UNIVERSIDAD NACIONAL DEL CENTRO DEL PERÚ

FACULTAD DE INGENIERÍA DE SISTEMAS

MONOGRAFÍA

INTRODUCCIÓN A JAVASCRIPT Y PRINCIPIOS BÁSICOS

PRESENTADO POR:

BOZA LARA, Dany Lenin

CLEMENTE QUINTANA, Junior Ivan

FIGUEROA INGA, Magaly Marycielo

LAIZA SOTACURO, Jenifer Fiorella

ROJAS MALDONADO, Jampiero Daylor

Docente: Ing. Jaime. Suasnabar Terrel

HUANCAYO – PERÚ

2025

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN:

JavaScript es uno de los lenguajes de programación más utilizados en el desarrollo web. Su versatilidad y capacidad para ejecutarse directamente en los navegadores permiten la creación de páginas dinámicas, interactivas y funcionales. Este informe tiene como objetivo explorar los conceptos fundamentales de JavaScript en su nivel básico, el abordar estos temas esenciales proporcionará una base sólida para el desarrollo web moderno y la creación de experiencias de usuario más completas.

OBJETIVO GENERAL

Explicar y aplicar los conceptos básicos de JavaScript, comprendiendo su funcionamiento

OBJETIVO ESPECÍFICO

- Explicar el funcionamiento de JavaScript y su integración con los navegadores.
- Identificar y utilizar variables, tipos de datos y operadores en scripts básicos.
- Implementar estructuras de control como condicionales y bucles.
- Definir y ejecutar funciones con diferentes tipos de parámetros y retornos.
- Manipular el DOM para modificar contenido y responder a eventos del usuario.
- Utilizar eventos y manejadores para mejorar la interactividad de la página.
- Comprender el uso de JSON y el almacenamiento local con `localStorage` y `sessionStorage`.
- Aplicar técnicas básicas de depuración y uso de la consola para detectar errores.

JavaScript es una herramienta esencial para cualquier desarrollador web. Comprender sus fundamentos no solo permite crear sitios web más atractivos y funcionales, sino que también abre la puerta al uso de tecnologías más avanzadas. Aprender JavaScript básico es el primer paso para dominar frameworks modernos como React, Angular o Vue, y para integrarse efectivamente en equipos de desarrollo de software. En un entorno donde la tecnología digital es cada vez más relevante, tener conocimientos en JavaScript representa una competencia clave en el mercado laboral actual.

Para el desarrollo del informe se realizó una investigación documental apoyada en fuentes académicas, manuales oficiales y recursos en línea. Se llevaron a cabo ejercicios prácticos para aplicar cada concepto abordado, utilizando editores de código como Visual Studio Code y herramientas del navegador (como la consola y el inspector de elementos) para la verificación y depuración de los scripts.

DESARROLLO

1. Introducción a JavaScript y su ejecución en navegadores

JavaScript es un lenguaje de programación interpretado y orientado a objetos, ampliamente utilizado en el desarrollo web para crear páginas interactivas. Junto con HTML y CSS, forma la base del desarrollo del frontend. Este informe aborda los conceptos básicos de JavaScript, su ejecución en los navegadores y el papel fundamental de los motores de JavaScript. [1]

1.1. ¿Qué es JavaScript?

JavaScript es un lenguaje de scripting que permite implementar funcionalidades dinámicas en sitios web. Se ejecuta principalmente del lado del cliente, es decir, dentro del navegador del usuario. Algunas de sus aplicaciones comunes incluyen:

- Validación de formularios
- Interacción con el usuario mediante eventos
- Manipulación dinámica del contenido HTML (DOM)
- Llamadas a servidores mediante AJAX o fetch
- Animaciones y efectos visuales

1.2. Ejecución de JavaScript en Navegadores

Los navegadores web modernos como Google Chrome, Mozilla Firefox, Safari y Microsoft Edge tienen incorporados motores de JavaScript que interpretan y ejecutan el código JavaScript que se encuentra dentro de las páginas web. [1]

Proceso de ejecución:

- El navegador carga la estructura de la página (HTML).
- Encuentra las etiquetas <script> y ejecuta el código JavaScript.
- El motor del navegador interpreta el código y lo ejecuta en el entorno del cliente.

1.3. ¿Qué es un Motor de JavaScript (JavaScript Engine)?

Un motor de JavaScript es el componente responsable de interpretar y ejecutar el código JavaScript. Se encuentra embebido dentro del navegador o en entornos como Node.js.

Funciones principales de un motor JS:

- Leer (parsear) el código fuente
- Compilarlo o interpretarlo
- Ejecutarlo
- Administrar memoria (recolección de basura)
- Optimizar el rendimiento automáticamente

Principales motores de JavaScript:

- Chrome/ Node.js → usa el motor **V8**
- Firefox → usa **SpiderMonkey**
- Safari → usa **JavaScriptCore**

JavaScript es esencial para el desarrollo web moderno, permitiendo interactividad y dinamismo en los sitios. Su ejecución depende de los motores integrados en los navegadores, los cuales han evolucionado para ofrecer un rendimiento altamente optimizado. Entender cómo se ejecuta JavaScript y qué papel cumple el motor es fundamental para desarrollar aplicaciones web eficientes y modernas. [1]

2. Variables y tipos de datos

JavaScript es un lenguaje de programación interpretado ampliamente utilizado para desarrollar aplicaciones web interactivas. Una de sus bases más importantes es el manejo de variables y tipos de datos, que permiten almacenar, procesar y manipular información. En este informe se explican los principales tipos de datos en JavaScript, su clasificación en primitivos y no primitivos, y se proporciona un ejemplo práctico con código comentado. [2]

2.1. ¿Qué es una variable?

Una variable en JavaScript es un contenedor que almacena un valor en memoria. Se pueden declarar utilizando las siguientes palabras clave:

- let: Permite crear variables con alcance limitado al bloque donde se declaran.
- const: Crea constantes cuyo valor no puede cambiar.
- var: Sintaxis antigua con alcance de función (no recomendable en código moderno).

2.2. Clasificación de los Tipos de Datos

2.2.1. Datos Primitivos

Los datos primitivos son los tipos más simples que representa JavaScript. Se almacenan por valor directamente en la variable y son inmutables. [2]

Tabla 1. Tipos de datos primitivos

Tipo	**Descripción**	**Ejemplo**
String	Texto plano	"Hola", 'Mundo'
Number	Números enteros o decimales	25, 3.14
Boolean	Valor lógico	true, false
Undefined	Variable declarada sin valor	let x;
Null	Ausencia intencional de valor	let dato = null
Symbol	Identificador único	Symbol("id")
BigInt	Números enteros extremadamente grandes	123456789012345678901n

Fuente: Elaboración propia

Los valores primitivos no tienen métodos ni propiedades, aunque JavaScript a veces los envuelve temporalmente en objetos para operar con ellos (por ejemplo, length en strings).

2.2.2. Datos No Primitivos

Los datos no primitivos, también llamados estructurados, son más complejos. Se almacenan por referencia (no por valor) y son mutables. [2]

Tabla 2. Tipos de datos no primitivos

Tipo	**Descripción**	**Ejemplo**
Object	Conjunto de pares clave-valor	{ nombre: "Ana", edad: 20 }
Array	Lista ordenada de elementos	["rojo", "verde", "azul"]
Function	Código ejecutable	function saludar() { ... }
Otros	Estructuras avanzadas como Map, Set, Date, etc.	

Fuente: Elaboración propia

Ejemplo de tipos:

```
// ======================

// DATOS PRIMITIVOS

// ======================
```

```
// STRING: Texto plano

let nombre = "Juan"; // Tipo: String

// NUMBER: Número entero o decimal

let edad = 30; // Tipo: Number

// BOOLEAN: Lógico, verdadero o falso

let esEstudiante = true; // Tipo: Boolean

// UNDEFINED: Variable declarada pero sin valor

let sinValor; // Tipo: Undefined

// NULL: Valor nulo intencionalmente

let datos = null; // Tipo: Null

// SYMBOL: Identificador único

let id = Symbol("id"); // Tipo: Symbol

// BIGINT: Números muy grandes

let numeroGrande = 1234567890123456789012345678901234567890n; // Tipo: BigInt

// ========================

// DATOS NO PRIMITIVOS

// ========================

// ARRAY: Lista ordenada de elementos

let frutas = ["manzana", "plátano", "uva"]; // Tipo: Array

// OBJECT: Agrupación de pares clave-valor

let persona = {

  nombre: "Ana",

  edad: 25,
```

```
  esEstudiante: true
}; // Tipo: Object
// FUNCTION: Bloque de código ejecutable
function saludar() {
  return "Hola, " + nombre;
}
// ======================
// EJEMPLOS DE USO
// ======================
console.log("Nombre:", nombre);
console.log("Edad:", edad);
console.log("¿Es estudiante?", esEstudiante);
console.log("Frutas:", frutas);
console.log("Primera fruta:", frutas[0]);
console.log("Persona:", persona);
console.log("Saludo:", saludar());
console.log("Número grande:", numeroGrande);
```

Comprender la diferencia entre datos primitivos y no primitivos es esencial para escribir código eficiente y predecible en JavaScript. Mientras que los primitivos permiten manejar datos simples de forma directa, los no primitivos ofrecen estructuras más complejas y flexibles para representar entidades, colecciones y lógica del programa.

3. Operadores

Los operadores básicos son símbolos que permiten efectuar distintas operaciones con variables y valores, estos se agrupan en varias categorías, como los operadores aritméticos,

de comparación y los lógicos. Cada grupo cumple una función específica y se emplea según el tipo de acción que se necesite realizar en el código. [3]

3.1. Operadores Aritméticos

Los operadores aritméticos permiten realizar cálculos matemáticos simples entre números. Son los mismos que usamos en la matemática cotidiana. En JavaScript, estos operadores incluyen, suma, resta, etc. [4]

- Suma (+): Añade dos valores.
- Resta (-): Resta un valor de otro.
- Multiplicación (*): Multiplica dos valores.
- División (/): Divide un valor entre otro.
- Módulo (%): Devuelve el residuo de una división.
- Incremento (++): Aumenta el valor de una variable en una unidad.
- Decremento (--): Disminuye el valor de una variable en una unidad.

Estos operadores son esenciales para manipular datos numéricos, especialmente en bucles o cálculos dentro de funciones.

Figura 1. Ejercicios con los operadores Aritméticos

```
// Declaramos dos variables
let a = 10;
let b = 3;

// Suma
let suma = a + b;
console.log("Suma:", suma); // 13

// Resta
let resta = a - b;
console.log("Resta:", resta); // 7

// Multiplicación
let multiplicacion = a * b;
console.log("Multiplicación:", multiplicacion); // 30

// División
let division = a / b;
console.log("División:", division); // 3.333...

// Módulo (resto de la división)
let modulo = a % b;
console.log("Módulo:", modulo); // 1

// Potenciación
let potencia = a ** b;
console.log("Potencia:", potencia); // 1000 (10^3)

// Incremento
a++;
console.log("Incremento de a:", a); // 11

// Decremento
b--;
console.log("Decremento de b:", b); // 2
```

Fuente: Elaboración propia

Figura 2. Resultado en el terminal

```
Suma: 13
Resta: 7
Multiplicación: 30
División: 3.3333333333333335
Módulo: 1
Potencia: 1000
Incremento de a: 11
Decremento de b: 2
```

Fuente: Elaboración propia

3.2. Operadores Lógicos

Los operadores lógicos permiten combinar múltiples condiciones en una sola expresión, lo cual es muy útil en decisiones complejas. Los principales operadores son:

- AND (&&): Devuelve true sólo si todas las condiciones son verdaderas.
- OR (||): Devuelve true si al menos una de las condiciones es verdadera.
- NOT (!): Invierte el valor lógico; si algo es true, lo convierte en false, y viceversa.

Estos operadores son comúnmente usados en sentencias condicionales y ciclos de control de flujo. [5]

Ejemplos:

Figura 3. Ejercicios con operadores Lógicos

```
1
2   let a = true;
3   let b = false;
4   let c = true;
5
6   // Operador AND (&&): Devuelve true solo si ambas condiciones son verdaderas
7   let andResult = a && c;  // true porque ambas son verdaderas
8   console.log("Resultado AND (&&):", andResult); // true
9
10  // Operador OR (||): Devuelve true si al menos una de las condiciones es verdadera
11  let orResult = a || b;  // true porque 'a' es verdadera
12  console.log("Resultado OR (||):", orResult); // true
13
14  // Operador NOT (!): Invierte el valor de la condición
15  let notResult = !b;  // true porque 'b' es false, y al aplicar NOT se convierte en true
16  console.log("Resultado NOT (!):", notResult); // true
17
18  // Combinación de los tres operadores
19  let combinedResult = (a && c) || !b;  // (true && true) || !false -> true || true -> true
20  console.log("Combinación de AND, OR y NOT:", combinedResult); // true
```

Fuente: Elaboración propia

Figura 4. Resultado de los operadores lógicos en el terminal

```
Resultado AND (&&): true
Resultado OR (||): true
Resultado AND (&&): true
Resultado OR (||): true
Resultado OR (||): true
Resultado NOT (!): true
Combinación de AND, OR y NOT: true
```

Fuente: Elaboración propia

3.3. Operadores de Comparación

Estos operadores permiten comparar dos valores y devuelven un valor booleano (true o false), lo cual es fundamental para estructuras condicionales como if, else o bucles while. Algunos son:

- Igualdad débil (==): Compara si dos valores son equivalentes (aunque sean de diferente tipo).
- Desigualdad (!=): Verifica si dos valores no son iguales.
- Mayor que (>): Comprueba si un valor es mayor que otro.
- Menor que (<): Comprueba si un valor es menor que otro.
- Mayor o igual (>=): Verifica si un valor es mayor o igual que otro.
- Menor o igual (<=): Verifica si un valor es menor o igual que otro.

En JavaScript también existen operadores de comparación estricta (===, !==) que comparan valor y tipo. [6]

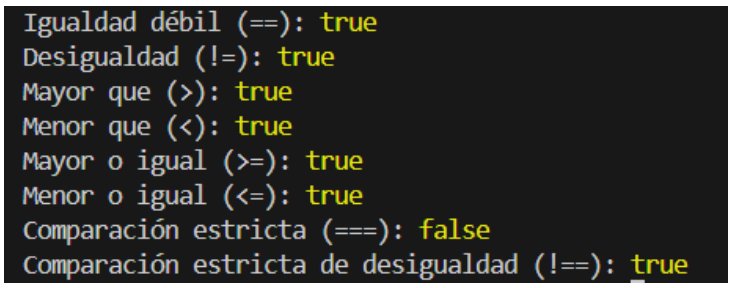
Ejemplo:

Figura 5. Ejercicios con operadores de comparación

```
let x = 10;
let y = 5;
let z = '10';

// Igualdad débil (==)
let igual = x == z;  // true, porque 10 es igual a '10' (en tipo string, pero ambos valores son equivalentes)
console.log("Igualdad débil (==):", igual); // true

// Desigualdad (!=)
let desigual = x != y;  // true, porque 10 no es igual a 5
console.log("Desigualdad (!=):", desigual); // true

// Mayor que (>)
let mayor = x > y;  // true, porque 10 es mayor que 5
console.log("Mayor que (>):", mayor); // true

// Menor que (<)
let menor = y < x;  // true, porque 5 es menor que 10
console.log("Menor que (<):", menor); // true

// Mayor o igual (>=)
let mayorIgual = x >= y;  // true, porque 10 es mayor o igual que 5
console.log("Mayor o igual (>=):", mayorIgual); // true

// Menor o igual (<=)
let menorIgual = y <= x;  // true, porque 5 es menor o igual que 10
console.log("Menor o igual (<=):", menorIgual); // true

// Comparación estricta (===)
let estrictaIgual = x === z;  // false, porque aunque ambos sean 10, 'x' es un número y 'z' es un string
console.log("Comparación estricta (===):", estrictaIgual); // false

// Comparación estricta de desigualdad (!==)
let estrictaDesigual = x !== z;  // true, porque el tipo de x es diferente al de z
console.log("Comparación estricta de desigualdad (!==):", estrictaDesigual); // true
```

Fuente: Elaboración propia

Figura 6. Resultados de los operadores de comparación en el terminal



Fuente: Elaboración propia

4. Condicionales

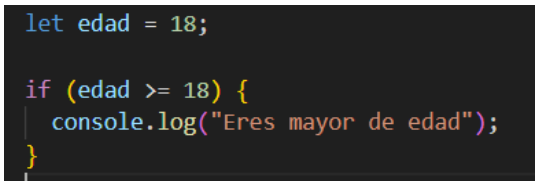
La sintaxis de JavaScript permite controlar el flujo del programa mediante estructuras como if, if/else y switch. Estas herramientas permiten ejecutar diferentes bloques de código según si se cumplen ciertas condiciones. Son fundamentales para desarrollar aplicaciones interactivas y flexibles que respondan a decisiones lógicas. [7]

4.1. Condicional If

El uso del if en JavaScript es esencial para trabajar con condiciones. Permite ejecutar un bloque de código únicamente cuando se cumple una determinada condición. Dominar su uso es importante para controlar el flujo lógico de una aplicación. [8] Su sintaxis es sencilla: se escribe if, seguido de una condición entre paréntesis y el código a ejecutar dentro de llaves.

Ejemplo:

Figura 7. Ejercicios con la condicional if

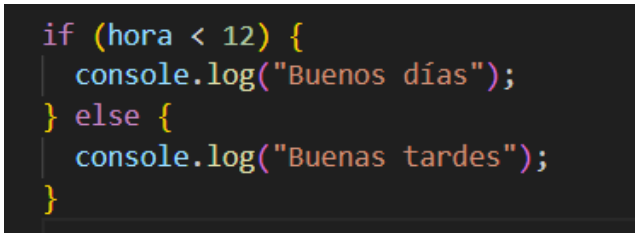


Fuente: Elaboración propia

4.2. Condicional If/Else

La instrucción if/else en JavaScript permite manejar el flujo del programa según diferentes escenarios. Es útil para ejecutar distintos bloques de código dependiendo de si se cumple o no una condición. Esta estructura evalúa una expresión que retorna true o false: si el resultado es verdadero, se ejecuta el bloque dentro del if; de lo contrario, se ejecuta el bloque contenido en else. [7]

Figura 8. Ejercicios con la condicional if



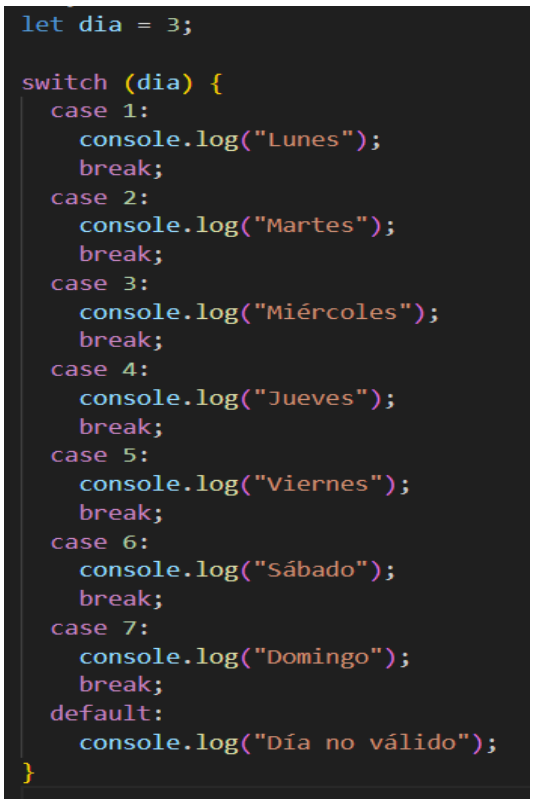
```
if (hora < 12) {
  console.log("Buenos días");
} else {
  console.log("Buenas tardes");
}
```

Fuente: Elaboración propia

4.3. Condicional Switch

La instrucción switch en JavaScript posee una sintaxis sencilla y clara. Comienza con la palabra clave switch, seguida de la expresión a evaluar entre paréntesis. Luego, se especifican los distintos posibles valores usando case, acompañados del código que debe ejecutarse. Para evitar que se ejecuten otros casos innecesariamente, es fundamental utilizar break al final de cada bloque. [7]

Figura 9. Ejercicios con la condicional switch



```
let dia = 3;

switch (dia) {
  case 1:
    console.log("Lunes");
    break;
  case 2:
    console.log("Martes");
    break;
  case 3:
    console.log("Miércoles");
    break;
  case 4:
    console.log("Jueves");
    break;
  case 5:
    console.log("Viernes");
    break;
  case 6:
    console.log("Sábado");
    break;
  case 7:
    console.log("Domingo");
    break;
  default:
    console.log("Día no válido");
}
```

Fuente: Elaboración propia

Ejercicio:

Figura 10. Evaluación de stock y generación de pedido en Script

```
// Datos del producto
let stockActual = 8;
let stockMinimo = 10;
let precioUnitario = 5.5;
let clienteFrecuente = true;

// Verificamos si es necesario hacer un pedido
if (stockActual < stockMinimo) {
  console.log("⚠️ ¡Atención! El stock es bajo. Se recomienda hacer un nuevo pedido.");

  // Cálculo del pedido sugerido
  let cantidadPedido = stockMinimo - stockActual + 5;
  let costoTotal = cantidadPedido * precioUnitario;

  // Aplicamos descuento si el cliente es frecuente y el monto es mayor a 50
  if (clienteFrecuente && costoTotal > 50) {
    costoTotal *= 0.9; // 10% de descuento
    console.log("🎉 Cliente frecuente: se aplicó un 10% de descuento.");
  }

  console.log("📦 Cantidad sugerida a pedir:", cantidadPedido);
  console.log("💰 Costo total estimado del pedido:", costoTotal.toFixed(2));
} else {
  console.log("✅ El stock actual es suficiente. No se requiere hacer pedidos.");
}
```

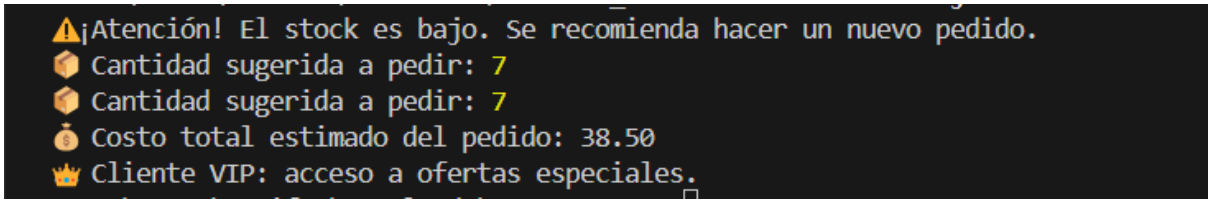
Fuente: Elaboración propia

Figura 11. Evaluación del tipo de cliente en Script

```
switch (tipoCliente) {
  case "Nuevo":
    console.log("📄 Bienvenido, nuevo cliente. Gracias por confiar en nosotros.");
    break;
  case "Frecuente":
    console.log("📈 Cliente frecuente: beneficios activos.");
    break;
  case "VIP":
    console.log("👑 Cliente VIP: acceso a ofertas especiales.");
    break;
  default:
    console.log("🤔 Tipo de cliente no identificado.");
}
```

Fuente: Elaboración propia

Figura 12. Resultados de la evaluación de stock y generación de pedido en Script



Fuente: Elaboración propia

5. Bucles

Los bucles son estructuras fundamentales en JavaScript que permiten ejecutar un bloque de código repetidamente mientras se cumpla una condición específica. Existen principalmente tres tipos de bucles en JavaScript: `for`, `while` y `do-while`. Cada uno tiene sus propias características y casos de uso ideales.

5.1. Bucle For

El bucle `for` es probablemente el más utilizado cuando conocemos de antemano el número de iteraciones que queremos realizar.

```
for (inicialización; condición; expresión-final) {
  // código a ejecutar en cada iteración
}
```

Según Mozilla Developer Network (MDN), "La declaración for crea un bucle que consiste en tres expresiones opcionales, encerradas entre paréntesis y separadas por punto y coma, seguidas de una declaración o secuencia de declaraciones que se ejecutarán en el bucle".[9]

Figura 13. Ejercicio bucle for

Fuente: Elaboración propia

5.2. Bucle While

El bucle `while` ejecuta su bloque de código mientras la condición especificada sea verdadera. Es útil cuando no sabemos exactamente cuántas iteraciones necesitaremos.

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

De acuerdo con el libro "JavaScript: The Definitive Guide" de David Flanagan, "El bucle while es el más simple de JavaScript. Ejecuta un bloque de código mientras la expresión de control se evalúa como verdadera". [10]

Figura 14. Ejercicio bucle while

```
console.log("\n===== EJEMPLO DE BUCLE WHILE =====");  '\n===== EJEMPLO DE BUCLE WHILE ====='

function aproximarNumeroAureo(precision) {
  let fibAnterior = 0;
  let fibActual = 1;
  let contador = 0;
  let razonAnterior = 0;

  console.log("Aproximación al número áureo usando la secuencia de Fibonacci:");  'Aproximación al núme
  console.log("Iteración | Fibonacci | Razón | Diferencia con iteración anterior");  'Iteración | Fibon
  console.log("-----------------------------------------------------------");  '-------------------------

  while (true) {
    // Calculamos el siguiente número de Fibonacci
    const fibSiguiente = fibAnterior + fibActual;

    // Calculamos la razón entre el número actual y el anterior
    const razon = fibActual > 0 ? fibSiguiente / fibActual : 0;

    // Calculamos la diferencia con la razón anterior
    const diferencia = Math.abs(razon - razonAnterior);

    // Mostramos resultados
    if (contador > 0) {
      console.log(`${contador.toString().padStart(9, ' ')} | ${fibActual.toString().padStart(9, ' ')} |
      ${razon.toFixed(8)} | ${diferencia.toFixed(8)}`);
    }

    // Verificamos si hemos alcanzado la precisión deseada
    if (contador > 0 && diferencia < precision) {
      console.log(`\nConvergencia alcanzada con precisión de ${precision}`);  '\nConvergencia alcanzada
      console.log(`Aproximación del número áureo (ϕ): ${razon.toFixed(10)}`);  'Aproximación del número
      console.log(`Valor real de ϕ: 1.6180339887...`);  'Valor real de ϕ: 1.6180339887...'
      break;
    }
```

Fuente: Elaboración propia

Figura 15. Continuación del ejercicio bucle while

```
    // Actualizamos variables para la siguiente iteración
    fibAnterior = fibActual;
    fibActual = fibSiguiente;
    razonAnterior = razon;
    contador++;

    // Limitamos el número de iteraciones por seguridad
    if (contador > 30) {
      console.log("Límite de iteraciones alcanzado");
      break;
    }
  }
}

aproximarNumeroAureo(0.0000001);
```

Fuente: Elaboración propia

5.3. Bucle Do-While

El bucle `do-while` es similar al bucle `while`, pero con una diferencia crucial: garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa desde el principio.

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

La principal diferencia entre el bucle while y do-while es que do-while siempre ejecuta su cuerpo al menos una vez, mientras que while puede no ejecutar su cuerpo en absoluto si la condición es inicialmente falsa [11]

Figura 16. Ejercicio bucle Do-While

```
console.log("\n===== EJEMPLO DE BUCLE DO-WHILE =====");  '\n===== EJEMPLO DE BUCLE DO-WHILE ====='

function simularLanzamientoDados() {
  // Configuración inicial
  let lanzamientos = 0;
  let sumaDados = 0;
  let frecuencias = {};
  for (let i = 2; i <= 12; i++) {
    frecuencias[i] = 0;
  }

  let dobleSeisConsecutivo = 0;
  let maxLanzamientosSinSiete = 0;
  let lanzamientosSinSieteActual = 0;

  // Simulación de lanzamientos
  do {
    // Simulamos el lanzamiento de dos dados
    const dado1 = Math.floor(Math.random() * 6) + 1;
    const dado2 = Math.floor(Math.random() * 6) + 1;
    const total = dado1 + dado2;

    lanzamientos++;
    sumaDados += total;
    frecuencias[total]++;

    // Comprobamos si hemos lanzado doble seis
    if (dado1 === 6 && dado2 === 6) {
      dobleSeisConsecutivo++;
      if (dobleSeisConsecutivo >= 2) {
        console.log(`¡Doble seis consecutivo en los lanzamientos ${lanzamientos-1} y ${lanzamientos}!`);
      }
    } else {
      dobleSeisConsecutivo = 0;
    }
```

Fuente: Elaboración propia

Figura 17. Continuación del ejercicio Do-While

```
    if (total === 7) {
      if (lanzamientosSinSieteActual > maxLanzamientosSinSiete) {
        maxLanzamientosSinSiete = lanzamientosSinSieteActual;
      }
      lanzamientosSinSieteActual = 0;
    } else {
      lanzamientosSinSieteActual++;
    }
      // Condición de salida: obtenemos tres sietes o alcanzamos 30 lanzamientos
  } while (frecuencias[7] < 3 && lanzamientos < 30);
    // Actualizamos estadística final
  if (lanzamientosSinSieteActual > maxLanzamientosSinSiete) {
    maxLanzamientosSinSiete = lanzamientosSinSieteActual;
  }
    // Mostramos estadísticas
  console.log(`Simulación finalizada después de ${lanzamientos} lanzamientos`);  'Simulación finaliz
  console.log(`Promedio obtenido: ${(sumaDados / lanzamientos).toFixed(2)}`);  'Promedio obtenido: 7
  console.log(`Mayor racha sin obtener un siete: ${maxLanzamientosSinSiete} lanzamientos`);  'Mayor

  console.log("\nDistribución de resultados:");  '\nDistribución de resultados:'
  console.log("Valor | Frecuencia | Porcentaje | Gráfico");  'Valor | Frecuencia | Porcentaje | Gráf
  console.log("------|------------|------------|--------");  '------|------------|------------|-----
   for (let i = 2; i <= 12; i++) {
    const porcentaje = (frecuencias[i] / lanzamientos * 100).toFixed(1);
    const barra = '*'.repeat(Math.round(frecuencias[i] / lanzamientos * 40));
    console.log(`${i.toString().padStart(5, ' ')} | ${frecuencias[i].toString().padStart(10, ' ')} |
    ${porcentaje.padStart(10, ' ')}% | ${barra}`);
  }
    return {
    lanzamientos,
    promedio: sumaDados / lanzamientos,
    maxRachaSinSiete: maxLanzamientosSinSiete,
    frecuencias
  };
}
simularLanzamientoDados();
```

Fuente: Elaboración propia

Comparación entre bucles

La elección entre los diferentes tipos de bucles depende principalmente de:

- For: Ideal para iteraciones con un número conocido de repeticiones.
- While: Preferible cuando la condición de salida depende de algo más que un simple contador.
- Do-While: Útil cuando se necesita garantizar que el código se ejecute al menos una vez.

A menudo, es una cuestión de legibilidad y la naturaleza del problema que estás resolviendo lo que determina qué tipo de bucle es más apropiado. [12]

Figura 18. Ejercicio demostrando los tres tipos de bucles

```
/*
 * Este script combina los tres tipos de bucles (for, while, do-while) para:
 * 1. Generar un rango de números para analizar
 * 2. Encontrar números primos dentro de ese rango
 * 3. Calcular estadísticas sobre los números primos encontrados
 * 4. Mostrar los resultados de manera formateada
 */

// Función para verificar si un número es primo
function esPrimo(numero) {
  // Los números menores o iguales a 1 no son primos
  if (numero <= 1) return false;

  // 2 y 3 son primos
  if (numero <= 3) return true;

  // Si es divisible por 2 o 3, no es primo
  if (numero % 2 === 0 || numero % 3 === 0) return false;

  // Verificamos divisibilidad por números de la forma 6k ± 1
  let i = 5;
  while (i * i <= numero) {
    if (numero % i === 0 || numero % (i + 2) === 0) return false;
    i += 6;
  }

  return true;
}

// Función principal que combina diferentes tipos de bucles
function analizarPrimos(inicio, fin) {
  console.log(`Analizando números primos entre ${inicio} y ${fin}:`);  'Analizando números primos entre 1 y 1000:'
```

Fuente: Elaboración propia

Figura 19. Continuación de ejercicio

```
let numerosPrimos = [];
let sumaPrimos = 0;
let contadorPrimos = 0;
let maximoPrimo = 0;
let digitosFrecuentes = Array(10).fill(0); // Para contar frecuencia de dígito

// Usamos un bucle for para recorrer el rango de números
for (let numero = inicio; numero <= fin; numero++) {
  if (esPrimo(numero)) {
    numerosPrimos.push(numero);
    sumaPrimos += numero;
    contadorPrimos++;

    if (numero > maximoPrimo) {
      maximoPrimo = numero;
    }

    // Usamos do-while para analizar cada dígito del número primo
    let tempNumero = numero;
    do {
      let digito = tempNumero % 10;
      digitosFrecuentes[digito]++;
      tempNumero = Math.floor(tempNumero / 10);
    } while (tempNumero > 0);
  }
}

// Calculamos estadísticas
const promedioPrimos = contadorPrimos > 0 ? sumaPrimos / contadorPrimos : 0;

// Encontramos el dígito más frecuente usando while
let digitoMasFrecuente = 0;
let maxFrecuencia = 0;
let i = 0;
```

Fuente : Elaboración propia

Figura 20. Continuación de ejercicio

```
while (i < digitosFrecuentes.length) {
  if (digitosFrecuentes[i] > maxFrecuencia) {
    maxFrecuencia = digitosFrecuentes[i];
    digitoMasFrecuente = i;
  }
  i++;
}

// Agrupamos los números primos por longitud de dígitos
let agrupadosPorDigitos = {};
for (let i = 0; i < numerosPrimos.length; i++) {
  const longitud = numerosPrimos[i].toString().length;
  if (!agrupadosPorDigitos[longitud]) {
    agrupadosPorDigitos[longitud] = [];
  }
  agrupadosPorDigitos[longitud].push(numerosPrimos[i]);
}

// Mostramos resultados
console.log(`\nSe encontraron ${contadorPrimos} números primos`);  '\nSe encontraron 168 números primos'
console.log(`Suma total: ${sumaPrimos}`);  'Suma total: 76127'
console.log(`Promedio: ${promedioPrimos.toFixed(2)}`);  'Promedio: 453.14'
console.log(`Número primo más grande: ${maximoPrimo}`);  'Número primo más grande: 997'
console.log(`Dígito más frecuente en números primos: ${digitoMasFrecuente} (aparece ${maxFrecuencia} veces)`);

console.log("\nPrimos agrupados por cantidad de dígitos:");  '\nPrimos agrupados por cantidad de dígitos:'
let longitudes = Object.keys(agrupadosPorDigitos).sort((a, b) => a - b);

// Usando do-while para mostrar los grupos
let j = 0;
do {
  const longitud = longitudes[j];
  console.log(`${longitud} dígito(s): ${agrupadosPorDigitos[longitud].length} números`);  '1 dígito(s): 4 númer
```

Fuente : Elaboración propia

Figura 21. Continuación de ejercicio

```
    // Solo mostramos los primeros 5 para no saturar la consola
    if (agrupadosPorDigitos[longitud].length > 0) {
      let muestra = agrupadosPorDigitos[longitud].slice(0, 5);
      console.log(`  Muestra: ${muestra.join(', ')}${agrupadosPorDigitos[longitud].length > 5 ? '...' : ''}`);  ' Mu
    }

    j++;
  } while (j < longitudes.length);

  return {
    cantidad: contadorPrimos,
    suma: sumaPrimos,
    promedio: promedioPrimos,
    maximo: maximoPrimo,
    digitoMasFrecuente: digitoMasFrecuente,
    frecuenciaDigito: maxFrecuencia,
    primos: numerosPrimos
  };
}

// Ejecutamos el análisis en un rango de números
const resultados = analizarPrimos(1, 1000);

// Buscamos patrones en los números primos usando diferentes tipos de bucles
console.log("\n--- Análisis de patrones en números primos ---");  '\n--- Análisis de patrones en números primos ---'
```

Fuente: Elaboración propia

Figura 22. Continuación de ejercicio

```
  // Usamos for para buscar combinaciones de dos primos que sumen el número par
  for (let i = 0; i < resultados.primos.length && resultados.primos[i] < numeroPar; i++) {
    let primerPrimo = resultados.primos[i];
    let segundoPrimo = numeroPar - primerPrimo;

    // Verificamos si el segundo número también es primo
    if (esPrimo(segundoPrimo)) {
      console.log(`${numeroPar} = ${primerPrimo} + ${segundoPrimo}`);   '10 = 3 + 7', '20 = 
      encontrado = true;
      break; // Solo mostramos la primera combinación
    }
  }

  if (!encontrado) {
    console.log(`No se encontró combinación de primos para ${numeroPar}`);
  }

  indice++;
}
```

Fuente : Elaboración propia

Figura 23. Impresión en consola de los resultados

```
> 'Analizando números primos entre 1 y 1000:'
> '\nSe encontraron 168 números primos'
> 'Suma total: 76127'
> 'Promedio: 453.14'
> 'Número primo más grande: 997'
> 'Dígito más frecuente en números primos: 1 (aparece 78 veces)'
> '\nPrimos agrupados por cantidad de dígitos:'
> '1 dígito(s): 4 números'
> '  Muestra: 2, 3, 5, 7'
```

Fuente : Elaboración propia

6. Funciones

Las funciones son uno de los bloques fundamentales en JavaScript. Permiten estructurar el código, hacerlo reutilizable y mantenerlo organizado. Vamos a explorar los diferentes tipos y aspectos de las funciones en JavaScript.

6.1. Tipos de funciones

6.1.1. Declaración de Funciones

La declaración de funciones es la forma más común de definir una función:

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

Las declaraciones de funciones son elevadas (hoisted), lo que significa que pueden ser utilizadas antes de su declaración en el código [13]

6.1.2. Expresiones de Funciones

Una expresión de función define una función como parte de una expresión:

```
const saludar = function(nombre) {
  return "Hola, " + nombre + "!";
};
```

A diferencia de las declaraciones, las expresiones de funciones no son elevadas y solo pueden ser utilizadas después de su definición [14]

6.1.3. Funciones flecha (Arrow functions)

Introducidas en ES6, las funciones flecha ofrecen una sintaxis más concisa:

```
const saludar = (nombre) => {
  return "Hola, " + nombre + "!";
};

// Versión más corta para funciones de una sola línea
const saludarCorto = nombre => "Hola, " + nombre + "!";
```

Las funciones flecha no tienen su propio `this`, `arguments`, `super` o `new.target`, y no pueden ser utilizadas como constructores. [15]

6.2. Parámetros de funciones

6.2.1. Parámetros básicos

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

6.2.2. Parámetros predeterminados

En ES6, se introdujeron los parámetros predeterminados:

```
function saludar(nombre = "Invitado") {
   return "Hola, " + nombre + "!";
}
```

6.2.3. Parámetros rest

Permiten representar un número indefinido de argumentos como un array:

```
function sumar(...numeros) {
  return numeros.reduce((suma, numero) => suma + numero, 0);
}
```

6.2.4. Desestructuración de parámetros

```
function mostrarPersona({nombre, edad}) {
   console.log(`${nombre} tiene ${edad} años`);
}
```

6.3. Retorno de funciones

6.3.1. Retorno explícito

```
function multiplicar(a, b) {
```

```
   return a * b;
}
```

6.3.2. Retorno implícito (en funciones flecha)

```
const multiplicar = (a, b) => a * b;
```

6.3.3. Retorno de múltiples valores

```
function obtenerDimensiones() {
 return {
    ancho: 100,
    alto: 200
  };
}
```

6.4. Funciones avanzadas

6.4.1. Funciones inmediatamente invocadas (IIFE)

```
(function() {
  console.log("Esta función se ejecuta inmediatamente");
})();
```

6.4.2. Funciones de orden superior

Funciones que toman otras funciones como argumentos o las devuelven como resultado [10]

```
function operarNumeros(a, b, operacion) {
    return operacion(a, b);
}
```

6.4.3. Funciones generadoras

Introducidas en ES6, permiten pausar y reanudar la ejecución:

```
function* generadorNumeros() {
    yield 1;
    yield 2;
    yield 3;
}
```

Figura 24. Ejercicio utilizando todas las funciones

```
/**
 * Ejercicio de Funciones JavaScript (Nivel 4)
 *
 * Objetivo: Crear un sistema de gestión de productos para una tienda online
 * utilizando diferentes tipos de funciones en JavaScript.
 */

// 1. Declaración de función: Para inicializar la base de datos de productos
function inicializarBaseDeDatos() {
  console.log("Base de datos de productos inicializada");  'Base de datos de
  return [
    { id: 1, nombre: "Laptop", precio: 1200, categoria: "electrónica" },
    { id: 2, nombre: "Smartphone", precio: 800, categoria: "electrónica" },
    { id: 3, nombre: "Auriculares", precio: 100, categoria: "accesorios" },
    { id: 4, nombre: "Teclado", precio: 50, categoria: "accesorios" },
    { id: 5, nombre: "Monitor", precio: 300, categoria: "electrónica" }
  ];
}

// 2. Expresión de función: Para buscar productos por su ID
const buscarProductoPorId = function(productos, id) {
  return productos.find(producto => producto.id === id);
};

// 3. Función flecha: Para filtrar productos por categoría
const filtrarPorCategoria = (productos, categoria) => {
  return productos.filter(producto => producto.categoria === categoria);
};

// 4. Función con parámetros predeterminados: Para aplicar descuento
function aplicarDescuento(precio, porcentaje = 10) {
  return precio - (precio * porcentaje / 100);
}
```

Fuente : Elaboración propia

Figura 25. Continuación de ejercicio utilizando todas las funciones

```
// 5. Función con parámetros rest: Para calcular el precio total de varios productos
function calcularTotal(...precios) {
  return precios.reduce((total, precio) => total + precio, 0);
}

// 6. Función con desestructuración de parámetros
function mostrarInformacionProducto({ id, nombre, precio }) {
  return `Producto #${id}: ${nombre} - $${precio}`;
}

// 7. Función de orden superior: Para ordenar productos por precio
function ordenarProductos(productos, criterioOrdenamiento) {
  return [...productos].sort(criterioOrdenamiento);
}

// 8. IIFE (Función inmediatamente invocada) para configurar el sistema
const configuracion = (function() {
  const impuestos = 0.16;
  const moneda = "USD";
  return {
    obtenerImpuestos: () => impuestos,
    obtenerMoneda: () => moneda,
    formatearPrecio: (precio) => `${precio} ${moneda}`
  };
})();

// 9. Función generadora: Para generar IDs únicos para nuevos productos
function* generadorId() {
  let id = 6; // Continuamos desde el último ID existente
  while (true) {
    yield id++;
  }
}
```

Fuente : Elaboración propia

Figura 26. Continuación de ejercicio utilizando todas las funciones

```
// Demostración del uso de las funciones

// Inicializamos la base de datos
const productos = inicializarBaseDeDatos();
console.log("Productos cargados:", productos);  ... a: 'accesorios' }, { id: 5, nombre: 'Monitor', precio: 300, categoria:

// Buscamos un producto por ID
const producto2 = buscarProductoPorId(productos, 2);
console.log("Producto encontrado:", producto2);  [ 'Producto encontrado:', { id: 2, nombre: 'Smartphone', precio: 800, cat

// Filtramos productos por categoría
const productosElectronicos = filtrarPorCategoria(productos, "electrónica");
console.log("Productos electrónicos:", productosElectronicos);  ... lectrónica' }, { id: 5, nombre: 'Monitor', precio: 300

// Aplicamos descuento a un producto
const precioConDescuento = aplicarDescuento(producto2.precio, 15);
console.log(`Precio original: $${producto2.precio}, Con 15% descuento: $${precioConDescuento}`);  'Precio original: $800,

// Aplicamos descuento predeterminado (10%)
const precioConDescuentoPredeterminado = aplicarDescuento(producto2.precio);
console.log(`Precio original: $${producto2.precio}, Con descuento predeterminado: $${precioConDescuentoPredeterminado}`);

// Calculamos el total de varios precios
const totalCompra = calcularTotal(productos[0].precio, productos[2].precio, productos[3].precio);
console.log("Total de la compra:", totalCompra);  [ 'Total de la compra:', 1350 ]

// Mostramos información del producto usando desestructuración
console.log(mostrarInformacionProducto(productos[0]));  'Producto #1: Laptop - $1200'

// Ordenamos productos por precio (de menor a mayor)
const productosOrdenados = ordenarProductos(productos, (a, b) => a.precio - b.precio);
console.log("Productos ordenados por precio:", productosOrdenados);  ... lectrónica' }, { id: 1, nombre: 'Laptop', precio:
```

Fuente : Elaboración propia

Figura 27. Continuación de ejercicio utilizando todas las funciones

```
// Usamos configuración de la IIFE
console.log("Impuestos:", configuracion.obtenerImpuestos());  [ 'Impuestos:', 0.16 ]
console.log("Moneda:", configuracion.obtenerMoneda());  [ 'Moneda:', 'USD' ]
console.log("Precio formateado:", configuracion.formatearPrecio(100));  [ 'Precio formateado:', '100 USD' ]

// Usamos el generador para crear nuevos IDs
const generador = generadorId();
console.log("Nuevo ID:", generador.next().value);  [ 'Nuevo ID:', 6 ]
console.log("Nuevo ID:", generador.next().value);  [ 'Nuevo ID:', 7 ]

// Función para agregar un nuevo producto
function agregarProducto(productos, nombre, precio, categoria) {
  const generador = generadorId();
  const nuevoId = generador.next().value;
  const nuevoProducto = { id: nuevoId, nombre, precio, categoria };
  return [...productos, nuevoProducto];
}

// Agregamos un nuevo producto
const productosActualizados = agregarProducto(productos, "Tablet", 350, "electrónica");
console.log("Productos después de la adición:", productosActualizados);  ... electrónica' }, { id: 6, nombr
```

Fuente : Elaboración propia

7. Manipulación del DOM

El Document Object Model (DOM) es una representación en forma de árbol de los elementos de un documento HTML. JavaScript puede manipular este modelo para cambiar dinámicamente el contenido, la estructura y el estilo de una página web. [16]

7.1. Selección de elementos

Para manipular elementos en el DOM, primero debemos seleccionarlos. JavaScript ofrece varios métodos para esto:

Métodos tradicionales

Tabla 3. Propiedades básicas de selección en JavaScript [16]

Método	**Descripción**
document.getElementById('miId')	Selecciona un único elemento que tiene el ID especificado.
document.getElementsByClassName('miClase')	Selecciona todos los elementos con la clase dada. Devuelve una *HTMLCollection*.

document.getElementsByTagName('p')	Selecciona todos los elementos con la etiqueta especificada. Devuelve una *HTMLCollection*.
document.querySelector('.miClase')	Selecciona el primer elemento que coincida con el selector CSS dado.
document.querySelectorAll('div.miClase')	Selecciona todos los elementos que coincidan con el selector CSS dado. Devuelve una *NodeList*.

Fuente: Elaboración propia

7.2. Modificación de contenido

Una vez seleccionados los elementos, podemos modificar su contenido y atributos.

Propiedades básicas.

Tabla 4. Propiedades básicas de modificación en JavaScript [16]

Código / Método	**Descripción**
elemento.textContent = 'Nuevo texto';	Cambia el contenido de texto del elemento, sin interpretar HTML.
elemento.innerHTML = 'Texto en negrita';	Cambia el contenido HTML del elemento, permitiendo etiquetas HTML.
elemento.setAttribute('atributo', 'valor');	Establece o actualiza el valor de un atributo específico del elemento.
elemento.getAttribute('atributo');	Obtiene el valor actual de un atributo del elemento.
elemento.removeAttribute('atributo');	Elimina un atributo del elemento.

elemento.id = 'nuevoId';	Cambia directamente el atributo id del elemento.
elemento.className = 'nuevaClase';	Cambia directamente el atributo class del elemento (reemplaza todas las clases).
elemento.href = 'nueva-url.html';	Modifica directamente el atributo href de un enlace <a>.

Fuente: Elaboración propia

7.3. Classlist para manipulación de clases

Tabla 5. Manipulación de clases en JavaScript [16]

Código / Método	**Descripción**
elemento.className = 'nuevaClase';	Establece las clases del elemento, sobrescribiendo cualquier clase existente.
elemento.classList.add('nueva-clase');	Añade una clase al atributo class del elemento sin afectar otras clases.
elemento.classList.remove('clase-existente');	Elimina una clase específica del atributo class del elemento.
elemento.classList.toggle('clase-activa');	Añade la clase si no existe; la elimina si ya está presente.
elemento.classList.contains('clase-especial')	Verifica si el elemento tiene una clase específica. Devuelve true o false.

Fuente: Elaboración propia

7.4. Creación y eliminación de elementos

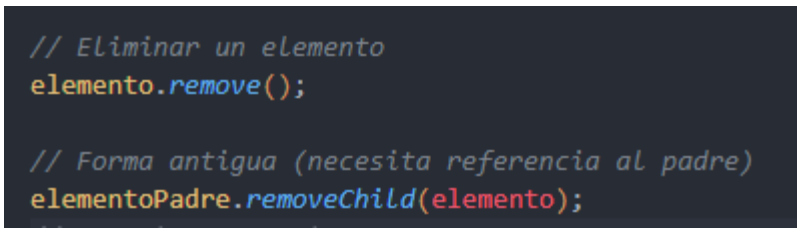
Podemos crear nuevos elementos y añadirlos al DOM, o eliminar elementos existentes.

Figura 28. Creación de elementos

```
// Crear un nuevo elemento
const nuevoElemento = document.createElement('div');

// Añadir contenido y atributos
nuevoElemento.textContent = 'Hola Mundo';
nuevoElemento.className = 'mi-clase';

// Añadir al DOM
document.body.appendChild(nuevoElemento); // Al final del body
elementoPadre.insertBefore(nuevoElemento, elementoReferencia); // Antes de elementoReferencia
elementoPadre.prepend(nuevoElemento); // Como primer hijo
elementoPadre.append(nuevoElemento); // Como último hijo (similar a appendChild)
```

Fuente: Elaboración propia

Figura 29. Eliminación de elementos



```
// Eliminar un elemento
elemento.remove();

// Forma antigua (necesita referencia al padre)
elementoPadre.removeChild(elemento);
```

Fuente: Elaboración propia

7.5. Manipulación de estilos

Podemos modificar los estilos CSS de los elementos directamente desde JavaScript.

Figura 30. Modificación directa de estilos

```
// Modificar estilos directamente (camelCase para propiedades compuestas)
elemento.style.backgroundColor = 'red';
elemento.style.fontSize = '20px';
elemento.style.display = 'none'; // Ocultar elemento

// Obtener estilos calculados (solo lectura)
const estilos = window.getComputedStyle(elemento);
const colorFondo = estilos.getPropertyValue('background-color');
```

Fuente: Elaboración propia

7.6. Manipulación de hojas de estilo

También podemos manipular las reglas CSS directamente:

Figura 31. Manipulación de las reglas CSS

```
// Añadir una regla CSS dinámicamente
const hojaEstilo = document.createElement('style');
hojaEstilo.innerHTML = `
    .mi-clase {
        color: red;
        font-weight: bold;
    }
`;
document.head.appendChild(hojaEstilo);
```

Fuente: Elaboración propia

8. Eventos y manejadores

8.1. Eventos del DOM

Los eventos son acciones que ocurren en el documento, como clicks, movimientos del ratón, pulsaciones de teclas, etc. Podemos escuchar estos eventos y ejecutar código en respuesta.

Tipos comunes de eventos

Tabla 6. Clasificación de eventos en JavaScript [16]

Categoría	**Evento**	**Descripción**
Ratón (mouse)	**elemento.addEventListener('click', e => { console.log('click'); });**	Se dispara cuando el usuario hace clic en un elemento.
	elemento.addEventListener('dblclick', e => { console.log('dblclick'); });	Se activa al hacer doble clic sobre un elemento.

	elemento.addEventListener('mouseenter', e => { console.log('mouseenter'); });	Ocurre cuando el puntero entra en el área del elemento.
	elemento.addEventListener('mouseleave', e => { console.log('mouseleave'); });	Ocurre cuando el puntero sale del área del elemento.
	elemento.addEventListener('mousemove', e => { console.log('mousemove'); });	Se ejecuta cada vez que el puntero del ratón se mueve sobre un elemento.
Teclado	**elemento.addEventListener('keydown', e => { console.log('keydown'); });**	Se activa cuando una tecla se presiona (antes de soltarla).
	elemento.addEventListener('keyup', e => { console.log('keyup'); });	Se activa cuando una tecla es soltada.
	elemento.addEventListener('keypress', e => { console.log('keypress'); });	Se activa cuando se presiona una tecla que genera un carácter (obsoleto).
Formulario	**elemento.addEventListener('submit', e => { e.preventDefault(); console.log('submit'); });**	Se dispara al enviar un formulario.
	elemento.addEventListener('change', e => { console.log('change'); });	Se activa cuando el valor de un input cambia y pierde el foco.

	elemento.addEventListener('input', e => { console.log('input'); });	Se ejecuta cada vez que el valor de un input cambia (en tiempo real).
	elemento.addEventListener('focus', e => { console.log('focus'); });	Se activa cuando un elemento recibe el foco.
	elemento.addEventListener('blur', e => { console.log('blur'); });	Se activa cuando un elemento pierde el foco.
Ventana (window)	**window.addEventListener('load', e => { console.log('load'); });**	Se dispara cuando la página ha terminado de cargar por completo.
	window.addEventListener('resize', e => { console.log('resize'); });	Se ejecuta cuando se cambia el tamaño de la ventana del navegador.
	window.addEventListener('scroll', e => { console.log('scroll'); });	Se activa al desplazarse la página o un elemento con scroll.
Touch (pantalla táctil)	**elemento.addEventListener('touchstart', e => { console.log('touchstart'); });**	Se dispara cuando el usuario toca la pantalla.

	elemento.addEventListener('touchmove', e => { console.log('touchmove'); });	Se activa cuando el dedo del usuario se mueve por la pantalla.
	elemento.addEventListener('touchend', e => { console.log('touchend'); });	Se dispara cuando el usuario deja de tocar la pantalla.
Multimedia	**elemento.addEventListener('play', e => { console.log('play'); });**	Se ejecuta cuando un medio (audio/video) comienza a reproducirse.
	elemento.addEventListener('pause', e => { console.log('pause'); });	Se activa cuando se pausa la reproducción de un medio.
	elemento.addEventListener('ended', e => { console.log('ended'); });	Se dispara cuando la reproducción del medio ha finalizado.

Fuente: Elaboración propia

8.2. Manejadores de eventos

Hay varias formas de asignar manejadores de eventos:

Figura 32. Asignación de manejadores

```
// 1. Atributo HTML
<button onclick="alert('Hola')">Click me</button>

// 2. Propiedad del elemento
elemento.onclick = function() {
  alert('Hola');
};

// 3. addEventListener (recomendado)
elemento.addEventListener('click', function(evento) {
  console.log('Click registrado', evento);
});
```

Fuente Elaboración propia

Nota: addEventListener es el método preferido porque permite agregar múltiples escuchadores para el mismo evento y ofrece más control (como el uso de captura/burbujeo).

9. JSON y Almacenamiento Local

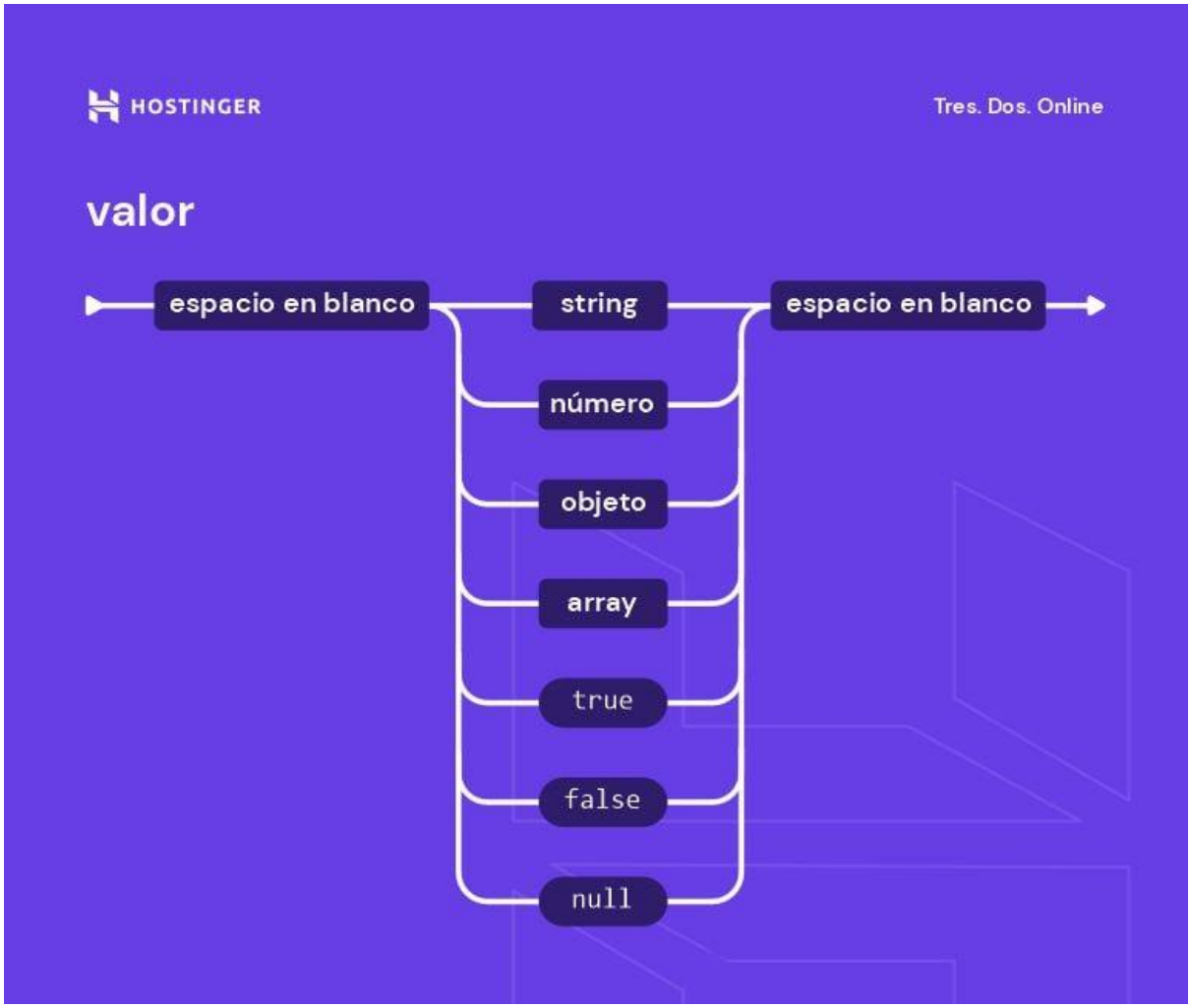
9.1. JSON

Se trata de un formato para guardar e intercambiar información que cualquier persona pueda leer. Los archivos json contienen solo texto y usan la extensión **.json**.
Hay dos elementos centrales en un objeto JSON: claves (Keys) y valores (Values).

- Las **Keys** deben ser cadenas de caracteres (strings). Como su nombre en español lo indica, estas contienen una secuencia de caracteres rodeados de comillas.
- Los **Values** son un tipo de datos JSON válido. Puede tener la forma de un arreglo (array), objeto, cadena (string), booleano, número o nulo.

Un objeto JSON comienza y termina con llaves**{}**. Puede tener dos o más pares de **claves/valor** dentro, con una **coma** para separarlos. Así mismo, cada key es seguida por **dos puntos** para distinguirla del valor. [17]

Figura 33. Estructura de JSON



Fuente. https://www.hostinger.com/es/tutoriales/que-es-json

9.2. ¿Qué es el almacenamiento local?

Almacenamiento local (o *localStorage*) es una funcionalidad del navegador web que permite guardar datos en el navegador del usuario, de forma persistente (aunque se cierre el navegador).

Se guarda en pares clave/valor como en un diccionario, y los valores deben estar en formato string. [18]

Tabla 7. Características del localStorage y sessionStorage

Característica	localStorage	sessionStorage
Persistencia	Persistente. No se borra al cerrar el navegador.	Temporal. Se borra al cerrar la pestaña.
Ámbito	Disponible para todas las pestañas del mismo sitio.	Solo para la pestaña en la que se creó.
Tamaño aproximado	5–10 MB	5 MB
Uso típico	Guardar preferencias del usuario, sesiones largas.	Datos temporales, como formularios o navegación de una sola sesión.

Fuente: elaboración propia

Figura 34. Manejo de localStorage

```
<script>
  function guardarLocal() {
    const usuario = { nombre: "Ana", edad: 25 };
    console.log("Guardando en localStorage:", usuario);
    localStorage.setItem("usuario", JSON.stringify(usuario));
    alert("Datos guardados en localStorage");
  }

  function leerLocal() {
    const datos = localStorage.getItem("usuario");
    console.log("Leyendo de localStorage:", datos);
    if (datos) {
      const usuario = JSON.parse(datos);
      document.getElementById("resultadoLocal").textContent = `Nombre: ${usuario.nombre}, Edad: ${usuario.edad}`;
    } else {
      document.getElementById("resultadoLocal").textContent = "No hay datos en localStorage";
    }
  }

  function borrarLocal() {
    console.log("Eliminando usuario de localStorage");
    localStorage.removeItem("usuario");
    document.getElementById("resultadoLocal").textContent = "Datos borrados de localStorage";
  }
```

Fuente: Elaboración propia

Figura 35. Manejo de sessionStorage

```
  function guardarSession() {
    const sesion = { tema: "oscuro", idioma: "es" };
    console.log("Guardando en sessionStorage:", sesion);
    debugger;
    sessionStorage.setItem("config", JSON.stringify(sesion));
    alert("Datos guardados en sessionStorage");
  }

  function leerSession() {
    const datos = sessionStorage.getItem("config");
    console.log("Leyendo de sessionStorage:", datos);
    debugger;
    if (datos) {
      const config = JSON.parse(datos);
      document.getElementById("resultadoSession").textContent = `Tema: ${config.tema}, Idioma: ${config.idioma}`;
    } else {
      document.getElementById("resultadoSession").textContent = "No hay datos en sessionStorage";
    }
  }

  function borrarSession() {
    console.log("Eliminando config de sessionStorage");
    debugger;
    sessionStorage.removeItem("config");
    document.getElementById("resultadoSession").textContent = "Datos borrados de sessionStorage";
  }
</script>
```

Fuente: Elaboración propia

10. Depuración y uso de consola

10.1. ¿Qué es la consola del navegador?

La consola del navegador es parte de las herramientas de desarrollo que ofrecen Chrome, Firefox, Edge, etc. Te permite:

- Ver errores del código.
- Mostrar mensajes (console.log()).
- Probar código directamente.
- Inspeccionar el contenido de localStorage y sessionStorage.

Figura 36. Manejo de Debugger

```
function guardarSession() {
  const sesion = { tema: "oscuro", idioma: "es" };
  console.log("Guardando en sessionStorage:", sesion);
  debugger;
  sessionStorage.setItem("config", JSON.stringify(sesion));
  alert("Datos guardados en sessionStorage");
}

function leerSession() {
  const datos = sessionStorage.getItem("config");
  console.log("Leyendo de sessionStorage:", datos);
  debugger;
  if (datos) {
    const config = JSON.parse(datos);
    document.getElementById("resultadoSession").textContent = `Tema: ${config.tema}, Idioma: ${config.idioma}`;
  } else {
    document.getElementById("resultadoSession").textContent = "No hay datos en sessionStorage";
  }
}

function borrarSession() {
  console.log("Eliminando config de sessionStorage");
  debugger;
  sessionStorage.removeItem("config");
  document.getElementById("resultadoSession").textContent = "Datos borrados de sessionStorage";
}
</script>
```

Fuente: Elaboración propia

CONCLUSIÓN

El aprendizaje de JavaScript básico constituye una etapa fundamental en la formación de cualquier desarrollador web, ya que este lenguaje es el principal responsable de dotar de interactividad y dinamismo a los sitios web. A lo largo del estudio de sus conceptos clave, se ha evidenciado su versatilidad y su estrecha integración con los navegadores, lo cual lo convierte en una herramienta indispensable en el entorno del desarrollo frontend.

Las funciones representan uno de los pilares del lenguaje, ya que permiten encapsular bloques de código reutilizables. Entender las diferencias entre funciones declaradas y expresadas, así como el uso de parámetros y valores de retorno, proporciona al desarrollador mayor control y modularidad en su trabajo.

Uno de los aspectos más relevantes del uso práctico de JavaScript es la manipulación del DOM. Gracias a la capacidad de seleccionar elementos del documento, modificar su contenido y responder a eventos, JavaScript hace posible transformar la interfaz de usuario en tiempo real. El uso de métodos como querySelector, innerHTML y classList habilita un control fino sobre la estructura visual del sitio.

La gestión de eventos, a través de métodos como addEventListener o atributos como onclick, permite que las aplicaciones reaccionen a las acciones del usuario, como clics, movimientos del mouse o entradas de teclado. Esta capacidad reactiva es lo que define en gran medida la experiencia de usuario moderna y dinámica en la web.

El trabajo con JSON como formato ligero de intercambio de datos y el uso de almacenamiento local (localStorage y sessionStorage) introducen al desarrollador en el mundo de la persistencia de datos en el navegador. Estas herramientas permiten guardar información del usuario, mantener el estado de la aplicación entre sesiones y mejorar la funcionalidad sin requerir conexiones constantes a un servidor.

Finalmente, el uso de herramientas de depuración como console.log, console.error, y la instrucción debugger, forman parte de las buenas prácticas en el desarrollo con JavaScript. Estas técnicas facilitan la identificación y corrección de errores, lo que es crucial para garantizar el correcto funcionamiento y mantenimiento del código.

En conjunto, los conocimientos adquiridos sobre JavaScript básico no solo proporcionan una sólida base técnica, sino que también permiten comenzar a construir aplicaciones web interactivas, estructuradas y funcionales. Dominar estos fundamentos es un paso esencial para avanzar hacia conceptos más complejos como programación orientada a objetos, asincronía, frameworks modernos (como React o Vue), y el desarrollo completo del lado del cliente.

BIBLIOGRAFÍA

[1]

[2]

[3] N. del Kodev, “Operadores Básicos en JavaScript: Guía Introductoria,” *Mi blog personal*. [En línea]. Disponible: https://nelkodev.com/javascript/introduccion-a-los-operadores-basicos-en-javascript/#¿Que_son_los_Operadores_Basicos. [Accedido: 10-may-2025].

[4] GCFLearnFree.org, “Operadores aritméticos,” *GCFGlobal*. [En línea]. Disponible: https://edu.gcfglobal.org/es/conceptos-basicos-de-programacion/operadores-aritmeticos/1/. [Accedido: 11-may-2025].

[5] TiposDe.net, “Tipos de operadores lógicos en programación,” *TiposDe.net*. [En línea]. Disponible: https://tiposde.net/tipos-de-operadores-logicos-en-programacion/. [Accedido: 11-may-2025].

[6] LenguajeJS, “Operadores de comparación,” *LenguajeJS.com*. [En línea]. Disponible: https://lenguajejs.com/javascript/operadores/comparacion/. [Accedido: 12-may-2025].

[7] BigCode, “Estructuras de Control en JavaScript: Condicionales,” *BigCode*. [En línea]. Disponible: https://bigcode.es/estructuras-de-control-en-javascript-condicionales/. [Accedido: 12-may-2025].

[8] R. D. Hernandez, “JavaScript if-Else y If-Then: Sentencias condicionales en JS,” *freeCodeCamp en español*, 20 de abril de 2022. [En línea]. Disponible: https://www.freecodecamp.org/espanol/news/javascript-if-else-y-if-then-sentencias-condicionales-en-js/. [Accedido: 13-may-2025].

[9] "for - JavaScript | MDN," Mozilla Developer Network, 2024. [Online]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/for. [Accessed: 13-May-2025].

[10] D. Flanagan, JavaScript: The Definitive Guide, 7th ed., Sebastopol, CA, USA: O'Reilly Media, 2020. [Online]. Available: https://www.oreilly.com/library/view/javascript-the-definitive/9781491952016/

[11] "Loops: while and for," JavaScript.info, 2023. [Online]. Available: https://javascript.info/while-for. [Accessed: 13-May-2025].

[12] "Functions - JavaScript," Mozilla Developer Network, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions. [Accessed: 12-May-2025].

[13] M. Haverbeke, Eloquent JavaScript: A Modern Introduction to Programming, 3rd ed. No Starch Press, 2018. [Online]. Available: https://eloquentjavascript.net/

[14] N. C. Zakas, Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers. San Francisco, CA: No Starch Press, 2016. [Accessed: 13-May-2025].

[15] K. Simpson, You Don't Know JS Yet: Get Started. Sebastopol, CA: O'Reilly Media, 2020. [Accessed: 13-May-2025].

[16] OpenAI, *ChatGPT* [Modelo de lenguaje AI], OpenAI, San Francisco, CA, USA. Disponible en: https://chat.openai.com. [Accedido: may-2025].

[17] D. A and D. A, “¿Qué es JSON?,” ES Tutoriales, Jan. 10, 2023. https://www.hostinger.com/es/tutoriales/que-es-json [Accessed: 13-May-2025].

[18] I. Kantor, “LocalStorage, SessionStorage.” https://es.javascript.info/localstorage [Accessed: 13-May-2025].