# Unidad 2

## Manejo de la sintaxis del lenguaje



Profesor: Santiago Fernández

# Contenido

## 2. CONTENTS

## OBJETIVOS

- Comprender y aplicar los conceptos fundamentales relacionados con las variables, incluyendo su declaración, ámbito de utilización y buenas prácticas de nombramiento.
- Identificar y utilizar correctamente los diferentes tipos de datos disponibles en JavaScript, como números, cadenas de texto, valores booleanos, objetos y símbolos.
- Dominar las técnicas de conversión entre tipos de datos en JavaScript, incluyendo la conversión de cadenas a números y viceversa.
- Utilizar literales adecuadamente en el código, tanto para representar valores numéricos y de texto, como para crear objetos y símbolos.
- Dominar el proceso de asignación de valores a variables, comprendiendo la diferencia entre asignaciones simples y asignaciones con operadores.
- Aplicar correctamente los operadores disponibles en JavaScript para realizar operaciones aritméticas, comparaciones y lógicas.
- Comprender y construir expresiones en JavaScript, utilizando variables, literales y operadores para realizar cálculos y evaluaciones.
- Utilizar comentarios de manera efectiva para documentar el código, explicar su funcionamiento y hacer anotaciones relevantes para futuras referencias.
- Aplicar las diferentes sentencias disponibles en JavaScript, tanto de control de flujo (como if, else, switch) como de repetición (como for, while, do...while) para lograr la lógica deseada en los programas.
- Dominar el uso de bloques de código para agrupar instrucciones relacionadas y mejorar la legibilidad y mantenibilidad del código.

## 2. MANEJO DE LA SINTAXIS DEL LENGUAJE

En la Unidad 1, se introdujeron los conceptos fundamentales de JavaScript, sus capacidades y limitaciones, así como su integración con tecnologías web como HTML y CSS. En esta unidad, profundizarás en la sintaxis de JavaScript, la cual es esencial para aplicar esos conceptos y desarrollar aplicaciones web dinámicas y eficientes.

Aprender la sintaxis de JavaScript es como aprender el abecedario de un nuevo idioma. Es el primer paso para poder "hablar" con las computadoras y crear programas que hagan exactamente lo que queremos. JavaScript, aunque comparte similitudes con lenguajes como Java o C++, tiene su propia estructura y reglas que lo hacen perfecto para la web.

**¿Sabías que…?**

- Cualquier código en JavaScript puede "conversar" con la consola del navegador. Esta es una herramienta increíblemente poderosa para probar y depurar tu código.
- Para hablar con la consola, usa el comando console.log. Aquí puedes poner mensajes que te ayudarán a entender qué está pasando en tu programa.
- Por ejemplo, si escribes console.log("¡Hola, mundo!");, verás ese mensaje en la consola de tu navegador.
- Esta función es esencial para los desarrolladores web, ya que les permite ver los valores de las variables en tiempo real y encontrar errores más rápidamente.
- Todos los navegadores modernos vienen equipados con una consola integrada, lista para ayudarte en tu aventura de programación.

### 2.1. REGLAS BÁSICAS DE LA SINTAXIS DE JAVASCRIPT

**Punto y Coma:** En JavaScript, el punto y coma indica el final de una instrucción. Aunque el motor de JavaScript puede inferir su presencia, es una buena práctica incluirlo para evitar errores sutiles.

```
let x = 10;
const nombre = 'Ana';
```

**Nombres de Variables**: Los nombres deben ser descriptivos y seguir la convención camelCase, donde cada nueva palabra comienza con una letra mayúscula, excepto la primera.

```
let edadUsuario = 25;
const PI = 3.14;
```

**Declaración de Variables**: let y const son las palabras clave para declarar variables. let permite reasignar valores, mientras que const es para valores que no cambiarán.

```
let saldo = 100;
const tasaInteres = 0.05;
```

**Funciones**: Las funciones se pueden declarar de manera tradicional o como funciones flecha, una sintaxis más corta y sin su propio contexto de this.

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

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

**Cadenas de Texto**: JavaScript acepta comillas simples, dobles y *template literals*. Estos últimos son útiles para insertar expresiones dentro de una cadena.

```
const mensaje = 'Bienvenido al mundo de JavaScript';
const plantilla = `Hola, ${nombre}, ¿cómo estás?`;
```

**Comparaciones**: Para comparaciones que incluyan el tipo de dato, usa === y !==. Evita == y !=, que realizan una conversión de tipo automática.

```
const edad = 18;
if (edad >= 18) {
    console.log('Eres mayor de edad');
}
```

**Bloques de Código**: Usa llaves {} para definir bloques de código y mantén una indentación consistente para mejorar la legibilidad.

```
function calcularArea(base, altura) {
    const area = base * altura;
    return area;
}
```

**Comentarios**: Los comentarios son esenciales para explicar el código. Usa // para una sola línea o /* */ para múltiples líneas.

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

/*
Este es un comentario
de múltiples líneas
*/
```

**Estructuras de Control**: if, else, switch, for, while y do...while son estructuras que controlan el flujo del programa.

```
let temperatura = 25;
if (temperatura > 30) {
    console.log('Hace calor');
} else {
    console.log('Hace fresco');
}
```

**Arrays**: Los arrays se declaran con corchetes y permiten almacenar listas ordenadas de valores.

```
const colores = ['rojo', 'verde', 'azul'];
console.log(colores[1]); // Imprime 'verde'
```

**Objetos**: Los objetos se declaran con llaves y representan colecciones de pares clave-valor.

```
const persona = {
    nombre: 'Carlos',
    edad: 30,
    profesion: 'Desarrollador'
};
console.log(persona.nombre); // Imprime 'Carlos'
```

**Truthy y Falsy**: Entiende qué valores se consideran verdaderos o falsos en contextos booleanos.

```
const valor = 0;
if (valor) {
    console.log('Valor verdadero');
} else {
    console.log('Valor falso');
}
```

**Asincronía**: JavaScript maneja operaciones asincrónicas mediante callbacks, promises y async/await.

```
function obtenerDatos() {
    return new Promise((resolve, reject) => {
        // Simulación de operación asincrónica
        setTimeout(() => {
            resolve('Datos obtenidos');
        }, 1000);
    });
```

```
}
```

**Módulos**: La importación y exportación de módulos ayuda a organizar el código en unidades reutilizables y mantenibles.

```
// Importar un módulo
import { calcularImpuestos } from './impuestos.js';
```

**Manejo de Errores**: Usa try...catch para capturar y manejar errores.

```
try {
    // Código que podría generar un error
    throw new Error('Error personalizado');
} catch (error) {
    console.error('Se produjo un error:', error.message);
}
```

| Manejo de la sintaxis del lenguaje | Descripción | Ejemplos |
|---|---|---|
| **Punto y Coma** | En JavaScript, el punto y coma indica el final de una instrucción. Aunque el motor de JavaScript puede inferir su presencia, es una buena práctica incluirlo para evitar errores sutiles. | **let x = 10;** <br> **const nombre = 'Ana';** |
| **Nombres de Variables** | Los nombres deben ser descriptivos y seguir la convención camelCase, donde cada nueva palabra comienza con una letra mayúscula, excepto la primera. | **let edadUsuario = 25;** <br> **const PI = 3.14;** |
| **Declaración de Variables** | let y const son las palabras clave para declarar variables. let permite reasignar valores, mientras que const es para valores que no cambiarán. | **let saldo = 100;** <br> **const tasaInteres = 0.05;** |
| **Funciones** | Las funciones se pueden declarar de manera tradicional o como funciones flecha, una sintaxis más corta y sin su propio contexto de this. | **function saludar(nombre) { return 'Hola, ' + nombre; }** <br> **const suma = (a, b) => a + b;** |
| **Cadenas de Texto** | JavaScript acepta comillas simples, dobles y template literals. Estos últimos son útiles para insertar expresiones dentro de una cadena. | **const mensaje = 'Bienvenido al mundo de JavaScript';** <br> **const plantilla =** \Hola, ${nombre}, ¿cómo estás?``; |
| **Comparaciones** | Para comparaciones que incluyan el tipo de dato, usa === y !==. Evita == y !=, que realizan una conversión de tipo automática. | **const edad = 18;** <br> **if (edad >= 18) { console.log('Eres mayor de edad'); }** |
| **Bloques de Código** | Usa llaves {} para definir bloques de código y mantén una indentación consistente para mejorar la legibilidad. | **function calcularArea(base, altura) {** <br> **const area = base * altura;** <br> **return area;** <br> **}** |
| **Comentarios** | Los comentarios son esenciales para explicar el código. Usa // para una sola línea o /* */ para múltiples líneas. | **// Este es un comentario de una sola línea** <br> **/*** <br> **Este es un comentario** <br> **de múltiples líneas** <br> ***/** |
| **Estructuras de Control** | if, else, switch, for, while y do...while son estructuras que controlan el flujo del programa. | **let temperatura = 25;** <br> **if (temperatura > 30) { console.log('Hace calor'); } else { console.log('Hace fresco'); }** |
| **Arrays** | Los arrays se declaran con corchetes y permiten almacenar listas ordenadas de valores. | **const colores = ['rojo', 'verde', 'azul'];** <br> **console.log(colores[1]); // Imprime 'verde'** |

| | | |
|---|---|---|
| **Objetos** | Los objetos se declaran con llaves y representan colecciones de pares clave-valor. | **const persona = { nombre: 'Carlos', edad: 30, profesion: 'Desarrollador' };** <br> **console.log(persona.nombre); // Imprime 'Carlos'** |
| **Truthy y Falsy** | Entiende qué valores se consideran verdaderos o falsos en contextos booleanos. | **const valor = 0;** <br> **if (valor) { console.log('Valor verdadero'); } else { console.log('Valor falso'); }** |
| **Asincronía** | JavaScript maneja operaciones asincrónicas mediante callbacks, promises y async/await. | **function obtenerDatos() { return new Promise((resolve, reject) => { setTimeout(() => { resolve('Datos obtenidos'); }, 1000); }); }** |
| **Módulos** | La importación y exportación de módulos ayuda a organizar el código en unidades reutilizables y mantenibles. | **// Importar un módulo** <br> **import { calcularImpuestos } from './impuestos.js';** |
| **Manejo de Errores** | Usa try...catch para capturar y manejar errores. | **try {** <br> **// Código que podría generar un error** <br> **throw new Error('Error personalizado');** <br> **} catch (error) {** <br> **console.error('Se produjo un error:', error.message);** <br> **}** |

## 2.2. VARIABLES

Las variables son contenedores para almacenar datos que pueden cambiar durante la ejecución de un programa. En JavaScript, se pueden declarar variables utilizando las palabras clave **var**, **let** y **const**.

- **Declaración**: Para crear una variable, utilizamos la palabra clave let o const, seguida del nombre que queremos darle. Por ejemplo: let edad;.
- **Asignación**: Luego, asignamos un valor a la variable utilizando el operador de asignación (=). Por ejemplo: edad = 25;.

**Tipos de Variables**

### 2.2.1. VARIABLES MUTABLES (**LET**):

- Puedes cambiar su valor después de declararlas.
- Útiles cuando necesitas actualizar la información.

```
let nombre = "Ana";
nombre = "Carlos"; // Cambiamos el valor de la variable
```

### 2.2.2. VARIABLES INMUTABLES (**CONST**):

- Su valor no puede cambiar después de la asignación inicial.
- Útiles para constantes o valores que no deben modificarse.

```
const PI = 3.14159;
```

**Buenas Prácticas**

- **Nombres Descriptivos**: Elige nombres significativos para tus variables. Por ejemplo, en lugar de x, usa edad o nombreCompleto.
- **Camel Case**: Si el nombre de la variable tiene varias palabras, únelas en camel case (la primera palabra en minúscula y las siguientes en mayúscula). Por ejemplo: nombreCompleto.
- **Inicialización**: Siempre asigna un valor inicial a tus variables para evitar errores.

**Ejemplo Práctico**

Supongamos que queremos almacenar la edad de una persona:

```
let edad = 30; // Declaración y asignación
console.log("La edad es:", edad); // Imprime el valor de la variable
```

```
// Ejemplo de declaración de variables
var edad = 25;
let nombre = "Juan";
const PI = 3.1416;
```

## 2.3. ÁMBITO DE UNA VARIABLE

¿Qué es el ámbito de una variable?

El ámbito de una variable se refiere a la parte del código en la que dicha variable puede ser utilizada. Es como una especie de "zona de acción" para la variable. Es importante entender esto para saber cómo y dónde podemos trabajar con nuestras variables.

En JavaScript, existen diferentes tipos de ámbitos. Por ejemplo, cuando declaramos una variable con la palabra clave var, su ámbito se extiende a lo largo de toda la función en la que se encuentra. Esto significa que podemos acceder a esa variable desde cualquier parte de la función, incluso si la declaramos dentro de bloques condicionales o bucles. Por ejemplo:

```
function ejemplo() {
    if (true) {
      var x = 10; // x puede ser usada en toda la función ejemplo()
    }
    console.log(x); // 10
  }
```

Sin embargo, si declaramos una variable con let o const, su ámbito se limita al bloque en el que se encuentra. Un bloque puede ser un bloque condicional (if, else, switch) o un bucle (for, while, do...while). Esto significa que solo podemos acceder a esa variable dentro de ese bloque en particular. Echemos un vistazo:

```
function ejemplo() {
    if (true) {
      let y = 20; // y solo puede ser usada dentro de este bloque if
      const z = 30; // z solo puede ser usada dentro de este bloque if
      console.log(y); // 20
      console.log(z); // 30
    }
    console.log(y); // Error: y no está definido en este ámbito
    console.log(z); // Error: z no está definido en este ámbito
  }
```

## 2.4. TIPO DE DATOS

### 2.4.1. NÚMEROS (NUMBER):

- Este tipo de datos se utiliza para representar valores numéricos, ya sean enteros o de punto flotante (decimales).
- Ejemplos: **let edad = 25;**, **let precio = 9.99;**

### 2.4.2. CADENAS DE TEXTO (STRING):

- Las cadenas de texto se utilizan para representar texto o caracteres alfanuméricos.
- Se pueden declarar utilizando comillas simples (') o dobles (").
- Ejemplos: **let nombre = 'Juan';**, **let mensaje = "Hola mundo";**

### 2.4.3. BOOLEANOS (BOOLEAN):

- Los booleanos son variables que pueden tener solo dos valores: **true** o **false**.
- Se utilizan para representar valores de verdad o falsedad en expresiones condicionales y de control.
- Ejemplos: **let esMayor = true;**, **let activado = false;**

### 2.4.4. OBJETOS (OBJECT):

- Los objetos son estructuras de datos complejas que pueden contener múltiples valores y métodos.
- Se componen de pares clave-valor, donde cada clave es una propiedad y cada valor puede ser de cualquier tipo de dato, incluidos otros objetos.

### 2.4.5. ARREGLOS (ARRAY):

- Los arreglos son listas ordenadas de valores, que pueden ser de cualquier tipo de dato.
- Se declaran utilizando corchetes (**[]**) y cada elemento del arreglo se separa por comas.
- Ejemplo: **let numeros = [1, 2, 3, 4, 5];**

### 2.4.6. VALORES ESPECIALES:

- JavaScript también tiene algunos valores especiales que representan la ausencia de valor (**null** y **undefined**) y valores no numéricos (**NaN**).
- **null**: se utiliza para representar la ausencia intencional de un valor.
- **undefined**: se utiliza para representar la falta de definición de una variable o propiedad.
- **NaN** (Not a Number): se utiliza para representar un valor que no es un número válido.
- Ejemplos: **let valorNulo = null;**, **let valorIndefinido;**, **let resultado = 10 / "texto";**

## 2.5. INTRODUCCIÓN A LAS FUNCIONES EN JAVASCRIPT

Las funciones son como recetas en programación: nos permiten reutilizar código para realizar una tarea específica cuantas veces queramos. En JavaScript, escribir una función es como decirle al navegador: "Aquí tienes un conjunto de pasos para hacer algo cuando te lo pida".

Por ejemplo, si queremos sumar dos números, definimos una función así:

```
function suma(a, b) {
    return a + b;
}
```

La palabra function le dice a JavaScript que estamos creando una función. Luego viene el nombre de la función, suma, y entre paréntesis, los ingredientes o **parámetros** (a y b) que necesita para trabajar.

**Recuerda:**

- Una función puede tener **cero, uno o varios parámetros**.
- El **cuerpo de la función**, donde está el código que se ejecuta, va entre **llaves** {}.
- Si queremos que la función nos dé algo de vuelta, usamos la palabra return.

Para usar nuestra función suma, hacemos esto:

```
var c = suma(3, 3); // c será 6
c = suma(c, c); // ahora c será 12
console.log ( "El valor de c será ahora: " + c);
```

Primero, llamamos a la función suma con dos números (3 y 3). Luego, usamos el resultado para llamar a suma otra vez. Al final, mostramos por consola un mensaje que incluye el resultado final.

## 2.6. PALABRAS RESERVADAS EN JAVASCRIPT:

Las palabras reservadas en JavaScript son términos que tienen un significado especial y están reservados por el propio lenguaje. Estas palabras tienen funciones específicas en el código y no pueden ser utilizadas como nombres de variables, funciones u otros identificadores en el programa. Utilizar una palabra reservada como identificador generará un error en el código.

- **break**: Se utiliza para salir de un bucle (**for**, **while**, **do-while**) o de una estructura de control (**switch**) de manera anticipada.

- **case**: Se utiliza en la estructura **switch** para definir los casos a comparar.
- **catch**: Se utiliza en bloques **try-catch** para capturar excepciones que pueden ocurrir en el código.
- **class**: Se utiliza para definir una clase en JavaScript para la programación orientada a objetos.
- **const**: Se utiliza para declarar una constante cuyo valor no puede ser reasignado después de la inicialización.
- **continue**: Se utiliza para saltar a la siguiente iteración de un bucle (**for**, **while**, **do-while**) sin ejecutar el resto del código en el ciclo actual.
- **debugger**: Se utiliza para detener la ejecución del código en un punto específico para propósitos de depuración.
- **default**: Se utiliza en la estructura **switch** como un caso por defecto cuando ninguno de los otros casos coincide.
- **delete**: Se utiliza para eliminar una propiedad de un objeto.
- **do**: Se utiliza para iniciar un bucle **do-while**.
- **else**: Se utiliza junto con **if** para ejecutar un bloque de código cuando la condición especificada en **if** es falsa.

## 2.7. CONVERSIÓN DE TIPOS DE DATOS

**Conversión Implícita:** JavaScript realiza conversiones implícitas de tipos de datos automáticamente en ciertas situaciones, como en operaciones aritméticas o concatenación de cadenas de texto.

```
let num = 10 + "5"; // Resultado: "105" (cadena de texto)
```

- Conversión de Número a Cadena de Texto **String()**:

En ocasiones necesitamos convertir un número en una cadena de texto. Esto puede ser útil si queremos mostrar el número como parte de un mensaje o si necesitamos combinarlo con otros textos.
Para hacer esta conversión, utilizamos la función String(). Esta función toma cualquier valor, como un número, y lo convierte en una cadena de texto.

```
// Ejemplo de conversión de número a cadena de texto
let numero = 123;
let cadenaDeTexto = String(numero);
console.log(cadenaDeTexto); // Salida: "123"
```

- Conversión de Cadena de Texto a Número **parseInt()** o **parseFloat()**:

Cuando programamos en JavaScript, a veces necesitamos cambiar un dato de un tipo a otro. Por ejemplo, si tienes un número en forma de texto (como “123”) y quieres sumarlo como si fuera un número real, necesitas convertirlo.

JavaScript es muy flexible y hace muchas de estas conversiones automáticamente, pero también nos ofrece herramientas especiales para hacerlo nosotros mismos. Aquí hay dos muy importantes:

- parseInt(): Esta función toma un texto que parece un número (como “123”) y lo transforma en un número entero (123).
- parseFloat(): Es similar a parseInt(), pero se usa para obtener números con decimales (como 123.45).

Estas funciones son muy útiles cuando queremos asegurarnos de que los datos con los que estamos trabajando son del tipo correcto para nuestras operaciones en JavaScript.

```
let texto = "10";
let numero = parseInt(texto); // Resultado: 10 (número entero)
```

- Conversión de Booleano a Cadena de Texto **String()**:

En JavaScript, a veces queremos convertir un valor booleano (es decir, true o false) en una cadena de texto. Esto puede ser útil si necesitas mostrar el valor en la pantalla o guardarlo como texto.
Para hacer esta conversión, usamos la función String(). Esta función toma cualquier valor, como un booleano, y lo convierte en una cadena de texto.

Aquí tienes un ejemplo:

```
// Ejemplo de conversión de booleano a cadena de texto
let valorBooleano = true;
let cadenaDeTexto = String(valorBooleano);
console.log(cadenaDeTexto); // Salida: "true"
```

- Conversión de Cadena de Texto a Booleano **Boolean()**:

```
let texto = "true";
let booleano = Boolean(texto); // Resultado: true (booleano)
```

- Conversión de Número a Booleano **Boolean()** :

Cualquier número distinto de cero se convierte en **true**, y cero se convierte en **false**.

```
let numero = 10;
let booleano = Boolean(numero); // Resultado: true (booleano)
```

- Conversión de Booleano a Número **Number()**:

  **true** se convierte en 1 y **false** se convierte en 0.

```
let booleano = true;
let numero = Number(booleano); // Resultado: 1 (número)
```

- Conversión de Objeto a Cadena de Texto **toString()**:

```
let objeto = { nombre: "Juan", edad: 25 };
let texto = objeto.toString(); // Resultado: "[object Object]"
```

- Conversión de Cadena de Texto a Objeto **JSON.parse()**:

  Utilizando la función **JSON.parse()** para convertir una cadena de texto en un objeto JSON.

```
let texto = '{"nombre": "Juan", "edad": 25}';
let objeto = JSON.parse(texto); // Resultado: { nombre: "Juan", edad: 25 }
```

## 2.8. OPERADORES

Los operadores son las herramientas que te permiten realizar acciones específicas con los valores de tus variables. En JavaScript, hay varios tipos de operadores que puedes usar para hacer desde simples cálculos matemáticos hasta tomar decisiones lógicas en tu código.

### 2.8.1. OPERADORES DE ASIGNACIÓN:

Son fundamentales para modificar el valor de una variable de forma eficiente. Además del operador básico de asignación (**=**), existen operadores compuestos que combinan la asignación con una operación aritmética, como **+=**, **-=**, ***=**, **/=**, entre otros.

| Operador | Ejemplo | Resultado |
|---|---|---|
| **Asignación (=)** | let x = 5; | x ahora es 5 |
| **Asignación de adición (+=)** | x += 3; | x ahora es 8 |
| **Asignación de resta (-=)** | x -= 3; | x ahora es 2 |

### 2.8.2. OPERADORES DE COMPARACIÓN:

Los operadores de comparación en JavaScript son esenciales para evaluar condiciones y tomar decisiones en el código. Desde comparar valores simples hasta verificar igualdad estricta, estos operadores permiten una amplia gama de validaciones. Los operadores más comunes incluyen **==** para igualdad, **!=** para desigualdad, **===** para igualdad estricta (comparando valor y tipo), **!==** para desigualdad estricta, **>** para mayor que, **<** para menor que, **>=** para mayor o igual que, y **<=** para menor o igual que.

| Operador | Ejemplo | Resultado |
|---|---|---|
| **Igualdad (==)** | 5 == '5'; | Verdadero |
| **Igualdad estricta (===)** | 5 === '5'; | Falso |

### 2.8.3. OPERADORES LÓGICOS:

Son herramientas vitales para evaluar múltiples condiciones y controlar el flujo del programa. Estos operadores permiten combinar expresiones booleanas para formar condiciones más complejas. Los operadores lógicos más comunes son **&&** (AND lógico), **||** (OR lógico) y **!** (NOT lógico). El operador **&&** devuelve **true** si ambas expresiones son verdaderas, **||** devuelve **true** si al menos una expresión es verdadera, y **!** invierte el valor de una expresión booleana.

| Operador | Ejemplo | Resultado | Descripción |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **AND (&&)** | true && false; | false | Ambos operandos deben ser verdaderos para que el resultado sea verdadero. |
| **OR (`\|\|')** | True \|\| false | True | Al menos uno de los operandos deben ser verdaderos para que el resultado sea verdadero. |
| **NOT (!)** | !true; | false | Invierte el valor booleano del operando. |

Ejemplo con operadores de comparación, lógicos y de asignación

```
// Ejemplo con operadores de comparación y lógicos
let edad = 18;
let acceso = (edad >= 18) && (edad < 65); // acceso es verdadero si la edad
está entre 18 y 65

// Ejemplo con operadores de asignación
let puntos = 10;
puntos += 20; // puntos ahora es 30
```

Ejemplo con operadores aritméticos

```
// Ejemplo con operadores aritméticos
let manzanas = 3;
let manzanasConseguidas = 2;
let totalManzanas = manzanas + manzanasConseguidas; // totalManzanas será
5
```

Ejemplo de operadores dentro de una sentencia

```
// Ejemplo con operadores lógicos
let cieloAzul = true;
let solBrillando = false;

if (cieloAzul && solBrillando) {
    console.log("¡Es un día perfecto para un picnic!");
}
else if (cieloAzul || solBrillando) {
    console.log("Aún es un buen día para salir.");
}
else {
    console.log("Quizás mejor nos quedamos en casa.");
}
```

## 2.9. COMENTARIOS AL CÓDIGO EN JAVASCRIPT:

Los comentarios en JavaScript son fragmentos de texto que se utilizan para explicar el funcionamiento del código y hacerlo más legible para los desarrolladores. Los comentarios no se ejecutan y son ignorados por el intérprete de JavaScript.

En JavaScript, existen dos tipos de comentarios: comentarios de una sola línea y comentarios de varias líneas.

Los comentarios de una sola línea se crean precediendo el texto con **//**, y todo lo que sigue después de **//** en la misma línea se considera un comentario.

```
// Este es un comentario de una sola línea
let x = 10; // Se inicializa la variable x con el valor 10
```

Los comentarios de varias líneas se crean entre **/*** y ***/**, y todo lo que esté dentro de estos delimitadores se considera un comentario.

```
/*
Este es un comentario
de varias líneas
que abarca múltiples
líneas de código
*/
let y = 20; // Se inicializa la variable y con el valor 20
```

## 2.10. SENTENCIAS EN JAVASCRIPT:

Las sentencias en JavaScript son unidades de código que realizan una acción o una serie de acciones. Cada sentencia en JavaScript termina con un punto y coma **;**. Las sentencias pueden ser simples o compuestas, y se utilizan para controlar el flujo del programa y realizar operaciones.

Las sentencias pueden incluir declaraciones de variables, llamadas a funciones, estructuras de control de flujo (como **if**, **else**, **for**, **while**, **do-while**), expresiones y comentarios.

Por ejemplo, una sentencia simple puede ser la declaración de una variable:

```
```

```
let nombre = 'Juan'; // Sentencia de declaración de variable
```

Una sentencia compuesta puede ser un bloque de código dentro de una estructura de control de flujo:

```
let x = 10; // Se inicializa la variable x con el valor 10

if (edad >= 18) { // Sentencia de estructura de control if
    console.log('Es mayor de edad'); // Sentencia de llamada a función
} else {
    console.log('Es menor de edad'); // Otra sentencia de llamada a función
}
```

## 2.11. SENTENCIAS O TOMA DE DECISIONES

La toma de decisiones es un aspecto fundamental, donde los programas deben ser capaces de responder de manera inteligente a diferentes situaciones. En JavaScript, esto se logra mediante estructuras de control de flujo, como las sentencias **if**, **else if** y **else**.

### 2.11.1. SENTENCIA IF

La sentencia **if** permite ejecutar un bloque de código si se cumple una condición específica. Si la condición es verdadera, se ejecuta el bloque de código dentro del **if**, de lo contrario, se pasa al siguiente bloque de código.

```
let edad = 18;
if (edad >= 18) {
    console.log('Es mayor de edad');
}
```

### 2.11.2. SENTENCIA ELSE IF

La sentencia **else if** permite evaluar múltiples condiciones en secuencia. Si la primera condición no se cumple, se evalúa la siguiente condición y se ejecuta el bloque de código correspondiente si se cumple.

```
let hora = 14;
if (hora < 12) {
    console.log('Buenos días');
} else if (hora < 18) {
    console.log('Buenas tardes');
} else {
    console.log('Buenas noches');
}
```

### 2.11.3. SENTENCIA ELSE

La sentencia **else** se utiliza para ejecutar un bloque de código si ninguna de las condiciones anteriores se cumple. Es opcional y se coloca al final de una serie de sentencias **if** o **else if**.

```
let dia = 'sábado';
if (dia === 'domingo') {
    console.log('Es día de descanso');
} else {
    console.log('Es día laborable');
}
```

### 2.11.4. OPERADOR TERNARIO

El operador ternario es una forma abreviada de escribir una sentencia **if-else** en una sola línea. Consiste en una expresión seguida de **?**, luego la expresión que se ejecuta si la condición es verdadera, seguida de **:** y la expresión que se ejecuta si la condición es falsa.

```
let edad = 20;
let mensaje = edad >= 18 ? "Es mayor de edad" : "Es menor de edad";
console.log(mensaje);
```

Este código verifica si la persona con 20 años es mayor de edad y muestra un mensaje apropiado en la consola. Es una forma eficiente de escribir una condición sin necesidad de usar varias líneas de código con if...else.

### 2.11.5. SWITCH

La sentencia **switch** evalúa una expresión y ejecuta el código asociado con el primer caso que coincide. Puede tener múltiples casos y un caso predeterminado para manejar condiciones que no coinciden con ninguno de los casos especificados.

```
let dia = 3;
switch (dia) {
  case 1:
    console.log("Lunes");
    break;
  case 2:
    console.log("Martes");
    break;
  case 3:
    console.log("Miércoles");
    break;
  default:
    console.log("Otro día");
}
```

En este ejemplo, como **dia** tiene el valor **3**, se ejecuta el caso **case 3**, imprimiendo "Miércoles" en la consola. La sentencia **break** es importante porque detiene la ejecución del **switch** una vez que se ha encontrado un caso que coincide, evitando que los casos siguientes sean evaluados innecesariamente.

### 2.11.6. TRY...CATCH

La sentencia **try...catch** se utiliza para manejar excepciones. El bloque **try** contiene el código que puede lanzar una excepción, mientras que el bloque **catch** se ejecuta si se produce una excepción en el bloque **try**. Esto permite manejar errores de manera controlada y tomar decisiones en consecuencia.

En el siguiente ejemplo se puede ver la resolución ante un error.

```
try {
    // Código que puede lanzar una excepción
    let resultado = 10 / 0;
  } catch (error) {
    // Código que maneja la excepción
    console.log("Se ha producido un error:", error.message);
  }
```

Este código intenta realizar una operación matemática inválida (dividir por cero) y, cuando se produce la excepción, muestra un mensaje de error en la consola. El uso de bloques try y catch nos permite controlar cómo manejamos las excepciones en nuestro código.

Es una buena práctica para evitar que los errores detengan por completo la ejecución de nuestro programa.

## 2.12. BUCLES EN JAVASCRIPT

### INTRODUCCIÓN A LOS BUCLES

En la programación, encontramos tareas repetitivas que, si las hiciéramos manualmente, nos tomarían una eternidad. Aquí es donde entran los **bucles**, una de las estructuras fundamentales en JavaScript que nos permiten automatizar y simplificar estas tareas.

#### 2.12.1. BUCLES FOR

El bucle for es nuestro aliado cuando conocemos el número exacto de veces que queremos que se ejecute un bloque de código. Su sintaxis es la siguiente:

```
for (inicialización; condición; actualización) {
  // Código a ejecutar en cada iteración
}
```

**Ejemplo Práctico:**

Imaginemos que queremos imprimir en consola los números del 1 al 5. Con un bucle for, lo hacemos así:

```
for (let numero = 1; numero <= 5; numero++) {
  console.log(numero);
}
```

**Consejos para el Uso del Bucle** for

- Utiliza el bucle for cuando sepas cuántas iteraciones necesitas.
- Asegúrate de actualizar correctamente la variable dentro del bucle para evitar bucles infinitos.

### 2.12.2. BUCLES WHILE

El bucle **while** ejecuta repetidamente un bloque de código mientras una condición especificada sea verdadera. Antes de cada iteración, se evalúa la condición. Si la condición es verdadera, el bloque de código se ejecuta. Si la condición es falsa, el bucle se detiene.

El bucle while es ideal cuando la cantidad de repeticiones depende de una condición que no conocemos de antemano. Su estructura es más sencilla:

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

**Ejemplo Práctico:**

Este bucle imprimirá "El contador es: 0", "El contador es: 1", ..., "El contador es: 4", ya que se detiene cuando **contador** es igual a 5.

```
let contador = 0;
while (contador < 5) {
  console.log("El contador es: " + contador);
  contador++;
}
```

**Consejos para el Uso del Bucle** while

- Utiliza el bucle while cuando no sepas cuántas iteraciones necesitas y la condición dependa de una variable que puede cambiar.
- Asegúrate de actualizar correctamente las variables dentro del bucle para evitar bucles infinitos.

### 2.12.3. BUCLE DO...WHILE

El bucle **do...while** es similar al bucle **while**, pero **garantiza** que el bloque de código se ejecute al menos una vez, incluso si la condición es inicialmente falsa. Después de cada iteración, se evalúa la condición. Si es verdadera, el bucle se repite; de lo contrario, el bucle se detiene.

```
let contador = 0;
```

```
do {
  console.log("El contador es: " + contador);
  contador++;
} while (contador < 5);
```

Este bucle también imprimirá "El contador es: 0", "El contador es: 1", ..., "El contador es: 4".

**Consejos para el Uso del Bucle** do...while

- Utiliza el bucle do...while cuando necesitas ejecutar un código al menos una vez antes de verificar la condición.
- Asegúrate de actualizar correctamente las variables dentro del bucle para evitar bucles infinitos.

### 2.12.4. BUCLE FOR...IN

El bucle **for...in** se utiliza para iterar sobre las propiedades de un objeto. En cada iteración, una variable toma el nombre de una propiedad del objeto.

```
const persona = {
    nombre: 'Juan',
    edad: 30,
    ciudad: 'Madrid'
  };

  for (let clave in persona) {
    console.log(clave + ": " + persona[clave]);
  }
```

Este bucle imprimirá "nombre: Juan", "edad: 30", "ciudad: Madrid", iterando sobre cada propiedad del objeto **persona**.

**Consejos para el Uso del Bucle** for...in

- Utiliza el bucle for...in para recorrer las propiedades de los objetos y obtener sus claves y valores.
- Ten cuidado al usar for...in en arrays, ya que también iterará sobre las propiedades adicionales y métodos del prototipo del array.

### 2.12.5. BUCLE FOR...OF

El bucle **for...of** se utiliza para iterar sobre elementos de objetos iterables, como arreglos y cadenas de texto.

```
```

```
const colores = ['rojo', 'verde', 'azul'];
for (let color of colores) {
  console.log(color);
}
```

Este bucle imprimirá "rojo", "verde" y "azul", iterando sobre cada elemento del array **colores**.

**Consejos para el Uso del Bucle** for...of

- Utiliza el bucle for...of cuando necesitas iterar sobre los valores de una colección.
- Es especialmente útil para recorrer arrays, strings y otros objetos iterables.

### CUADRO COMPARATIVO DE BUCLES

Cada uno de estos bucles tiene su propia sintaxis y aplicación específica en JavaScript. Es importante entender cómo y cuándo utilizar cada uno para escribir código efectivo y eficiente.

| Bucle | Descripción | Ejemplo |
|---|---|---|
| **while** | Ejecuta un bloque de código mientras una condición especificada sea verdadera. | while (condición) { /* código */ } |
| **do...while** | Ejecuta un bloque de código al menos una vez y luego repite mientras una condición sea verdadera. | do { /* código */ } while (condición); |
| **for** | Crea un bucle con un contador y ejecuta un bloque de código mientras se cumpla una condición. | for (inicialización; condición; incremento) { /* código */ } |
| **for...in** | Itera sobre las propiedades de un objeto. | for (variable in objeto) { /* código */ } |
| **for...of** | Itera sobre los elementos de un objeto iterable, como un arreglo o una cadena de texto. | for (variable of iterable) { /* código */ } |

## 2.13. INTRODUCCIÓN A LOS ARRAYS

Un array, en términos simples, es una colección ordenada de elementos. Cada elemento en un array tiene una posición, conocida como su índice, que comienza desde 0. Esto significa que el primer elemento del array está en el índice 0, el segundo en el índice 1, y así sucesivamente.

### ¿Por qué usar Arrays?

Los arrays son especialmente útiles cuando necesitas almacenar y manejar una cantidad de datos que está relacionada o es del mismo tipo. Por ejemplo, si quisieras mantener una lista de las calificaciones de un estudiante, en lugar de usar múltiples variables, podrías simplemente usar un array.

## Creando Arrays

JavaScript ofrece varias formas de crear arrays. Aquí te muestro las más comunes.

### Literal de Array

La forma más sencilla de crear un array es usando corchetes []. Por ejemplo:

```
let frutas = ["manzana", "banana", "cereza"];
console.log(frutas);
```

## ACCESO Y MODIFICACIÓN DE ELEMENTOS

Acceder a un elemento en un array es sencillo: simplemente usa el índice del elemento dentro de corchetes. Para modificar un elemento, asigna un nuevo valor al índice correspondiente.

```
let frutas = ["manzana", "banana", "cereza"];
console.log(frutas[1]); // Salida: banana

// Cambiar el valor
frutas[1] = "mango";
console.log(frutas[1]); // Salida: mango
```

## Métodos Comunes de Arrays

JavaScript proporciona una variedad de métodos útiles para trabajar con arrays. Aquí exploraremos algunos de los más utilizados.

### push y pop

push(): Añade uno o más elementos al final del array.

pop(): Elimina el último elemento de un array y lo devuelve.

```
let colores = ["rojo", "verde", "azul"];
colores.push("amarillo");
console.log(colores); // Salida: ["rojo", "verde", "azul", "amarillo"]

let ultimo = colores.pop();
console.log(ultimo); // Salida: "amarillo"
```

```
console.log(colores); // Salida: ["rojo", "verde", "azul"]
```

### shift y unshift

shift(): Elimina el primer elemento del array y lo devuelve.

unshift(): Añade uno o más elementos al inicio del array.

```
let colores = ["rojo", "verde", "azul"];
colores.unshift("amarillo");
console.log(colores); // Salida: ["amarillo", "rojo", "verde", "azul"]

let primero = colores.shift();
console.log(primero); // Salida: "amarillo"
console.log(colores); // Salida: ["rojo", "verde", "azul"]
```

### Recorriendo Arrays

Para recorrer un array, puedes utilizar el bucle for tradicional, for...of o métodos como forEach.

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

// Uso de for tradicional
for (let i = 0; i < numeros.length; i++) {
  console.log(numeros[i]);
}

// Uso de forEach
numeros.forEach(function(numero) {
  console.log(numero);
});

// Uso de for...of
for (let numero of numeros) {
  console.log(numero);
}
```

## 2.14. PRÁCTICA GUIADA

**Práctica Guiada: Lista de Tareas con HTML, CSS y JavaScript**

En esta práctica guiada, crearemos una aplicación de lista de tareas donde los usuarios puedan añadir tareas, verlas en una lista y eliminarlas. Utilizaremos HTML para la estructura, CSS para el estilo y JavaScript para la funcionalidad, integrando el uso de arrays y métodos de manipulación de arrays.

**Objetivo:**

Desarrollar una aplicación de lista de tareas que permita a los usuarios gestionar sus tareas diarias. Aprenderás a manipular arrays en JavaScript y a interactuar con el DOM para reflejar esos cambios.

**Duración:**

La actividad debería completarse en 1 hora, aproximadamente.

**Paso 1: Estructura HTML**

Crea un archivo HTML llamado index.html. Este contendrá un formulario para añadir tareas, un botón para añadirlas y una lista donde se mostrarán.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Lista de Tareas</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>Lista de Tareas</h1>
    <form id="formularioTareas">
        <input type="text" id="entradaTarea" placeholder="Añade una nueva
tarea">
```

```
        <button type="submit">Añadir Tarea</button>
    </form>
    <ul id="listaTareas">
    </ul>
    <script src="script.js"></script>
</body>
</html>
```

**Paso 2: Estilo con CSS**

Crea un archivo CSS llamado style.css para mejorar la apariencia de la página.

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    margin: 40px;
    text-align: center;
}

#formularioTareas input[type="text"] {
    padding: 10px;
    width: 300px;
    margin-right: 5px;
}

#listaTareas {
    margin-top: 20px;
    list-style: none;
}
```

```css
#listaTareas li {
    background-color: #fff;
    margin-top: 10px;
    padding: 10px;
    border-radius: 5px;
    cursor: pointer;
}

#listaTareas li:hover {
    background-color: #e9e9e9;
}
```

**Paso 3: Funcionalidad con JavaScript**

Crea un archivo JavaScript llamado script.js. Aquí, implementaremos la adición de tareas al array y su visualización en el DOM. También permitiremos eliminar tareas al hacer clic en ellas.

```javascript
// Se declara un array para almacenar las tareas.
let listaDeTareas = [];

// Se añade un escuchador de eventos al formulario para manejar el envío del formulario.
document.getElementById('formularioTareas').addEventListener('submit', function(evento) {
    // Previene el comportamiento por defecto del formulario, que es recargar la página.
    evento.preventDefault();

    // Obtiene el valor ingresado en el campo de texto del formulario.
```

```
    let nuevaTarea = document.getElementById('entradaTarea').value;


    // Verifica si el texto ingresado no está vacío (después de quitar
espacios en blanco).
    if (nuevaTarea.trim() !== '') {
        // Añade la nueva tarea al array de tareas.
        listaDeTareas.push(nuevaTarea);
        // Llama a la función que actualiza la lista de tareas en el DOM.
        actualizarListaDOM();
        // Limpia el campo de entrada para que esté listo para una nueva
entrada.
        document.getElementById('entradaTarea').value = '';
    }
});


// Función que actualiza el contenido de la lista de tareas en el DOM.
function actualizarListaDOM() {
    // Selecciona el elemento ul donde se mostrarán las tareas.
    let listaUl = document.getElementById('listaTareas');
    // Limpia el contenido actual del elemento ul para evitar duplicaciones.
    listaUl.innerHTML = '';


    // Itera sobre el array de tareas, utilizando cada tarea y su índice.
    listaDeTareas.forEach((tarea, indice) => {
        // Crea un nuevo elemento li para cada tarea.
        let li = document.createElement('li');
        // Establece el contenido de texto del li al valor de la tarea
actual.
        li.textContent = tarea;
```

```
        // Añade un escuchador de clics al li que manejará la eliminación
de la tarea.
        li.onclick = function() {
            // Elimina la tarea del array basándose en su índice.
            listaDeTareas.splice(indice, 1);
            // Actualiza el DOM para reflejar los cambios en el array de
tareas.
            actualizarListaDOM();
        };
        // Añade el nuevo elemento li al ul.
        listaUl.appendChild(li);
    });
}
```

Este código JavaScript maneja una lista de tareas utilizando un array y manipulación del DOM. Al enviar el formulario, se añade una tarea al array y se actualiza la vista del usuario. Cada tarea en la lista puede eliminarse al hacer clic en ella, lo cual actualiza tanto el array como la vista del DOM.

El uso de evento.preventDefault() es crucial para evitar que el formulario se envíe de la manera tradicional (recargando la página), permitiendo que la página mantenga su estado actual y maneje la lógica de forma dinámica con JavaScript.

## 2.15. EJERCICIOS PRÁCTICOS DE ARRAY

1. Crea un Array: Define un array llamado animales que contenga los siguientes elementos: 'perro', 'gato', 'caballo'.
2. Accede a Elementos Específicos: Imprime el tercer elemento del array animales.
3. Modificar Elementos: Cambia el segundo elemento de animales a 'tigre'.
4. Agregar y Eliminar Elementos: Añade 'conejo' al final del array animales y elimina el primer elemento.
5. Recorrer un Array: Utiliza un bucle para imprimir cada elemento del array animales.
6. Array de Números: Crea un array números con los siguientes elementos: 10, 20, 30, 40, 50. Usa forEach para sumar todos los elementos y mostrar el total.
7. Búsqueda en Array: Dado el array números, encuentra e imprime el primer número mayor que 25 utilizando el método find.

## 2.16. EJERCICIOS DE LA UNIDAD:

**Objetivo:**

El objetivo principal de estos ejercicios es que apliques y refuerces los conocimientos que has adquirido sobre la sintaxis de JavaScript. Al completar estos ejercicios, serás capaz de manipular variables, interactuar con el usuario a través de la consola y el método prompt(), y utilizar operadores y estructuras de control de flujo de manera efectiva. Estos ejercicios están diseñados para que practiques la creación de funciones, la conversión de tipos de datos, la implementación de eventos en elementos HTML, y el uso de bucles y sentencias condicionales para resolver problemas comunes de programación.

Además, estos ejercicios te ayudarán a familiarizarte con la depuración y el uso de herramientas del navegador, como la consola, para verificar y corregir tu código. Al finalizar los ejercicios, habrás desarrollado una comprensión más profunda de cómo escribir código limpio y eficiente, documentar tu trabajo con comentarios, y utilizar técnicas de programación modernas para crear aplicaciones web dinámicas y funcionales.

**Duración**: 3.5hs.

**Entrega:** Para la entrega de los ejercicios de la Unidad 2, debes crear una carpeta llamada Unidad2_Ejercicios que contenga un archivo HTML (index.html) y un archivo JavaScript (scripts.js) en los cuales hayas implementado y comentado todas las soluciones de los ejercicios propuestos. Asegúrate de que el archivo HTML esté correctamente enlazado al archivo JavaScript y que todo el código esté claramente organizado y documentado. Una vez que hayas terminado, comprime la carpeta Unidad2_Ejercicios en un archivo ZIP y súbelo a la plataforma.

- Escribe un programa que calcule el área y la longitud de una circunferencia de radio 5, utilizando las variables area y longitud.
- Crea un programa que pida al operador su nombre por un ejemplo completo de uso del prompt() y lo escriba en la consola junto con un mensaje de bienvenida.
- Desarrolla un script que permita saber el tipo de una variable utilizando el operador typeof. Crea ejemplos con variables de tipo string, number, boolean y object.
- Escribe una función que tome como parámetro una cadena de texto (string) y la convierta en un número entero utilizando parseInt(). Comprueba si el resultado es un número par o impar.
- Crea un programa que utilice el método addEventListener para detectar cuando el ratón pasa por encima de un elemento HTML específico y cambie su color de fondo.
- Escribe un programa que solicite al usuario ingresar su edad y utilice una sentencia if-else para determinar si es mayor o menor de edad. Muestra el resultado en la consola.
- Crea una función que reciba dos parámetros numéricos y devuelva el resultado de elevar el primer número a la potencia del segundo utilizando el operador de exponenciación (**).
- Desarrolla un script que utilice un bucle while para generar una tabla de multiplicar del número 7, mostrando los resultados en la consola.
- Escribe un programa que solicite al usuario ingresar un número del 1 al 7 y utilice una sentencia switch para mostrar el día de la semana correspondiente (por ejemplo, 1 = Lunes, 2 = Martes, etc.).

## 2.17. RESUMEN

- Los programadores utilizan la sentencia console.log() para escribir información por consola. El operador de decimales en JavaScript es el punto y no la coma.
- JavaScript es sensible a las mayúsculas y minúsculas. Es importante diferenciar entre mayúsculas y minúsculas al escribir código.
- Los tipos de datos en Javascript son 5 string, number, boolean, array y object.
- Se utiliza null en las variables para indicar que no existen o que no tienen todavía un valor asignado.
- Existen funciones como parseInt() o parseFloat() para convertir datos como String a Number. También se pueden utilizar las funciones toFixed(), toString() o toUTCString() dependiendo de los tipos de datos a convertir.
- Un operador importante es el de asignación (=). Con el método addEventListener se puede vincular un evento a una función JavaScript.
- Los eventos son cualquier suceso que puede ocurrir a un elemento HTML como onclick, onchange, onmouseover, etc.
- Con el método innerHTML se puede acceder al contenido de un elemento HTML desde JavaScript.
- Las palabras reservadas en JavaScript, como function, var, let, const, entre otras, no pueden ser utilizadas como nombres de variables o funciones.
- Las funciones se pueden declarar utilizando la palabra clave function, seguida del nombre de la función y los parámetros entre paréntesis. El código a ejecutar se coloca entre llaves {}.
- JavaScript permite la creación de objetos utilizando la sintaxis de llaves {}, donde se definen pares de clave-valor separados por comas.
- Los comentarios en el código son importantes para documentar y explicar el funcionamiento del programa. Se pueden utilizar comentarios de una línea con // o de múltiples líneas con /* */.
- Las sentencias de control de flujo, como if, else if, else, switch, permiten tomar decisiones basadas en condiciones específicas.
- Los bucles, como for, while y do...while, se utilizan para repetir un bloque de código varias veces, basándose en una condición o un número determinado de iteraciones.
- Es importante entender el ámbito de las variables, ya que determina su visibilidad y accesibilidad dentro del código. Las variables declaradas con var tienen un ámbito de función, mientras que let y const tienen un ámbito de bloque.

## 2.18. BIBLIOGRAFÍA


- Mozilla Developer Network (MDN) URL: https://developer.mozilla.org/
- W3Schools URL: https://www.w3schools.com/
- DesarrolloWeb.com URL: https://desarrolloweb.com/
- AMX Web URL: https://amxweb.com/