Tabla de contenidos

1. Tipos de Datos

2. Variables

3. Operaciones y Expresiones

4. Sentencias Condicionales

5. De la Lógica Visual al Código: Diagramas de Flujo

1. Tipos de Datos

JavaScript es un lenguaje de tipado dinámico, lo que significa que no es necesario declarar el tipo de dato que una variable contendrá. El tipo se determina automáticamente en tiempo de ejecución.

Tipos de datos Primitivos (Simples): Son los datos más básicos y fundamentales.

- **String:** Una secuencia de caracteres utilizada para representar texto. Se define con comillas simples (`' '`), dobles (`" "`) o plantillas literales (`` ` ``).

```
let saludo = "Hola, mundo";
let nombre = 'Ada Lovelace';
```

- **Number:** Representa tanto números enteros como de punto flotante.

```
let cantidad = 100;
let precio = 99.95;
let temperatura = -10;
```

- **Boolean:** Representa un valor lógico: `true` (verdadero) o `false` (falso). Es fundamental para la toma de decisiones en el código.

```
let esMayorDeEdad = true;
let tieneDescuento = false;
```

- **Otros primitivos:** `undefined` (una variable declarada sin valor), `null` (ausencia intencional de un valor), `Symbol` y `BigInt`.

Tipos de datos Complejos: Se utilizan para agrupar y organizar valores.

- **Object:** Una colección de pares clave-valor. Es el tipo de dato más fundamental en JavaScript, usado para agrupar datos y funcionalidades relacionadas.

```
let coche = {
    marca: "Toyota",
    modelo: "Corolla",
    año: 2022,
    estaEncendido: false,
    encender: function() {
        this.estaEncendido = true;
    }
};
console.log(coche.marca); // "Toyota"
```

- **Array (Arreglo):** Un tipo especial de objeto, optimizado para almacenar una colección ordenada de valores. Se accede a sus elementos mediante un índice numérico que comienza en 0.

```
let colores = ["rojo", "verde", "azul"];
console.log(colores[0]); // "rojo"
console.log(colores.length); // 3
```

2. Variables

¿Qué es una variable?

Ahora que conocemos los tipos de datos, podemos hablar de dónde almacenarlos. Una variable es un contenedor con nombre que se utiliza para guardar un valor. Es un espacio de memoria reservado al que le asignamos una etiqueta (el nombre de la variable) para poder referenciar y utilizar su contenido a lo-largo del programa.

Analogía: Imagina una caja de almacenamiento. Puedes ponerle una etiqueta, como "Juguetes" o "Libros", y dentro guardar los objetos correspondientes. En programación, la caja es el espacio en memoria, la etiqueta es el nombre de la variable y los objetos dentro son los datos o el valor. Puedes cambiar el contenido de la caja cuando quieras; de ahí el término "variable".

Consejos para nombrar una variable: El nombre de una variable (identificador) debe ser descriptivo y seguir ciertas reglas y convenciones:

- **Reglas (Obligatorias):**
  - Deben comenzar con una letra, un guion bajo (`_`) o el símbolo de dólar (`$`).
  - Después del primer carácter, pueden contener letras, números, guiones bajos o símbolos de dólar.
  - Son sensibles a mayúsculas y minúsculas (`nombre` y `Nombre` son dos variables diferentes).
  - No pueden ser palabras reservadas del lenguaje (como `if`, `else`, `let`, `function`, etc.).
- **Convenciones (Buenas Prácticas):**
  - Utilizar **camelCase** para nombres de variables compuestos por varias palabras (ej. `nombreCompletoDelUsuario`).
  - Los nombres deben ser descriptivos y reflejar el dato que contienen (ej. usar `edadUsuario` en lugar de `e` o `data1`).
  - Evitar empezar con guion bajo (`_`) o dólar (`$`) a menos que se siga una convención específica (común en algunas librerías o para indicar variables "privadas").

Declaración de una variable: Declarar una variable es el acto de crearla o registrar su identificador en el ámbito (scope) actual. En JavaScript moderno se utilizan las palabras clave `let` y `const`.

```
let edad; // Declaramos una variable llamada 'edad'
let nombre, apellido, direccion; // Podemos declarar múltiples variables a la vez
```

Inicialización de una variable: Inicializar una variable es el acto de asignarle un valor por primera vez. Se realiza con el operador de asignación (`=`).

```
let edad; // Declaración
edad = 30; // Inicialización

// También podemos declarar e inicializar en la misma línea
let ciudad = "Londres";
```

Si una variable es declarada pero no inicializada, su valor por defecto es `undefined`.

Variables Constantes (`const`): Una constante es un tipo de variable cuyo valor no puede ser reasignado una vez que ha sido inicializado. Se declaran con la palabra clave `const` y **deben** ser inicializadas en el momento de su declaración.

```
const PI = 3.14159;
// PI = 3.14; // Esto generaría un error: TypeError: Assignment to constant variable.

const CONFIGURACION = {
  tema: "oscuro",
  idioma: "es"
};
// Ojo: Si la constante es un objeto o un array, su contenido interno sí puede cambiar.
CONFIGURACION.tema = "claro"; // Esto es válido.
console.log(CONFIGURACION.tema); // "claro"
```

Buena práctica: Usa `const` por defecto para todas tus variables. Cámbiala a `let` solo si sabes que necesitarás reasignar su valor. Esto hace el código más predecible y menos propenso a errores.

Scope (Ámbito) de las Variables

El scope determina la accesibilidad o visibilidad de las variables en diferentes partes del código.

- **Scope Global:** Una variable declarada fuera de cualquier función o bloque (`{}`) es global. Es accesible desde cualquier parte del código. Se debe evitar en lo posible la creación de variables globales para prevenir conflictos.
- **Scope de Función:** Variables declaradas con `var` dentro de una función solo son accesibles dentro de esa función.

- **Scope de Bloque:** Variables declaradas con `let` y `const` dentro de un bloque (delimitado por llaves `{...}`, como en un `if` o un `for`) solo son accesibles dentro de ese bloque. Esta es una de las principales mejoras sobre `var`.

```
let variableGlobal = "Soy global";

function miFuncion() {
    let variableDeFuncion = "Estoy dentro de la función";
    console.log(variableGlobal); // "Soy global"

    if (true) {
        let variableDeBloque = "Estoy dentro del bloque if";
        console.log(variableDeFuncion); // "Estoy dentro de la función"
    }
    // console.log(variableDeBloque); // Error: variableDeBloque is not defined
}
miFuncion();
// console.log(variableDeFuncion); // Error: variableDeFuncion is not defined
```

3. Operaciones y Expresiones

Una **expresión** es cualquier fragmento de código que produce un valor. Los **operadores** son símbolos que realizan operaciones sobre uno o más operandos (valores o variables).

Operadores Aritméticos:

- `+` (Suma)
- `-` (Resta)
- `*` (Multiplicación)
- `/` (División)
- `%` (Módulo o Resto): Devuelve el resto de una división. `10 % 3` es `1`.
- `**` (Exponenciación): `2 ** 3` es `8`.

Precedencia de Operadores: Determina el orden en que se evalúan las operaciones en una expresión compleja. Es similar al orden matemático (PEMDAS).

1. Paréntesis `()`
2. Exponenciación `**`
3. Multiplicación `*`, División `/`, Módulo `%` (de izquierda a derecha)
4. Suma `+`, Resta `-` (de izquierda a derecha)

```
let resultado = 5 + 3 * 2; // 5 + 6 = 11
let resultadoConParentesis = (5 + 3) * 2; // 8 * 2 = 16
```

Operadores de Incremento y Decremento: Añaden o restan 1 a una variable numérica.

- `++` (Incremento): `x++` (post-incremento) o `++x` (pre-incremento).
- `--` (Decremento): `x--` (post-decremento) o `--x` (pre-decremento).

```
let contador = 10;
contador++; // contador ahora es 11
console.log(contador);

let y = 5;
let z = ++y; // y se incrementa a 6, y luego z se asigna a 6.
console.log(y, z); // 6, 6

let a = 5;
let b = a++; // b se asigna a 5, y luego a se incrementa a 6.
console.log(a, b); // 6, 5
```

Operadores de Comparación: Evalúan una comparación y devuelven un valor booleano (`true` o `false`).

- `==` (Igualdad laxa): Compara si dos valores son iguales, realizando conversión de tipo si es necesario. **(Evitar su uso)**.
- `===` (Igualdad estricta): Compara si dos valores son iguales y del mismo tipo. **(Recomendado)**.
- `!=` (Desigualdad laxa).
- `!==` (Desigualdad estricta).
- `>` (Mayor que), `<` (Menor que).
- `>=` (Mayor o igual que), `<=` (Menor o igual que).

```
console.log(5 == "5");   // true (convierte el string a número)
console.log(5 === "5");  // false (diferente tipo)
console.log(10 > 5);     // true
console.log(10 !== 10);  // false
```

Cadenas de Caracteres (Strings)

Creación y uso de comillas: Puedes usar comillas simples o dobles indistintamente, pero sé consistente. La elección a menudo depende de si la cadena contiene comillas.

```
let frase1 = "Ella dijo: 'Hola'";
let frase2 = 'El libro se llama "JavaScript Definitivo"';
```

Concatenación: Unir dos o más strings. Se hace con el operador `+`.

```
let nombre = "Grace";
let apellido = "Hopper";
let nombreCompleto = nombre + " " + apellido; // "Grace Hopper"
```

Plantillas Literales (Template Literals): Introducidas en ES6, son la forma moderna y más legible de trabajar con strings. Usan comillas invertidas (`` ` ``) y permiten incrustar expresiones dentro de la cadena con la sintaxis `${expresion}`.

```
let producto = "Café";
let precio = 4.5;
let cantidad = 2;

let mensaje = `Usted compró ${cantidad} unidades de ${producto}. El total es $${precio * cantidad}.`;
console.log(mensaje); // "Usted compró 2 unidades de Café. El total es $9."
```

4. Sentencias Condicionales

Las sentencias condicionales, también conocidas como estructuras de control de flujo, nos permiten dirigir el "camino" que toma nuestro programa al ejecutarse. En lugar de ejecutar todas las líneas de código en orden, podemos establecer reglas para que ciertos bloques de código solo se ejecuten si se cumplen determinadas condiciones.

Analogía: Piensa en las instrucciones de una receta. Una instrucción podría ser "Si no tienes sal, añade una pizca de salsa de soja". No siempre realizarás esa acción, solo bajo la condición específica de no tener sal. Las sentencias condicionales en programación funcionan de la misma manera.

2.1 Expresiones y Operaciones Lógicas

La base de toda condición es una expresión que se evalúa como `true` (verdadero) o `false` (falso). Para construir estas expresiones, utilizamos operadores de comparación (vistos anteriormente) y operadores lógicos.

- `&&` (AND Lógico): Devuelve `true` solo si **ambos** operandos son verdaderos.
  - `(edad > 18) && (tieneLicencia === true)`: Verdadero solo si la edad es mayor a 18 Y tiene licencia.
- `||` (OR Lógico): Devuelve `true` si **al menos uno** de los operandos es verdadero.
  - `(esFinDeSemana || esFeriado)`: Verdadero si es fin de semana O si es feriado.
- `!` (NOT Lógico): Invierte el valor booleano de un operando. `!true` es `false`.
  - `!estaLloviendo`: Verdadero si la variable `estaLloviendo` es falsa.

```
let esAdulto = true;
let tieneLicencia = false;

console.log(esAdulto && tieneLicencia); // false
console.log(esAdulto || tieneLicencia); // true
console.log(!tieneLicencia);             // true
```

La Sentencia `if` (Condición Simple)

Es la estructura condicional más básica. Ejecuta un bloque de código **únicamente si la condición especificada es verdadera**. Si es falsa, el bloque de código se ignora y el programa continúa con la siguiente instrucción después del `if`.

Sintaxis:

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

Ejemplo: Mostrar un mensaje de bienvenida solo si el usuario ha iniciado sesión.

```
let usuarioEstaLogueado = true;

if (usuarioEstaLogueado) {
    console.log("¡Bienvenido de nuevo a la plataforma!");
}

console.log("Fin del programa."); // Esta línea se ejecuta siempre
```

Si `usuarioEstaLogueado` fuera `false`, solo se vería en la consola "Fin del programa.".

La Sentencia `if...else` (Condición Binaria)

Esta estructura nos permite definir dos caminos: uno a tomar si la condición es verdadera (`if`) y otro si es falsa (`else`). Es ideal para situaciones de "esto o aquello".

Sintaxis:

```
if (condicion) {
    // Código a ejecutar si la condición es true
} else {
    // Código a ejecutar si la condición es false
}
```

Ejemplo: Determinar si una persona puede votar (basado en el diagrama de flujo).

```
// 1. Obtenemos la edad del usuario
let edadUsuario = 17;

// 2. Evaluamos la condición con la estructura if...else
if (edadUsuario >= 18) {
    // 3a. Este bloque se ejecuta si la condición es verdadera
    console.log("La persona tiene " + edadUsuario + " años. Puede votar.");
} else {
    // 3b. Este bloque se ejecuta si la condición es falsa
    console.log("La persona tiene " + edadUsuario + " años. Aún no puede votar.");
}
```

La Sentencia `if...else if...else` (Condiciones Múltiples)

Cuando necesitamos evaluar más de dos posibilidades, podemos encadenar múltiples condiciones. El programa evalúa las condiciones en orden: en cuanto encuentra una que es verdadera, ejecuta su bloque de código y omite el resto de la cadena. Si ninguna es verdadera, se ejecuta el bloque `else` final (si existe).

Sintaxis:

```
if (condicion1) {
    // Se ejecuta si condicion1 es true
} else if (condicion2) {
    // Se ejecuta si condicion1 es false Y condicion2 es true
} else if (condicion3) {
    // Se ejecuta si las anteriores son false Y condicion3 es true
} else {
    // Se ejecuta si NINGUNA de las condiciones anteriores es true
}
```

Ejemplo: Clasificar a una persona por su edad.

```
let edad = 70;

if (edad < 0) {
    console.log("Edad no válida.");
} else if (edad < 18) {
    console.log("Es menor de edad.");
} else if (edad <= 65) {
    console.log("Es un adulto.");
} else {
    console.log("Es un adulto mayor.");
}
```

La Sentencia `switch` (Selección Múltiple)

La sentencia `switch` es una alternativa a un `if...else if...else` largo. Es particularmente útil cuando se quiere comparar el valor de **una sola variable** contra una lista de posibles valores (casos).

Analogía: Piensa en el `switch` como un menú de opciones. La variable es tu elección, y cada `case` es una de las opciones del menú. El `default` es lo que sucede si tu elección no está en el menú.

Sintaxis:

```
switch (expresion) {
    case valor1:
        // Código a ejecutar si expresion === valor1
        break; // La palabra 'break' es crucial
    case valor2:
        // Código a ejecutar si expresion === valor2
        break;
    default:
        // Código a ejecutar si no coincide con ningún case
}
```

- **`case`**: Define un valor específico con el que se comparará la `expresion`. La comparación es estricta (`===`).
- **`break`**: Termina la ejecución del `switch`. Si se omite, el código continuará ejecutándose en el siguiente `case` (lo que se conoce como "fall-through"), lo cual suele ser un error.
- **`default`**: (Opcional) Se ejecuta si ningún `case` coincide con el valor de la `expresion`.

Ejemplo: Mostrar un mensaje según el nivel de suscripción del usuario.

```
let nivelSuscripcion = "premium";
let mensajeUsuario;

switch (nivelSuscripcion) {
    case "free":
        mensajeUsuario = "Tienes acceso a nuestros cursos gratuitos.";
        break;
    case "basic":
        mensajeUsuario = "Tienes acceso a los cursos gratuitos y a los cursos básicos.";
        break;
    case "premium":
        mensajeUsuario = "¡Acceso ilimitado! Tienes acceso a todos los cursos y contenido exclusivo.";
        break;
    default:
        mensajeUsuario = "Nivel de suscripción no reconocido. Por favor, contacta a soporte.";
        break;
}

console.log(mensajeUsuario);
```

Manejando Condiciones de Borde (Edge Cases)

Las condiciones de borde son entradas que están en los límites de lo que nuestro código espera y que a menudo pueden causar errores si no se manejan. Es una buena práctica anticipar y validar las entradas.

Problema: ¿Qué pasa en nuestro programa de votación si el usuario ingresa texto en lugar de un número, como "veinte", o un valor sin sentido como -5?

Debemos hacer nuestro código más robusto para manejar estas situaciones, usualmente con una condición `if` al principio.

```
// Versión mejorada del código de votación
let entradaUsuario = "dieciocho"; // Simulemos una entrada que podría ser un string no numérico
let edadNum = Number(entradaUsuario); // Intentamos convertir la entrada a número. Si falla, el resultado es NaN.

// Primero, validamos la entrada para manejar los casos de borde.
if (isNaN(edadNum) || edadNum < 0) {
    console.log("Entrada no válida. Por favor, ingrese una edad como un número positivo.");
} else if (edadNum >= 18) {
    console.log("La persona tiene " + edadNum + " años. Puede votar.");
} else { // Si es un número válido pero menor que 18
    console.log("La persona tiene " + edadNum + " años. No puede votar.");
}

// isNaN() es una función que devuelve true si el valor NO es un número (Not-a-Number).
```

Este enfoque de pensar en posibles fallos y proteger el código contra ellos es una práctica esencial en el desarrollo de software profesional.

5. De la Lógica Visual al Código: Diagramas de Flujo

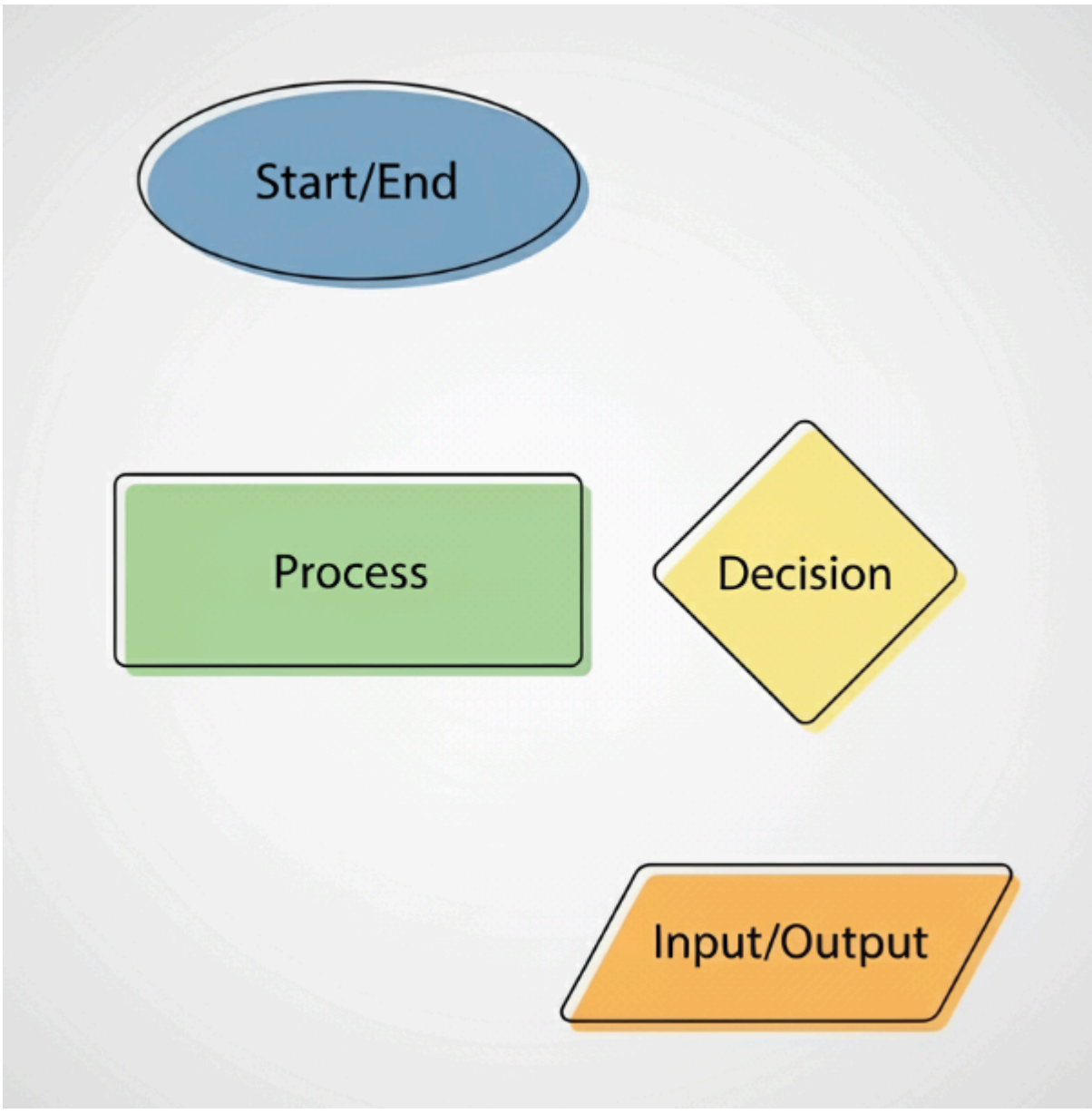
A menudo, antes de escribir código, es útil esbozar la lógica de nuestro programa de forma visual. Una de las herramientas más estándar para esto es el **diagrama de flujo**.

¿Qué es un Diagrama de Flujo?

Un diagrama de flujo es una representación gráfica de un algoritmo o proceso. Utiliza un conjunto de símbolos estandarizados para representar diferentes tipos de instrucciones y el flujo de control que las conecta. Esbozar un diagrama de flujo nos ayuda a planificar y visualizar la lógica condicional antes de escribir una sola línea de código.

Símbolos Comunes:

- **Óvalo:** Inicio o Fin del algoritmo.
- **Rectángulo:** Proceso (una operación o asignación).
- **Rombo:** Decisión (una condición `if` que tiene dos salidas: Sí/No o Verdadero/Falso).
- **Paralelogramo:** Entrada o Salida de datos.
- **Flechas:** Indican la dirección del flujo.



Implementando un Código a Partir de un Diagrama de Flujo

Veamos cómo traducir un problema simple, representado en un diagrama de flujo, a código JavaScript.

Problema: Un sistema debe determinar si un usuario puede acceder a un contenido para mayores de edad.

1. Esbozando el Diagrama de Flujo:

El diagrama tendría la siguiente secuencia:

1. **Inicio** (Óvalo).
2. **Entrada:** "Obtener la edad del usuario" (Paralelogramo).
3. **Decisión:** "¿Es la edad >= 18?" (Rombo).
    - Si la respuesta es **Sí**, el flujo va a un Proceso.
    - Si la respuesta es **No**, el flujo va a otro Proceso.

4. **Proceso (Sí):** "Mostrar mensaje: 'Acceso Permitido'" (Rectángulo).
5. **Proceso (No):** "Mostrar mensaje: 'Acceso Denegado'" (Rectángulo).
6. Ambos flujos convergen antes del final.
7. **Fin** (Óvalo).

2. Codificando la Rutina en JavaScript: La traducción del diagrama a código es directa. Cada símbolo corresponde a una parte de nuestro código.

```
// Este código es la implementación directa del diagrama de flujo anterior.

// --- PRIMER CASO DE PRUEBA ---

// El símbolo de Entrada (Paralelogramo) corresponde a obtener el dato.
let edadUsuario = 25;
console.log("Verificando edad:", edadUsuario);

// El símbolo de Decisión (Rombo) se traduce a una sentencia 'if'.
if (edadUsuario >= 18) {
    // El camino del "Sí" en el diagrama.
    // El símbolo de Proceso se traduce a una acción.
    console.log("Acceso Permitido al contenido.");
} else {
    // El camino del "No" en el diagrama.
    console.log("Acceso Denegado. Debes ser mayor de edad.");
}
// Salida esperada: Acceso Permitido al contenido.


// --- SEGUNDO CASO DE PRUEBA ---
console.log("\n--- Probando con otra edad ---");

// Reasignamos un nuevo valor a la variable para probar el otro camino.
edadUsuario = 16;
console.log("Verificando edad:", edadUsuario);

// Repetimos la misma lógica de decisión.
if (edadUsuario >= 18) {
    console.log("Acceso Permitido al contenido.");
} else {
    console.log("Acceso Denegado. Debes ser mayor de edad.");
}
// Salida esperada: Acceso Denegado. Debes ser mayor de edad.
```

Verificando el Algoritmo

El último criterio es crucial. Una vez que hemos escrito nuestro código, debemos verificar que funciona como se espera. La **consola de JavaScript** y el **inspector de elementos** del navegador son nuestras herramientas principales para esto. Al usar `console.log()` en puntos clave de nuestro código, podemos "ver" el flujo de ejecución y los valores de nuestras variables, asegurándonos de que nuestro código se comporta exactamente como lo planeamos en nuestro diagrama de flujo.