver el anexo [AIII.1. Arrays](#) para ver su fundamento.

### Sesión 1

#### 2.1. Ejemplo guiado

Vamos a introducir el tema con el siguiente ejemplo guiado.

##### 2.1.1 Gestión de puntuación de jugadores



#### Ejemplo guiado

Crear un sistema que gestione puntuaciones de jugadores usando arrays y objetos.

1. Crea un archivo nuevo en Visual Studio Code llamado *gestionarPuntuacion.js*
2. Creamos un array de jugadores con su puntuación (profundiza en el tema de arrays en el anexo [AIII.1. Arrays](#)):



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

3. Creamos un método para mostrar las puntuaciones, lo llamaremos *mostrarPuntuaciones()*. Este método recorrerá el array *jugadores* a través del método “*forEach()*” (visita el método [AIII.1.a. Métodos comunes](#) para profundizar”:



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

function mostrarPuntuaciones() {
  jugadores.forEach(jugador => {
    console.log(jugador.nombre + ": " + jugador.puntos + " puntos");
  });
}

mostrarPuntuaciones();
```

*Nota que el `forEach` almacena, en cada pase, el array en la variable `jugador` que con la función flecha “=>” accedemos a los atributos del array (nombre y puntos).*

## 2.1.2 Gestión de películas



### Ejemplo guiado

Gestionemos una lista de películas usando únicamente JavaScript puro (sin frameworks ni librerías externas).

Añadir películas:  
Añada la película al array (usando `array.push()`).

Eliminar películas:  
Cree un método que seleccione una película dado el índice.  
Después borra la película activa (usando `splice`).  
Si no hay selección, muestra una alerta.

Mostrar películas:  
Cada vez que el array se actualice (añadir o eliminar), se recorrerá el array con un `forEach()` para actualizar la visualización.

1.- Creamos el array `películas` y la variable `seleccionada` (ue guardará el índice de la película seleccionada).



```
// Lista de películas
let peliculas = [];
let seleccionada = null;
```

2.- Creamos el método **añadirPelícula** al que le paso el nombre.

Antes de nada comprobamos que nombre no esté vacío, para ello debe cumplirse una de las siguientes casos:

- `!nombre` → detecta si nombre es un valor falsy (`undefined`, `null`, `""`, `0`, `false`, `NaN`). Y que nombre no sea una cadena vacía.
- `nombre.trim() === ""` → Con trim elimino espacio antes y después de la cadena y la cadena que queda compruebo que no está vacía.

En caso de que se cumpla alguna emitimos un mensaje de error y retornamos, no seguimos ejecutando el código.


```
if (!nombre || nombre.trim() === "") {  
  console.log("Error: el nombre de la película no puede estar vacío");  
  return;  
}
```

Si todo va bien añadimos la película al array.

A continuación, emitimos un mensaje de Película añadida.

Y llamamos al método `mostrarPelículas` para mostrar por consola todo el array de películas.

El código queda como:



```
// Añadir película  
function añadirPelícula(nombre) {  
  // Compruebo que nombre está vacío  
  if (!nombre || nombre.trim() === "") {  
    console.log("Error: el nombre de la película no puede estar vacío");  
    return;  
  }  
  peliculas.push(nombre.trim());  
  console.log(`Película añadida: "${nombre}"`);  
  mostrarPelículas();  
}
```



#### **``Película añadida: "${nombre}"``**

Esta forma de escribir cadenas se llama “template literals” y se delimitan entre las comillas backticks (```).

Esta forma permite la **interpolación** de variables con `$( )`, es decir pasamos variables a la cadena encapsulado con en el símbolo `$( )`.

Por ejemplo:

```
let nombre = "Matrix";  
console.log(`Película añadida: "${nombre}"`); // Película añadida "Matrix".
```



3.- Antes de eliminar una película debemos seleccionarla. Así que, hagamos primero la función **seleccionarPelícula** al que se le pasa el índice:

Lo primero que tenemos que comprobar es si el parámetro índice es lo que esperamos. Y lo que esperamos es un índice de un array (el de películas) que debe estar entre 0 y la longitud del array.

Es decir, que si el índice es menor que 0 y mayor que la longitud del array debemos indicar que hay un error y detenemos la ejecución.

```
if (indice < 0 || indice >= peliculas.length) {  
  console.log("Índice no válido");  
  return;  
}
```

Si el índice es correcto, lo siguiente a hacer es comprobar si:

- seleccionamos una películas → la variable seleccionada es distinto de índice
- deseccionamos una película → la variable seleccionada es distinto de índice

```
if (seleccionada === indice) {  
  console.log(`Película deseleccionada: "${peliculas[indice]}"`);  
  seleccionada = null;  
} else {  
  seleccionada = indice;  
  console.log(`Película seleccionada: "${peliculas[indice]}"`);  
}
```



```
// Seleccionar/deseleccionar película por índice  
function seleccionarPelícula(indice) {  
  if (indice < 0 || indice >= peliculas.length) {  
    console.log("Índice no válido");  
    return;  
  }  
  if (seleccionada === indice) {  
    console.log(`Película deseleccionada: "${peliculas[indice]}"`);  
    seleccionada = null;  
  } else {  
    seleccionada = indice;  
    console.log(`Película seleccionada: "${peliculas[indice]}"`);  
  }  
  mostrarPeliculas();  
}
```

#### 4.- Continúaemos con **eliminarSeleccionada**:

Antes de nada comprobemos que la variable seleccionada no sea nula, en dicho caso emitimos un mensaje de que no hay película seleccionada y detemos la ejecución con el return.

```
if (seleccionada === null) {  
  console.log("No hay ninguna película seleccionada para eliminar");  
  return;  
}
```

En caso de que seleccionada tenga un valor mostramos un mensaje de la película a eliminar, luego la eliminamos con **splice** y mostramos el array de películas con la función mostrarPeliculas.

```
console.log(`Eliminada: "${peliculas[seleccionada]}"`);  
peliculas.splice(seleccionada, 1);  
seleccionada = null;  
mostrarPeliculas();
```



**peliculas.splice(seleccionada, 1);**

**splice** es un método de Array y permite añadir, eliminar o reemplazar elementos.

Por ejemplo: `let numeros = [1, 2, 3, 4, 5];`

`// Eliminar  
numeros.splice(2, 1); // elimina el "3"`

`// Insertar  
numeros.splice(2, 0, 99); // inserta 99 en la posición 2`

`// Reemplazar  
numeros.splice(1, 2, 7, 8); // reemplaza 2 elementos desde índice 1 por 7 y 8`

En nuestro ejercicio se elimina **1** elemento del array peliculas en la posición indicada por **seleccionada**.

El código queda;



```
// Eliminar película seleccionada
function eliminarSeleccionada() {
  if (seleccionada === null) {
    console.log("No hay ninguna película seleccionada para eliminar");
    return;
  }
  console.log(`Eliminada: "${películas[seleccionada]}"`);
  películas.splice(seleccionada, 1);
  seleccionada = null;
  mostrarPelículas();
}
```

5.- Implementemos por último la función `mostrarPelículas()`.

Esta tan solo ha de recorrer el array y mostrarlas. Ysi está seleccionada lo indicaremos con texto. Así queda el código:



```
// Mostrar películas
function mostrarPelículas() {
  console.log("Listado de películas:");
  películas.forEach((película, i) => {
    if (i === seleccionada) {
      console.log(`-> [${i}] ${película} (SELECCIONADA)`);
    } else {
      console.log(` [${i}] ${película}`);
    }
  });
  if (películas.length === 0) {
    console.log("vacío");
  }
}
```

6.- Finalmente probamos el código: añadimos unas películas, seleccionamos una y la eliminamos.



```
// Probando el código //

añadirPelícula("Matrix");
añadirPelícula("Inception");
añadirPelícula("Interstellar");

seleccionarPelícula(1); // Selecciona "Inception"
eliminarSeleccionada(); // Elimina "Inception"

seleccionarPelícula(0); // Selecciona "Matrix"
seleccionarPelícula(0); // La deselecciona
```

## 2.2. Ejemplo propuesto: control de alimentos en casa



Imagina que quieres controlar lo que hay en la nevera y, cuando consumes un alimento, añadirlo automáticamente a la lista de la compra para reponerlo.

### Requisitos

- Define un array llamado nevera con algunos alimentos iniciales (ejemplo: "leche", "huevos", "queso").
- Define un array vacío llamado listaCompra.
- Implementa una función para consumir un alimento:
  - + El alimento se elimina del array nevera.
  - + El alimento se añade al array listaCompra.
- Implementa una función para mostrar el estado actual de ambos arrays en consola.

Probamos al final que todo funciona.

## Sesión 2

### 2.3. Ejemplo guiado

Las variables son elementos sencillos, pero cuando queremos guardar datos como datos de un alumno y repetir estos por cada alumno, no es útil el uso de variables. Para tal fin en JavaScript se usan los Objetos. Echa un vistazo al anexo [AIII.2. Objetos](#) para abordar las siguientes actividades para practicar.

Adelantarte que en JavaScript se puede hacer referencia a un objeto a través de “this”, échale un ojo al anexo [AIII.3. El contexto this](#). Ya te dejo con la actividad.

#### 2.3.1 Gestión de estudiantes de Hogwarts



##### Ejemplo guiado

Crear un fichero javascript que permita gestionar estudiantes del Colegio Hogwarts, utilizando objetos en JavaScript y el concepto de this para que cada estudiante pueda presentarse.

Lógica del Programa:

Cada estudiante debe ser un objeto con las propiedades:

- name: Nombre del estudiante.
- house: Casa asignada.

Y debe tener un método present() que use this para mostrar un mensaje como: "¡Soy [nombre] de la casa [casa]! Mi orgullo es evidente."

Habrán un array de estudiantes.

Los estudiantes se agregarán al array a través del método agregar estudiantes.

Podremos listar de estudiantes debe mostrar:

- Nombre y casa de cada uno.

1.- Primero creamos el objeto Estudiante. En javascript se usa las Funciones Constructoras. Y sería algo así:



```
// Definición de una Función Constructora de Estudiante (actúa como plantilla)
function Estudiante(nombre, casa, edad) {
  this.nombre = nombre;
  this.casa = casa;
  this.edad = edad;

  this.presentarse = function () {
    console.log(`Hola, soy ${this.nombre}, tengo ${this.edad} años y pertenezco a la casa ${this.casa}.`);
  };
}
```



**function Estudiante(nombre, casa, edad)**

Las Funciones Constructoras existen desde el inicio del lenguaje (ECMAScript 1, 1997). Y es como una plantilla.

Posteriormente con la ECMAScript 2015 (ES6) se introdujo la sintaxis de **class**. Nuestra función constructora del ejercicio se podría escribir así:

```
class Estudiante() {  
  constructor (nombre, casa, edad) {  
    this.nombre = nombre;  
    this.casa = casa;  
    this.edad = edad;  
  }  
  
  // método  
  presentarse () {  
    console.log(`Hola, soy ${this.nombre}, tengo ${this.edad} años y pertenezco a la casa $  
      {this.casa}.`);  
  };  
}
```

En futuros ejercicios usaremos class.

2.- Declaramos el array de estudiantes:

```
const estudiantes = [];
```

3.- Creemos la función agregarEstudiante a la que se le pasa nombre, casa y edad.

Al array estudiantes debemos ir añadiendo los estudiantes que son objetos.

El objeto se crea así:

```
const nuevo = New Estudiante(nombre, casa, edad);
```

Y este nuevo se añade al array con push.



```
function agregarEstudiante(nombre, casa, edad) {  
  const nuevo = new Estudiante(nombre, casa, edad);  
  estudiantes.push(nuevo);  
  console.log(`Estudiante añadido: ${nombre} (${casa})`);  
}
```

4.- Ahora listaremos a los estudiantes a través de la función `listarEstudiantes`.

Es tan solo recorrer el array `estudiantes` y mostrar el mensaje:



```
function listarEstudiantes() {
  console.log("Listado de estudiantes de Hogwarts:");
  estudiantes.forEach((e, i) => {
    console.log(`${i + 1}. ${e.nombre} - Casa: ${e.casa}, Edad: ${e.edad}`);
  });
}
```

5.- Terminamos haciendo la función `todosSePresentan`.

Aquí recorreremos el array `estudiantes`. Recuerda que cada elemento del array es el objeto `Estudiante` que ya tiene la función de `presentarse`. Es por ello el uso de `e.presentarse()`.



```
function todosSePresentan() {
  estudiantes.forEach(e => e.presentarse());
}
```



**`estudiantes.forEach(e => e.presentarse());`**

Recuerda que al recorrer el array "e" va tomando los valores que se van recorriendo en el array. Y cada elemento "e" es un objeto (Función Constructora) `Estudiante`, por lo que a través de "." tengo acceso a los elementos del objeto, entre ellos la función "presentarse".

6.- Pruebas del ejercicio.

Añadimos alumnos, los listamos y luego que se presente:



```
// PRUEBAS
agregarEstudiante("Harry Potter", "Gryffindor", 11);
agregarEstudiante("Draco Malfoy", "Slytherin", 11);
agregarEstudiante("Luna Lovegood", "Ravenclaw", 12);

listarEstudiantes();

console.log("Presentaciones:");
todosSePresentan();
```

### 2.3.2 Inventario de la nave espacial Incom T-70



#### Ejemplo guiado

Desarrollar fichero javascript que gestione el inventario de la nave espacial Incom T-70 (el modelo de X-wing usado en la Resistencia).

Estructura de datos:

Usar un array de objetos llamado inventario

Cada producto debe tener: nombre (string), precio (number válido) y cantidad (number válido)

Funcionalidades:

Crear un menú para que el usuario pueda hacer las siguientes gestiones funcionales:

- `mostrarInventario()`: Actualiza ambas vistas
- `calcularTotal()`: Devuelve el valor total del inventario
- `agregarProducto()`: Con validación de datos
- `eliminarProducto()`: Por nombre exacto

Validaciones:

Precio y cantidad deben ser números válidos ( $\geq 0$ )

Mensajes de error descriptivos

Prevenir entradas vacías

1. En este ejercicio vamos a hacer un uso alternativo a `prompt-sync`. Usaremos el módulo nativo de Node “`readline`”.

*Así que lo primero que haremos es requerir esas librerías:*



```
const readline = require("readline");
```

*Lo siguiente a hacer es crear una interface para crear el flujo de entrada y salida de datos por la consola:*



```
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
```



## 2.- Creamos el inventario con valores por defecto



```
let inventario = [
  { nombre: "Cañón láser", precio: 1500, cantidad: 4 },
  { nombre: "Hipermotor", precio: 5000, cantidad: 2 },
  { nombre: "Escudo deflector", precio: 3200, cantidad: 3 },
  { nombre: "Droide astromecánico", precio: 2500, cantidad: 1 },
  { nombre: "Torreta de combate", precio: 1800, cantidad: 2 },
];
```

## 3.- Menu principal. Lo creo dejando comentado las llamadas a las funciones que iremos implementando.



```
function showMenu() {
  console.log(
    ===== MENÚ INVENTARIO INCOM T-70 =====
    1) Mostrar inventario (ambas vistas)
    2) Calcular total
    3) Agregar producto
    4) Eliminar producto por nombre exacto
    0) Salir
  );
}

rl.question("Opción: ", (opcionRaw) => {
  const opcion = opcionRaw.trim();
  switch (opcion) {
    case "1":
      // mostrarInventario();
      return showMenu();
    case "2":
      /*
      console.log(
        `Total del inventario: ${calcularTotal().toFixed(2)} créditos`
      );
      */
      return showMenu();
    case "3":
      // return agregarProducto(showMenu);
    case "4":
      // return eliminarProducto(showMenu);
    case "0":
      console.log("Fin.");
      rl.close();
      return;
    default:
      console.log("Opción no válida. Introduce 0, 1, 2, 3 o 4.");
      return showMenu();
  }
});
showMenu();
```

Para el menú pintamos con un `console.log` las opciones que se piden en el ejercicio.

Luego esperamos a que el usuario introduzca una opción por consola- Su captura lo haremos a través de `r1`, que definimos como interface. La manera, aprovechando la forma de función flecha, es:

```
r1.question("Opción: ", (optionRaw) => {...});
```

- `question` → Solicita dato por consola al usuario.
- `"Opción: "` → Será el texto que le saldrá al usuario por la consola.
- `OptionRaw` → Es la variable donde se guarda el dato introducido por el usuario.
- `=> {...}` → Es el contenido a hacer según la opción recibida.

La estructura de elección más idónea para este ejercicio es un Switch. ([Anexo AII.7.c. Switch](#)).

Para evitar un While se usa la llamada a sí mismo `showMenu()`.

4.- Creemos lo métodos de validación, pues los usaremos más adelante.



```
function validarNombre(nombre) {  
  if (!nombre || typeof nombre !== "string" || nombre.trim() === "") return "El nombre no puede estar vacío.";  
  return null;  
}  
function validarNumeroEtiqueta(valorStr, etiqueta) {  
  if (valorStr === undefined || valorStr === null) return `${etiqueta} es obligatorio.`;  
  const n = Number(valorStr);  
  if (!Number.isFinite(n)) return `${etiqueta} debe ser un número válido.`;  
  if (n < 0) return `${etiqueta} debe ser ≥ 0.`;  
  return null;  
}
```

Fíjate que en `validarNumeroEtiqueta(valorStr, etiqueta)`, el valor “etiqueta” es para indicar si es “Precio” o “Cantidad”.

5.- Realicemos `mostrarInventario` que a su vez llama a `formatearInventario` para mostrarlo con cierto texto:



```
function mostrarInventario() {  
  console.log("\n==== Vista tabular ====");  
  if (inventario.length === 0) console.log("Inventario vacío.");  
  else console.table(inventario, ["nombre", "precio", "cantidad"]);  
  console.log("\n==== Vista texto ====");  
  console.log(formatearInventarioTexto());  
}
```

6.- Veamos *formatearInventario*.

```
function formatearInventarioTexto() {
  const sep = "-".repeat(60);
  let out = `${sep}\nINVENTARIO INCOM T-70 (vista texto)\n${sep}\n`;
  if (inventario.length === 0) {
    out += "Inventario vacío.\n" + sep;
    return out;
  }
  inventario.forEach((p, i) => {
    out += `${i + 1}. ${p.nombre}\n Precio: ${p.precio} cr |
            Cantidad: ${p.cantidad} uds | Subtotal: ${(p.precio * p.cantidad).toFixed(2)} cr\n`;
  });
  out += `${sep}\nTOTAL: ${calcularTotal().toFixed(2)} créditos\n${sep}`;
  return out;
}
```

7.- Fíjate que en el cálculo del total se llama a la función *calcularTotal*



```
function calcularTotal() {
  return inventario.reduce((acc, p) => acc + p.precio * p.cantidad, 0);
}
```

Recordando lo visto en el anexo [AIII.1.f. Acumulación. Reduce\(\)](#) recorreremos inventario acumulando la suma de los productos de “precio \* cantidad”.

## 8.- Ahora creemos el método agregarProducto



```
function agregarProducto(done) {
  console.log("\n[Agregar producto]");
  rl.question("Nombre: ", (nombreRaw) => {
    const nombre = nombreRaw.trim();
    const e1 = validarNombre(nombre);
    if (e1) {
      console.log(`Error: ${e1}`);
      return done();
    }
    rl.question("Precio (≥0): ", (precioStr) => {
      const e2 = validarNumeroEtiqueta(precioStr, "Precio");
      if (e2) {
        console.log(`Error: ${e2}`);
        return done();
      }
      rl.question("Cantidad (≥0): ", (cantidadStr) => {
        const e3 = validarNumeroEtiqueta(cantidadStr, "Cantidad");
        if (e3) {
          console.log(`Error: ${e3}`);
          return done();
        }
        const precio = Number(precioStr);
        const cantidad = Number(cantidadStr);
        inventario.push({ nombre, precio, cantidad });
        console.log(`Producto añadido: "${nombre}" (${precio} cr, ${cantidad} uds).`);
        done();
      });
    });
  });
}
```

## 9.- Eliminar productos



```
function eliminarProducto(done) {
  console.log("\n[Eliminar producto por nombre exacto]");
  if (inventario.length === 0) {
    console.log("No hay productos para eliminar.");
    return done();
  }
  rl.question("Nombre exacto a eliminar: ", (nombreRaw) => {
    const nombre = nombreRaw.trim();
    const e = validarNombre(nombre);
    if (e) {
      console.log(`Error: ${e}`);
      return done();
    }
    const idx = inventario.findIndex((p) => p.nombre === nombre);
    if (idx === -1) {
      console.log(`No se encontró un producto con nombre EXACTO "${nombre}".`);
      return done();
    }
    const eliminado = inventario.splice(idx, 1)[0];
    console.log(`Eliminado: "${eliminado.nombre}".`);
    done();
  });
}
```

## 2.4. Ejemplo propuesto: gestión de comandas



Desarrollar una aplicación JavaScript que permita a un camarero gestionar comandas de forma intuitiva, con las siguientes funcionalidades.

Funcionalidades Requeridas:

Gestión de Menús:

Los platos disponibles se almacenarán en tres arrays independientes:

- Primeros platos
- Segundos platos
- Postres

Habrà otro array de Comanda donde se almacena los platos elegidos

Funcionalidad:

- Se mostrarà el menú con el array de primeros platos.
- Se podrá hacer una elección múltiple de primeros platos y se guardarán en el array Comanda.
- Se mostrarà el array de segundos.
- Se podrá hacer una elección múltiple de segundos platos y se guardarán en el array Comanda.
- Se mostrarà el array de postres.
- Se podrá hacer una elección múltiple de postres y se guardarán en el array Comanda.

Postres

- Se cargará y mostrarà el array de primeros platos a elegir
- El usuario podrá elegir lo que quiera. Estos se guarda en el array Comanda.

Emisión de Comanda:

- Se mostrarà el array de comanda con todos los platos elegidos

## Sesión 3

### 2.5. Ejemplo guiado

#### 2.5.1 Imprimir información de Jedis

##### Ejemplo guiado

Crear un fichero javascript que permita gestionar e imprimir información de Jedis almacenada en un array de objetos, utilizando los principales métodos de arrays en JavaScript.

Estructura de Datos:

Crear un array llamado jedis que contenga al menos 5 objetos.

Cada objeto representará un Jedi y debe tener las siguientes propiedades:

- nombre (string): Nombre completo del Jedi
- edad (number): Edad en años
- sistema (string): Sistema estelar de ubicación
- planeta (string): Planeta de origen o residencia
- email (string): Código de contacto (ej: usuario@templo.jedi)

Funcionalidades a Implementar:

Mostrar Todos los Jedis:

- Usar `forEach()` para recorrer el array y mostrar por consola:  
Nombre: [nombre], Edad: [edad], Contacto: [email]
- Mostrar los datos en formato tabla con `console.table()`

Filtrar Jedis Mayores de 18 Años:

- Usar `filter()` para crear un nuevo array solo con Jedis mayores de 18
- Mostrar solo sus nombres

Búsqueda por Nombre:

- Crear función `buscarJedi(nombre)` que use `find()` para devolver el objeto Jedi si existe.

Crear un menú para poder elegir entre las acciones:

- Mostrar Todos los Jedis.
- Filtrar Jedis Mayores de 18 Años
- Búsqueda por Nombre

Formatear Información:

- Usar `map()` para generar frases con el formato:  
"[Nombre] ([edad] años) -- Sistema [sistema] -- Planeta [planeta] -- Contacto: [email]"

## 1.- Creamos el array de Jedis a modo de ejemplo:



```
const jedis = [  
  { nombre: "Ahsoka Tano", edad: 19, sistema: "Togruta System", planeta: "Shili", email: "ahsoka@templo.jedi" },  
  { nombre: "Obi-Wan Kenobi", edad: 38, sistema: "Stewjon System", planeta: "Stewjon", email: "obiwan@templo.jedi" },  
  { nombre: "Anakin Skywalker", edad: 22, sistema: "Tatoo System", planeta: "Tatooine", email: "anakin@templo.jedi" },  
  { nombre: "Qui-Gon Jinn", edad: 48, sistema: "Coruscant System", planeta: "Coruscant", email: "quigon@templo.jedi" },  
  { nombre: "Luke Skywalker", edad: 19, sistema: "Tatoo System", planeta: "Tatooine", email: "luke@templo.jedi" },  
  { nombre: "Yoda", edad: 900, sistema: "Desconocido", planeta: "Desconocido", email: "yoda@templo.jedi" },  
];
```

## 2.- Uso de readline



```
const readline = require("readline");  
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
```



### 3.- Creamos el menú



```
function mostrarOpciones() {
  console.log("\n=== Menú ===");
  console.log("1) Mostrar Todos los Jedis");
  console.log("2) Filtrar Jedis Mayores de 18 Años");
  console.log("3) Búsqueda por Nombre");
  console.log("4) Formatear Información");
  console.log("0) Salir");
}

function manejarOpcion(opc) {
  switch (opc) {
    case "1":
      mostrarTodos();
      mostrarMenu();
      break;
    case "2":
      filtrarMayores();
      mostrarMenu();
      break;
    case "3":
      mostrarBusqueda();
      break; // continúa dentro de mostrarBusqueda()
    case "4":
      formatearInfo();
      mostrarMenu();
      break;
    case "0":
      rl.close();
      break;
    default:
      console.log("Opción inválida.");
      mostrarMenu();
  }
}

function mostrarMenu() {
  mostrarOpciones();
  rl.question("Selecciona una opción: ", manejarOpcion);
}

mostrarMenu();
```

#### 4.- Creación de las funcione para gestionar los Jedis:



```
/** Utilidades */
const normalize = (s) => s.normalize("NFD").replace(/\p{Diacritic}/gu, "").toLowerCase().trim();

/** Funcionalidades requeridas */
function mostrarTodos() {
  console.log("\n— Lista de Jedis (forEach):");
  jedis.forEach((j) => {
    console.log(`Nombre: ${j.nombre}, Edad: ${j.edad}, Contacto: ${j.email}`);
  });

  console.log("\n— Tabla de Jedis:");
  console.table(jedis);
}

function filtrarMayores() {
  const mayores = jedis.filter((j) => j.edad > 18);
  if (mayores.length === 0) {
    console.log("\nNo hay Jedis mayores de 18.");
    return;
  }
  console.log("\n— Jedis mayores de 18 (nombres):");
  mayores.forEach((j) => console.log(j.nombre));
}

function buscarJedi(nombre) {
  if (!nombre || typeof nombre !== "string") return null;
  const key = normalize(nombre);
  return jedis.find((j) => normalize(j.nombre) === key) || null;
}

function mostrarBusqueda() {
  rl.question("Introduce el nombre exacto a buscar: ", (entrada) => {
    const res = buscarJedi(entrada);
    if (res) {
      console.log("\n— Jedi encontrado:");
      console.table([res]); // Por qué: tabla compacta para un solo resultado.
    } else {
      console.log("\nNo se encontró un Jedi con ese nombre.");
    }
    mostrarMenu();
  });
}

function formatearInfo() {
  const lineas = jedis.map(
    (j) =>
      `${j.nombre} (${j.edad} años) -- Sistema ${j.sistema} -- Planeta ${j.planeta} -- Contacto: ${j.email}`
  );
  console.log("\n— Información formateada (map):");
  console.log(lineas.join("\n"));
}
```

*Vamos a explicar el significado de:*

```
const normalize = (s) => s.normalize("NFD").replace(/p{Diacritic}/gu, "").toLowerCase().trim();
```

*Elimina diferencias de acentos y mayúsculas al comparar textos.*

- `normalize("NFD")`: descompone caracteres acentuados ("Á" → "A" + ´).
- `.replace(/p{Diacritic}/gu, "")`: quita los diacríticos combinados (se vá´), queda "A".
- `.toLowerCase()`: comparación sin distinción de mayúsculas.
- `.trim()`: elimina espacios sobrantes.

*Resultado: comparación “a prueba de acentos, mayúsculas y espacios”.*

*Ejemplos*

- "Ánakin " → "anakin"
- "Luké" → "luke"
- " Obi-Wan" → "obi-wan"

## 2.5.1 Clasificación de Jedis



### Ejemplo guiado

Hacer una aplicación de gestión de Jedis en Javascript (in HTML ni CSS) implementando funciones para clasificar y analizar a los Jedis según su edad, utilizando los métodos `sort()` y `reduce()`.

Clasificación por Edad:

Ordenar de mayor a menor edad

- Usar el método `sort()` para mostrar un listado de los Jedis desde el más longevo hasta el más joven.

Ejemplo de salida:

1. Yoda (900 años)
2. Obi-Wan Kenobi (57 años)
3. Mace Windu (53 años)

Ordenar de menor a mayor edad

- Mostrar el listado inverso (del más joven al más viejo).

Top 3 de Jedis más longevos

- Extraer y mostrar solo los 3 primeros del ranking (mayores edades).

Estadísticas de Edad:

Calcular la edad promedio

- Usar `reduce()` para obtener la media de edad de todos los Jedis.
- Mostrar el resultado redondeado a 2 decimales (ej: 209.83 años).

Suma total de años de experiencia

- Sumar las edades de todos los Jedis y mostrar el resultado.

Identificar al Jedi más joven y más viejo

- Devolver el objeto completo del Jedi con la edad mínima y máxima.

Ejemplo:

Jedi más joven: { nombre: "Ahsoka Tano", edad: 17, ... }

Jedi más viejo: { nombre: "Yoda", edad: 900, ... }

Requisitos Técnicos:

Crea un menú para realizar las funcionalidades.

Usar `sort()` con una función de comparación para ordenar correctamente.

Aplicar `reduce()` para cálculos acumulativos (suma, promedio).

No modificar el array original de Jedis (usar copias con [...jedis]).

## 1.- Creamos el array de Jedis a modo de ejemplo:



```
const jedis = [
  { nombre: "Yoda", edad: 900, sistema: "Desconocido", planeta: "Desconocido", email: "yoda@templo.jedi" },
  { nombre: "Obi-Wan Kenobi", edad: 57, sistema: "Stewjon", planeta: "Stewjon", email: "obiwan@jedi.org" },
  { nombre: "Mace Windu", edad: 53, sistema: "Haruun Kal", planeta: "Haruun Kal", email: "mace@jedi.org" },
  { nombre: "Qui-Gon Jinn", edad: 60, sistema: "Coruscant", planeta: "Coruscant", email: "quigon@jedi.org" },
  { nombre: "Anakin Skywalker", edad: 45, sistema: "Tatooine", planeta: "Tatooine", email: "anakin@jedi.org" },
  { nombre: "Ahsoka Tano", edad: 17, sistema: "Shili", planeta: "Shili", email: "ahsoka@jedi.org" },
  { nombre: "Plo Koon", edad: 70, sistema: "Dorin", planeta: "Dorin", email: "plokoon@jedi.org" },
  { nombre: "Kit Fisto", edad: 50, sistema: "Glee Anselm", planeta: "Glee Anselm", email: "kit@jedi.org" },
];
```

## 2.- Uso de readline



```
const readline = require("readline");
const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
```

## 3.- Utilidades



```
const ordenarPorEdadDesc = (arr) => [...arr].sort((a, b) => b.edad - a.edad);
const ordenarPorEdadAsc = (arr) => [...arr].sort((a, b) => a.edad - b.edad);
const top3Longev = (arr) => ordenarPorEdadDesc(arr).slice(0, 3);

const sumaEdades = (arr) => arr.reduce((acc, j) => acc + j.edad, 0);
const edadMedia = (arr) => sumaEdades(arr) / arr.length;

const extremosEdad = (arr) =>
  arr.reduce(
    (acc, j) => {
      if (!acc.min || j.edad < acc.min.edad) acc.min = j;
      if (!acc.max || j.edad > acc.max.edad) acc.max = j;
      return acc;
    },
    { min: null, max: null }
  );
```

## 4.- Presentación de información



```
const imprimirListado = (arr) => arr.forEach((j, i) => console.log(`${i + 1}. ${j.nombre} (${j.edad} años)`));
const imprimirObjeto = (titulo, obj) => { console.log(titulo); console.log(obj); };
```

## 5.- Menú



```
async function menu() {
  let salir = false;
  while (!salir) {
    console.log("\n=== Gestión de Jedis por Edad ===");
    console.log("1) Listar de mayor a menor edad");
    console.log("2) Listar de menor a mayor edad");
    console.log("3) Top 3 más longevos");
    console.log("4) Calcular edad promedio");
    console.log("5) Suma total de edades");
    console.log("6) Jedi más joven y más viejo");
    console.log("0) Salir");

    const op = (await rl.question("Elige una opción: ")).trim();

    switch (op) {
      case "1":
        console.log("\n>> De mayor a menor:");
        imprimirListado(ordenarPorEdadDesc(jedis));
        break;
      case "2":
        console.log("\n>> De menor a mayor:");
        imprimirListado(ordenarPorEdadAsc(jedis));
        break;
      case "3":
        console.log("\n>> Top 3 más longevos:");
        imprimirListado(top3Longev(jedis));
        break;
      case "4":
        console.log("\n>> Edad promedio: ${edadMedia(jedis).toFixed(2)} años`");
        break;
      case "5":
        console.log("\n>> Suma total de edades: ${sumaEdades(jedis)} años`");
        break;
      case "6": {
        const { min, max } = extremosEdad(jedis);
        console.log("\n>> Extremos de edad:");
        imprimirObjeto("Jedi más joven:", min);
        imprimirObjeto("Jedi más viejo:", max);
        break;
      }
      case "0":
        salir = true;
        break;
      default:
        console.log("Opción no válida.");
    }
  }
  rl.close();
}

menu();
```

## 2.6. Ejemplo propuesto: Gestión de tienda



Vas a desarrollar un programa en JavaScript puro (sin librerías ni frameworks externos) para gestionar los productos de una tienda.

- 1.- Define los productos como objetos con las siguientes propiedades:
  - id (número entero único)
  - producto (nombre del artículo)
  - precio (número decimal)
  - descripción (texto breve)
- 2.- Declara un array de productos, que servirá como inventario de la tienda.
- 3.- Implementa las siguientes operaciones mediante funciones:
- 4.- Agregar producto: recibe los datos y lo añade al array.
- 5.- Borrar producto: elimina un producto del array a partir de su id.
- 6.- Ordenar productos: permite ordenar el array por precio o por producto (orden alfabético).
- 7.- Filtrar productos: devuelve todos los productos cuyo precio sea menor o igual a un valor dado.
- 8.- Realiza las pruebas de cada función llamándolas desde el propio programa y mostrando los resultados con `console.log`.

# ANEXO II. Variables, tipos de datos, asignaciones, operaciones, comentarios, sentencias, decisiones y bucles

## AII.1. Variables

Las variables se pueden declarar de las siguientes maneras:

Modo	Descripción	Uso	Ejemplo
<b>Automáticamente</b>	Se le asigna un valor		x=5;
<b>var</b>	Usado en código de 1995 hasta 2015 y navegadores antiguos.	Solo si debe soportar navegadores antiguos.	var x = 5;
<b>let</b>	Se agregaron a JavaScript en 2015	Variables que pueden tomar diferentes valores en el tiempo. Si no puede usar const.	let x =5;
<b>const</b>		Si no cambia el valor o tipo	const x=5;

Uso concreto de **let** (se introdujo en ES6 (2015) ).

Uso	Ejemplo
Tienen alcance de bloque	<pre>{   let x = 2; } // x can NOT be used here</pre>
Deben declararse antes de su uso	<pre>carName = "Saab"; let carName = "Volvo";</pre>
No se pueden redeclarar en el mismo ámbito	<pre>let x = "John Doe";  let x = 0;</pre>

## AII.2. Tipos de datos

Los tipos soportados por Javascript son:

Tipo	Ejemplo	Almacenamiento
String	let color = "Yellow";	
Number	let length = 16; let weight = 7.5; let z = 123e-5;	64 bits punto flotante.
Bigint	const bigNumber = BigInt("12345678901234567890123456789")	Números integer más allá del límite $2^{53} - 1$ .



	0");	
Boolean	let x = true;	
Undefined	let x; if (x === undefined) { text = "x is undefined"; } else { text = "x is defined"; }	Indica que una variable no tiene asignado un valor.
Null	Var persona = null; persona = {nombre="José", apellido="Miró"}	Se aplica a objetos para indicar ue es nada o que no existe.
Object	const person = {firstName:"John", lastName:"Doe"}; const cars = ["Saab", "Volvo", "BMW"]; const date = new Date("2022-03-25");	

## AII.2.a Conversiones de datos

### Conversión a String

Ejemplo	Conversión a String	Resultado
let numero = 123;	let strNumero = String(numero);	"123"
let booleano = true;	et strBooleano = String(booleano);	"true"
let objeto = { nombre: "Juan" };	let strObjeto = String(objeto)	"[object Object]"

Podemos usar también:

```
let strNumero2 = numero.toString(); // "123"
```

```
let strBooleano2 = booleano.toString(); // "true"
```

Pero .toString() no funciona con null y undefined.

### Conversión a Number, parseInt, parseFloat

Ejemplo	Conversión a Number	Resultado
let strNumero = "123";	let num1 = Number(strNumero);	123
let strNoNumero = "123abc"	let num2 = Number(strNoNumero);	NaN*
let boolean= true;	let num3 = Number(boolean)	1**
	let entero = parseInt("123.45");	123
	let decimal = parseFloat("123.45");	123.45

\* NaN significa Not a Number  
 \*\* true → 1, false → 0

## Conversión a Boolean

Ejemplo	Conversión a Boolean	Resultado
let strVacia = "";	let bool1 = Boolean(strVacia);	false
let strTexto = "Hola";	let bool1 = Boolean(strVacia);	true
let numeroCero = 0;	let bool1 = Boolean(strVacia);	false
let numeroPositivo = 123;	let bool1 = Boolean(strVacia);	true

## Conversión a BigInt

Ejemplo	Conversión a BigInt	Resultado
let strNumero = "12345678901234567890";	let bigInt1 = BigInt(strNumero);	12345678901234567890n

## AII.3. Asignaciones

Son las igualdades y algunas tienen una forma peculiar de trabajar.

Normalmente cuando a una misma variable queremos aumentar su valor haríamos lo siguiente:

X = X + 2;

En Javascript, al igual que otros lenguajes, podemos escribir esto mismo de la forma:

X += 2;

Esta operación se lee como “incremento y luego asigno a X”.

Y no es lo mismo que:

X =+ 2;

Que en este caso estamos asignando un número positivo.

## AII.4. Operaciones

### AII.4.a. Aritméticas

Operador	Operación	Operador
Igual	x = y	=
suma	x = x + y	+=
resta	x = x - y	-=
multiplicación	x = x * y	*=
división	x = x / y	/=

módulo	$x = x \% y$	$\% =$
esponencial	$x = x ** y$	$** =$

## AII.4.b. Desplazamiento de bits

Operador	Operador	Example	Equivale a
Desplaza a la izquierda con signo	$\ll =$	$x \ll = y$	$x = x \ll y$
Desplaza a la derecha con signo	$\gg =$	$x \gg = y$	$x = x \gg y$
Desplazamiento sin signo	$\ggg =$	$x \ggg = y$	$x = x \ggg y$

Un uso que se le suele dar es la obtención del peso de los colores primarios de un color dado. ¡Pruébalo tu mismo!



```
// Extraer componentes de color ARGB de un valor de 32 bits
let color = 0xFFA07A00; // Color naranja semitransparente

let alpha = color >>> 24; // 0xFF (255)
let red = (color >>> 16) & 0xFF; // 0xA0 (160)
let green = (color >>> 8) & 0xFF; // 0x7A (122)
let blue = color & 0xFF; // 0x00 (0)
```

## AII.4.c. Operaciones bit a bit

Son las realizadas entre números binarios pero bit a bit.

Operator	Example	Same As
$\& =$	$x \& = y$	$x = x \& y$
$\wedge =$	$x \wedge = y$	$x = x \wedge y$
$  =$	$x   = y$	$x = x   y$

## AII.5.d. Operadores lógicos

Son las realizadas entre números binarios.

Operator	Example	Same As
$\&\& =$	$x \&\& = y$	$x = x \&\& (x = y)$
$\ \& =$	$x \ \& = y$	$x = x \ \& (x = y)$
$?? =$	$x ?? = y$	$x = x ?? (x = y)$

### AII.5.e. Type of

En javascript Typeof es un tipo, no una función como en otros lenguajes de programación.  
Nos permite conocer el tipo de la variable.

### AII.5.f. Operadores Unarios

Son los operadores de incremento y decremento en una unidad.

Operador	Ejemplo
++	let x = 5; x++; let z = x;
--	let x = 5; x--; let z = x;

¿Crees que será los mismo?

Let x = 5;

let y = ++x;

qué

let y = x++;

Vamos a ver este efecto con el siguiente código:



```
console.log("Tened en cuenta el orden en que se aplica el operador");
var edadCarmen = 18;
console.log("Aplico el opeorador DESPUES. La edad de Carmen: " + edadCarmen++);
console.log("Vuelvo a imprimir la edad de Carmen: " + edadCarmen);
console.log("-----");
edadCarmen = 18;
console.log("Aplico el opeorador ANTES. La edad de Carmen: " + ++edadCarmen);
console.log("Vuelvo a imprimir la edad de Carmen: " + edadCarmen);
```

Estudia el código y observa la diferencia entre edadCarmen++ y ++edadCarmen.

### AII.5. Comentarios al código

Los comentarios de una sola línea comienzan con //.

Cualquier texto entre //y el final de la línea será ignorado por JavaScript (no se ejecutará).

**Ejemplo:** // Esto es un ejemplo

Los comentarios de varias líneas comienzan con `/*` y terminan con `*/`.

Cualquier texto entre `/*` y `*/` será ignorado por JavaScript.

**Ejemplo:**

```
/*  
Esto es un ejemplo de bloque comentado  
*/
```

## AII.6. Sentencias

Las sentencias son instrucciones individuales que realizan acciones. Son los bloques básicos de construcción de un programa JavaScript.

Las declaraciones de JavaScript se componen de: Valores, operadores, expresiones, palabras clave y comentarios.

Las sentencias se ejecutan, una por una, en el mismo orden en que están escritas.

Cada línea termina en punta y coma (;), no es obligatorio pero sí recomendable.

Si usamos “;” podríamos escribir: `a = 5; b = 6; c = a + b;` en una sola línea.

Los espacios son ignorados por javascript, así que: **`let person = "Hege";`** y **`let person="Hege";`** es lo mismo.

Una buena práctica es colocar espacios alrededor de los operadores ( `= + - * /` ):

```
let x = y + z;
```

Las declaraciones de JavaScript se pueden agrupar en bloques de código, dentro de llaves `{...}`. Usado en la declaración de funciones, por ejemplo.

```
function miFuncion() {  
    // Código de la función  
};
```

## AII.7. Decisiones

Las decisiones permiten controlar el flujo de ejecución del programa basado en condiciones.

### AII.7.a. if....else

Ejecuta un bloque de código si una condición es verdadera.

Veamos algunos ejemplos para que los pruebes.



```
var nombre = 'Pablo';
var estadoCivil = 'soltero';
var estaCasado = true;

if (estaCasado){
  //si es verdadera la condición
  console.log(nombre + ' esta casado');
}else {
  //si es falsa la condición
  console.log(nombre + ' esta soltero');
}
```

Si se requieren más condiciones se usará **else if**.



```
var nombre = 'Pablo';
var edad = 8;

// edad < 12 es un niño.
// edad > 11, es < 18, es un adolescente.
// edad > 17, es < 60, es un adulto.
// edad > 60, es un anciano.

if (edad < 12){
  console.log(nombre + ' es un niño.');
```

```
}else if ((edad > 11) && (edad < 18)){
  console.log(nombre + ' es un adolescente.');
```

```
}else if ((edad > 17) && (edad < 60)){
  console.log(nombre + ' es un adulto.');
```

```
}else{
  console.log(nombre + ' es un anciano.');
```

```
}
```

## Evaluación de variables lógicas

Nos referimos a cómo se evalúa una variable. Veamos un ejemplo:



```
var edad;  
//edad = 10;  
  
if(edad){  
  console.log('Variable esta definida');  
}else{  
  console.log('Variable no definida');  
}  
  
//operadores de igualda  
if(edad === '10'){  
  console.log('Variable con coersión');  
}else{  
  console.log('Variable sin coersión');  
}
```

Aquí he definido la variable pero no le he dado un valor y resuta:

Variable no definida	<a href="#">app.js:118</a>
Variable sin coersión	<a href="#">app.js:125</a>

Ahora si yo doy el valor 10 a edad se devolverá:

Variable esta definida	<a href="#">app.js:116</a>
Variable sin coersión	<a href="#">app.js:125</a>

Resulta que el segundo if estoy comprando un número con una cadena, por ello me dice que no hay coersión. Escribamos `if(edad === 10)` a ver qué ocurre. Resulta:

Variable esta definida	<a href="#">app.js:116</a>
Variable con coersión	<a href="#">app.js:123</a>

Ahora sí que tiene un valor y la comparación del segundo if corresponde los tipos.

### AII.7.b. Operador ternario

Otra forma de escribir el if, de forma más concisa es a través del operador ternario, que tiene una estructura como:

(condición a evaluar) ? (acción a realizar si es Verdadero) : (acción a realizar si es Falso) ;

Imagina que yo tuviese:

```
if (edad >= 18){
    console.log(nombre + ' es mayor de edad.');
```

```
}else {
    console.log(nombre + ' es un adolescente.');
```

Usando el operador ternario quedaría:

```
edad >= 18 ? console.log(nombre + ' es mayor de edad') :
    console.log(nombre + ' es un adolescente');
```

## AII.7.c. Switch

Cuando sabemos que valores puede tomar una variable y por cada uno de ellos tenemos que realizar acciones usaremos la estructura switch.

Veamos un ejemplo. Capturemos el día de la semana e imprimirlo.



El método **getDay()** devuelve el día de la semana como un número entre 0 y 6.

(Domingo=0, Lunes=1, Martes=2..)



```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
    day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```



## AII.8. Bucles

### AII.8.a. for

Se ejecuta un código tantas veces como se cumpla una condición.

Sigue la sintaxis:

```
for (expression 1; expression 2; expression 3) {  
  // code block to be executed  
}
```

La expresión 1 se ejecuta (una vez) antes de la ejecución del bloque de código.

La expresión 2 define la condición para ejecutar el bloque de código.

La expresión 3 se ejecuta (cada vez) después de que se haya ejecutado el bloque de código.

Por ejemplo:



```
for (var i = 10; i >= 1; i--){  
  console.log(i);  
}
```

¿Qué se imprime por consola?

### Otros usos de la expresión 1

Pero, la expresión 1 es opcional.

Puede omitir la expresión 1 cuando sus valores se establecen antes de que comience el bucle. En el siguiente ejemplo la variable “i” se inicializa a 2 antes del for, al igual que len.



```
let i = 2;  
let len = cars.length;  
let text = "";  
for (; i < len; i++) {  
  text += cars[i] + "<br>";  
}
```

Podemos iniciar muchos valores en la expresión 1 (separados por coma):



```
for (i = 0, len = cars.length, text = ""; i < len; i++) {  
  text += cars[i] + "<br>";  
}
```

### Otros usos de la expresión 3

La expresión 3 también se puede omitir (como cuando incrementas tus valores dentro del bucle). Véase el siguiente ejemplo donde “i” es incrementado dentro del bucle.



```
let i = 0;  
let len = cars.length;  
let text = "";  
for (; i < len; ) {  
  text += cars[i] + "<br>";  
  i++;  
}
```

### Alcance de var y let

Veamos el ejemplo de var:



```
var i = 5;  
  
for (var i = 0; i < 10; i++) {  
  console.log("Dentro del for i vale " + i);  
}  
console.log("Fuera del for i vale " + i);
```

Veamos el ejemplo de let:



```
let i = 5;  
  
for (let i = 0; i < 10; i++) {  
  console.log("Dentro del for i vale " + i);  
}  
console.log("Fuera del for i vale " + i);
```

¿Qué observas? ¿Qué conclusiones obtienes?

### AII.8.b. for/in

Declaración que recorre las propiedades de un objeto.

La sintaxis es:

```
for (key in object) {  
    // code block to be executed  
}
```

Veamos un ejemplo donde recorreremos el objeto “persona”.



```
const person = {fname:"David", lname:"Aria", age:25};  
  
let text = "";  
for (let x in person) {  
    text += person[x];  
}
```

### AII.8.c. for/of

Recorre los valores de un **objeto iterable**, como matrices, cadenas, mapas, listas de nodos y más.

Veamos el siguiente ejemplo:



```
const cars = ["BMW", "Volvo", "Mini"];  
  
let text = "";  
for (let x of cars) {  
    text += x;  
}  
  
console.log(text);
```

¿Qué imprime el ejemplo?

### AII.8.d. while

La sentencia while ejecuta un código mientras se cumpla una condición. Se tiene que inicializar la variable a chequear antes de iniciar el while.



```
var i = 10;
while(i >= 1){
  console.log(i);
  i--;
}
```

### AII.8.e. do...while

La sentencia do...while es similar al while, solo que se ejecuta una vez el código que hay dentro.



```
var i = 3;
do{
  console.log(i);
  i++;
}while(i <= 10)
```

## ANEXO III. Arrays, objetos y contexto this

### AIII.1. Arrays

Un array es una colección ordenada de elementos. Puede contener cualquier tipo de dato, incluidos otros arrays u objetos.

La Declaración y acceso es la siguiente:



```
let frutas = ["manzana", "plátano", "pera"];  
console.log(frutas[0]); // "manzana";
```

#### AIII.1.a. Métodos comunes

- **push()**: Añade un elemento al final.
- **pop()**: Elimina el último elemento.
- **shift()**, **unshift()**: Eliminan/añaden elementos al principio.
- **length**: Devuelve la longitud del array.
- **forEach()**: Recorre el array y ejecuta una función por cada elemento.
- **filter()**: permite filtrado según condición.
- **sort()**: ordenar array según condición.
- **reduce()**: aplica una regla al array para “reducirlo” a un número.
- **map()**: crea un nuevo array aplicando una función a cada elemento. No muta el original.

Ejemplo:



```
let numeros = [1, 2, 3];  
numeros.push(4); // [1, 2, 3, 4]  
numeros.forEach(num => console.log(num));
```

Crea un archivo con el código anterior y ejecútalo. ¿Qué obtenemos por consola?

#### AIII.1.b. Modificación de los elementos

Por ejemplo si en `vegetales = new Array('Tomate', 'Lechuga', 'Zanahoria');` quiero cambiar Lechuga por Apio haría:

```
vegetales[1] = "Apio";
```

### AIII.1.c. Encontrar un elemento, indexOf() y findIndex()

La forma de localizar el índice de un elemento es la siguiente:

Si tengo el arreglos

```
const ceviche = ['percado', 'marisco', 'cebolla'];
```

El índice de marisco sería:

```
ceviche.indexOf('marisco');
```

Devolverá: 1.

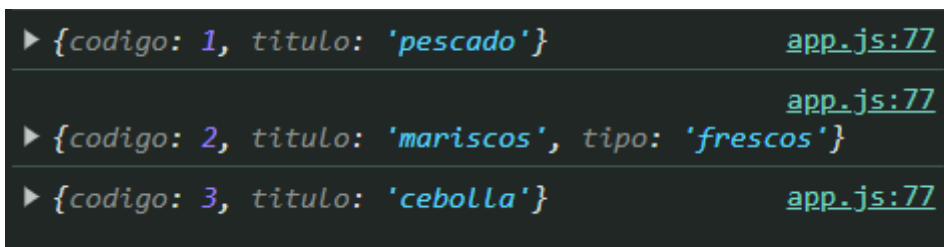
En el caso de tener un arreglo de objetos, la cosa cambia. Por ejemplo si tenemos:

```
const ceviche = [  
  {  
    codigo: 1,  
    titulo: 'pescado'  
  },  
  {  
    codigo: 2,  
    titulo: 'mariscos',  
    tipo: 'frescos'  
  },  
  {  
    codigo: 3,  
    titulo: 'cebolla'  
  }  
];
```

La forma de buscar un objeto es con findIndex(). Se usaría como:

```
const index = ceviche.findIndex(function(ing, index){  
  console.log(ing);  
});
```

Devuelve:



```
▶ {codigo: 1, titulo: 'pescado'} app.js:77  
▶ {codigo: 2, titulo: 'mariscos', tipo: 'frescos'} app.js:77  
▶ {codigo: 3, titulo: 'cebolla'} app.js:77
```

En el caso de buscar el ingrediente cebolla haría algo como:

```
const index = ceviche.findIndex(function(ing, index){
    return ing.titulo === 'cebolla';
});
console.log(index);
```

Que devolvería el índice 2.

Si fuese uno que no existiera devolvería -1.

### AIII.1.d. Filtrar arreglo según una condición. Filter()

Imaginemos que tengo el arreglo de países;

```
const paises = ['Colombia', 'Ecuador', 'Perú', 'España', 'Mexico'];
```

Y quiero filtrar los que contenga la letra “o”.

```
console.log(paises.filter(pais => pais.includes('o')));
```

Devuelve:

```
► (3) ['Colombia', 'Ecuador', 'Mexico'] app.js:74
```

Si quisiera los que comiencen por “Es”:

```
console.log(paises.filter(pais => pais.includes('Es')));
```

Devolvería:

```
► ['España'] app.js:74
```

fuese un

arreglo con objetos tendríamos que hacer algo como:

Puedo crear una función para filtrar. Y en este caso filtro por el ingrediente que contenga las letras “mar”:

```
const filtrarIngredientes = ceviche.filter(function(ing, index){
    const ingredientes = ing.titulo.includes('mar');
    return ingredientes;
});
console.log(filtrarIngredientes);
```

Esto devuelve:

```
▼ [{...}] ⓘ
  ► 0: {codigo: 2, titulo: 'mariscos', tipo: 'frescos'}
    length: 1
  ► [[Prototype]]: Array(0)
```

### AIII.1.e. Ordenación. sort()

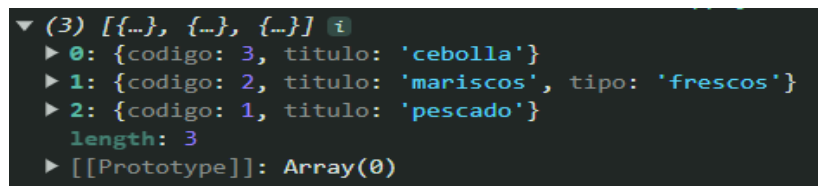
Para ordenar un arreglo de objeto y siguiendo con el ejemplo de ceviche:

```
const ceviche = [  
  {  
    codigo: 1,  
    titulo: 'pescado'  
  },  
  {  
    codigo: 2,  
    titulo: 'mariscos',  
    tipo: 'frescos'  
  },  
  {  
    codigo: 3,  
    titulo: 'cebolla'  
  }  
];
```

Para ordenarlo sigo lo que indica la teoría y es aplicar una función:

```
ceviche.sort(function(a,b){  
  if(a.titulo > b.titulo){  
    return 1;  
  }  
  if(a.titulo < b.titulo){  
    return -1  
  }  
  return 0;  
});  
console.log(ceviche);
```

Que devuelve:



```
▼ (3) [{...}, {...}, {...}] ⓘ  
  ▶ 0: {codigo: 3, titulo: 'cebolla'}  
  ▶ 1: {codigo: 2, titulo: 'mariscos', tipo: 'frescos'}  
  ▶ 2: {codigo: 1, titulo: 'pescado'}  
    length: 3  
  ▶ [[Prototype]]: Array(0)
```



### AIII.1.f. Acumulación. `reduce()`

Reduce acumula valores de un array para producir un único resultado. Dicho de otra forma, reduce ejecuta una función acumuladora elemento a elemento para condensar el array en un único valor.

Tiene la sintaxis: `miArray.reduce((callback(acumulador, valor, indice, array), valorInicial?))` ;

Si se omite `valorInicial`, el primer elemento será usado como acumulador y se empieza a contar en el índice 1. ¡Claro! Un array vacío sin `valorInicial` lanzará un `TypeError`.

Veamos casos típicos:

#### 1) Sumar y multiplicar

```
const nums = [1, 2, 3, 4];
const suma = nums.reduce((acc, n) => acc + n, 0);
const producto = nums.reduce((acc, n) => acc * n, 1);
console.log('suma:', suma);           // 10
console.log('producto:', producto);    // 24
```

Nota en `nums.reduce((acc, n) => acc + n, 0)`; veamos qué es cada cosa comparado con `miArray.reduce((callback(acumulador, valor, indice, array), valorInicial?))`:

- `.reduce(...)`: método que condensa el array en un valor.
- `(acc, n) => acc + n`: función acumuladora.
- **acc: acumulador**. Lleva el total parcial.
- **n: elemento actual** del array.
- **retorno implícito: acc + n** (sin llaves ni return).
- En `(acc, n, i, arr)`, no he usado *i* (índice) ni *array*
- **, 0**: valor inicial del acumulador.

La traza es:

```
Flujo con nums = [1,2,3,4]:
acc=0, n=1 ⇒ 1
acc=1, n=2 ⇒ 3
acc=3, n=3 ⇒ 6
acc=6, n=4 ⇒ 10
reduce devuelve 10; se asigna a suma.
```

#### 2) Máximo y mínimo

```
const max = nums.reduce((acc, n) => (n > acc ? n : acc), -Infinity);
const min = nums.reduce((acc, n) => (n < acc ? n : acc), Infinity);
console.log('max:', max); // 4
console.log('min:', min); // 1
```

Nota: `reduce((acc, n) => (n > acc ? n : acc), -Infinity);`

`n > acc ? n : acc` es el operador ternario, igual a un `if`: SI `n > acc` DEVUELVO `n`, SI NO `acc`.

`-Infinity` es el valor inicial del acumulador. Cualquier valor número infinito negativo.

### 3) Aplanar un nivel

```
const matriz = [[1, 2], [3, 4], [5]];
const plano = matriz.reduce((acc, arr) => acc.concat(arr), []);
console.log('plano:', plano); // [1,2,3,4,5]
```

### 4) Contar ocurrencias

```
const letras = ['a', 'b', 'a', 'c', 'b', 'a'];
const conteo = letras.reduce((acc, ch) => {
  acc[ch] = (acc[ch] || 0) + 1;
  return acc;
}, {});
console.log('conteo:', conteo); // { a: 3, b: 2, c: 1 }
```

### 5) Agrupar por clave (groupBy)

```
function groupBy(items, keyFn) {
  // Por qué: devuelve estructura útil para consultas.
  return items.reduce((acc, item) => {
    const k = keyFn(item);
    (acc[k] ??= []).push(item);
    return acc;
  }, {});
}
const users = [
  { id: 1, role: 'admin' },
  { id: 2, role: 'user' },
  { id: 3, role: 'admin' },
];
console.log('groupBy role:', groupBy(users, u => u.role));
// { admin: [{...},{...}], user: [{...}] }
```

### 6) Eliminar duplicados preservando orden

```
const conDup = [3, 1, 2, 3, 2, 4, 1];
const sinDup = conDup.reduce((acc, n) => {
  if (!acc.set.has(n)) { acc.set.add(n); acc.out.push(n); }
  return acc;
}, { set: new Set(), out: [] });
```

```
    return acc;
  }, { set: new Set(), out: [] }).out;
console.log('sinDup:', sinDup); // [3,1,2,4]
```

### **AIII.1.g. Acciones sobre array sin mutarlo. Map()**

Se usa para transformar cada elemento de un array en otro valor y obtener un nuevo array con la misma longitud. No muta el original. Es la herramienta estándar para cambios “uno a uno”.

La sintaxis es: `array.map((valor, indice, array) => nuevoValor, thisArg?)`

#### **Realizar cálculos**

```
const newArray = [1, 2, 3].map(n => n * 2);
```

#### **Obtener propiedades:**

```
const usuarios = [{id:1,n:"Ana"}, {id:2,n:"Luis"}];
const newArray = usuarios.map(u => u.id);
```

#### **Obtener el índice de un array**

```
const newArray = ["a","b","c"].map((v,i) => `${i+1}. ${v}`);
// ["1. a","2. b","3. C"]
```

#### **Modo asíncrono (uso de Promise.all):**

```
const urls = ["a.json","b.json"];
const datos = await Promise.all(urls.map(u => fetch(u).then(r => r.json())));
```

## **AIII.2. Objetos**

Los objetos permiten agrupar datos mediante pares clave-valor.

### **AIII.2.a. Declaración de objetos**

La forma de declaración es la siguiente:

```
objeto = {elemento1: 'valor1',
          elemento2: 'valor2'}
```

Por ejemplo:



```
let persona = {  
  nombre: "Ana",  
  edad: 25,  
  saludar: function() {  
    console.log("Hola, soy " + this.nombre);  
  }  
};
```

### AIII.2.b. Creación de objeto

Puedo crear el objeto con la ayuda de la **sintaxis Object** de la siguiente manera:  
Si hiciéramos el ejemplo de persona:



```
let persona = new Object();  
  persona.nombre = 'David';  
  persona.edad = 20;  
  persona.ciudad = 'Puerto de la Cruz';  
  persona.gustos = ['Tenis', 'Cine', 'Camping'];  
  persona.escasado = true;
```

### AIII.2.c. Acceso y modificación

Se realiza a través de *.propiedad*



```
console.log(persona.nombre); // "Ana"  
persona.edad = 26;  
persona.saludar();
```

## AIII.3. El contexto this

this es una palabra clave que hace referencia al contexto en el que se ejecuta una función. Su valor depende de cómo y dónde se llama a la función.

## Ejemplo:



```
const perro = {  
  nombre: "Rex",  
  ladrar: function() {  
    console.log(`¡Guau! Soy ${this.nombre}`);  
  }  
};  
  
perro.ladrar(); // "¡Guau! Soy Rex" (this = objeto `perro`)
```

En este caso this hace referencia al objeto perro.

this en eventos del DOM apunta al elemento que lo disparó, por ejemplo:



```
<button id="miBoton">Click aquí</button>  
  
<script>  
  document.getElementById("miBoton").addEventListener("click", function() {  
    console.log(this); // El botón (`<button id="miBoton">`)  
  });  
</script>
```

En este caso this es el objeto botón.