BIT | Bogotá Institute of Technology

Workshop Javascript

Andrés Felipe Serrano Barrios

Noviembre 11 de 2024

Contenido

Contexto de JavaScript

- **Creación**: JavaScript fue creado en 1995 por Brendan Eich en solo 10 días mientras trabajaba en Netscape.
- **Evolución**: Originalmente, JavaScript fue desarrollado para hacer páginas web interactivas, pero su uso ha evolucionado.
- **Características clave**:
  - Lenguaje interpretado y basado en eventos.
  - Utilizado tanto en el **frontend** como en el **backend**.
  - Compatible con la mayoría de navegadores.
- **Uso en la actualidad**:
  - Desarrollo de aplicaciones web y móviles.
  - Uso en el **backend** con frameworks como Node.js.
  - Interactividad y dinamismo en interfaces de usuario.
- **Ecosistema**:
  - Gran cantidad de librerías y frameworks (React, Angular, Vue, etc.).

¿Qué es JavaScript?

- JavaScript es un lenguaje de programación que permite añadir interactividad a las páginas web.
- Es uno de los lenguajes más utilizados en el mundo y forma parte del **stack web** junto con HTML y CSS.
- Se ejecuta en el navegador del usuario (**frontend**) y en el servidor (**backend**) con Node.js.

Dato interesante

JavaScript no está relacionado con Java. Fue nombrado así por razones de marketing.

Sintaxis Básica de JavaScript

- JavaScript es un lenguaje de **tipado débil**, lo que significa que no se especifica el tipo de datos al declarar variables.
- Palabras clave para declarar variables:
  - `var` - Uso tradicional, pero no recomendado.
  - `let` - Para variables que pueden cambiar.
  - `const` - Para variables que no cambian.

```
1   let nombre = "Andrés";
2   const edad = 30;
```

Diferencias entre `var`, `let` y `const`

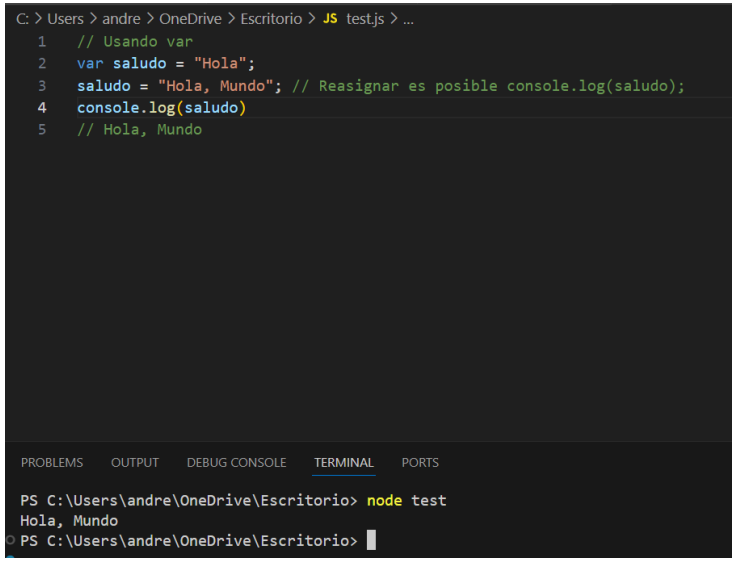
- En JavaScript, hay tres formas de declarar variables:
  - `var`
  - `let`
  - `const`
- Cada una tiene características únicas y se comporta de manera diferente.

Uso de `var`

Características de `var`

- **Alcance de función**: Accesible dentro de la función en la que se declara.
- Puede ser **redeclarada** y **reasignada**.
- Se eleva (**hoisting**) al inicio del contexto, pero su valor inicial será `undefined`.

Ejemplo de `var`:



Uso de `let`

Características de `let`

- **Alcance de bloque**: Solo es accesible dentro del bloque ({ }) en el que se declara.
- Puede ser **reasignada**, pero **no redeclarada** en el mismo ámbito.
- También se eleva, pero no se puede usar antes de declararla (`ReferenceError`).

Ejemplo de `let`:

```
1  // Usando let
2  let nombre = "Andrés";
3  nombre = "Carlos"; // Reasignar es posible
4  console.log(nombre); // Carlos
```

Uso de `const`

Características de `const`

- También tiene **alcance de bloque**.
- Debe ser inicializada al declararse y no se puede **reasignar**.
- Las propiedades internas de un objeto declarado con `const` pueden modificarse, aunque no se puede reasignar el objeto completo.

Ejemplo de `const`:

C: > Users > andre > OneDrive > Escritorio > JS test.js > ...

```
1   // Usando const
2   const saludo = "Hola";
3   saludo = "Hola, Mundo"; // Reasignar NO es posible
4   console.log(saludo)
5   // Hola, Mundo
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - Esc

```
PS C:\Users\andre\OneDrive\Escritorio> node test
C:\Users\andre\OneDrive\Escritorio\test.js:3
saludo = "Hola, Mundo"; // Reasignar es posible console.log(saludo);
       ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\andre\OneDrive\Escritorio\test.js:3:8)
    at Module._compile (node:internal/modules/cjs/loader:1241:14)

TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\andre\OneDrive\Escritorio\test.js:3:8)


TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\andre\OneDrive\Escritorio\test.js:3:8)
    at Module._compile (node:internal/modules/cjs/loader:1241:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1295:10)
    at Module.load (node:internal/modules/cjs/loader:1091:32)
    at Module._load (node:internal/modules/cjs/loader:938:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:83:12)
    at node:internal/main/run_main_module:23:47
```

Tipos de Datos en JavaScript

- **Primitivos**:
  - `string` - Cadenas de texto.
  - `number` - Números enteros y decimales.
  - `boolean` - Verdadero o falso.
  - `null` - Ausencia intencional de un valor.
  - `undefined` - Variable declarada pero sin valor asignado.
- **No primitivos**:
  - `object` - Estructuras complejas como arreglos, funciones y objetos.

Explorando los Tipos de Datos en JavaScript (Parte 1)

Instrucción:

- Explora los distintos tipos de datos en JavaScript escribiendo ejemplos en la consola de tu navegador.
- Prueba los siguientes conceptos:
  1. Declara una variable de tipo `string` y concaténala con otra cadena.
     - `let saludo = "Hola"; let nombre = "Mundo";`
  2. Crea una variable de tipo `number` e intenta hacer una operación matemática.
     - `let resultado = 10 * 3.5;`
  3. Usa un `boolean` en una condición `if`.
     - `let esAdulto = true; if (esAdulto) { console.log("Acceso permitido"); }`
  4. Declara una variable con `null` y otra con `undefined`, luego imprime sus valores.
     - `let variableNula = null; let variableSinDefinir;`

Explorando los Tipos de Datos en JavaScript (Parte 2)

Instrucción (continuación):

- Continúa probando los siguientes conceptos:
  5. Crea un `object` con propiedades y accede a una de ellas.
     - `let persona = { nombre: "Ana", edad: 28 };`
  6. Define un arreglo y accede a uno de sus elementos.
     - `let frutas = ["manzana", "banana", "pera"];`
  7. Declara una función y ejecútala.
     - `function saludar() { console.log("Hola, JavaScript"); }`
- Experimenta con estos tipos y comparte tus resultados con el grupo.

Declaración de Funciones en JavaScript

Funciones tradicionales:

- Las funciones se declaran utilizando la palabra clave `function`.
- Se pueden invocar en cualquier parte del código (hoisting).

Ejemplo:

C: > Users > andre > OneDrive > Escritorio > JS test.js > ...

```
1   function saludar(nombre) {
2       return "Hola, "+ nombre;
3   }
4   console.log(saludar("Maria"));
```

Funciones Flecha en JavaScript

Funciones flecha (`arrow functions`):

- Sintaxis más corta para declarar funciones.
- No tienen su propio contexto de `this`.
- No son *hoisted* como las funciones tradicionales.

Ejemplo:

C: > Users > andre > OneDrive > Escritorio > JS test.js > ...

```
1   const saludar = (nombre) => {
2       return "Hola, " + nombre;
3   }
4
5   console.log(saludar("Andres"));
```

Estructuras de Control en JavaScript

- Las estructuras de control permiten dirigir el flujo del programa.
- Se utilizan para ejecutar código de forma condicional o repetitiva.
- Tipos principales:
  - **Condicionales**: `if`, `else`, `switch`.
  - **Bucles**: `for`, `while`, `do while`.

Condicionales en JavaScript

Uso del condicional `if` y `else`:

- Se utiliza para ejecutar código de forma condicional.
- Si la condición es `true`, se ejecuta el bloque de código.

Ejemplo:

```
let edad = 18;
if (edad >= 18) {
    console.log("Eres mayor de edad");
} else {
    console.log("Eres menor de edad");
}
```

Resultado: `Eres mayor de edad`

Ciclo `for` en JavaScript

Uso del ciclo `for`:

- Se utiliza para ejecutar un bloque de código un número específico de veces.

Sintaxis:

```
for (inicialización; condición; incremento) {
    // código a ejecutar
}
```

Ejemplo:

```
for (let i = 0; i < 5; i++) {
    console.log("Número:  " + i);
}
```

Ciclo `while` en JavaScript

Uso del ciclo `while`:

- Ejecuta un bloque de código mientras una condición sea `true`.

Ejemplo:

```
let contador = 0;
while (contador < 3) {
    console.log("Contador:  " + contador);
    contador++;
}
```

Resultado:

- Contador: 0
- Contador: 1
- Contador: 2

Ciclo `do while` en JavaScript

Uso del ciclo `do while`:

- Ejecuta el bloque de código al menos una vez, luego repite mientras la condición sea `true`.

Ejemplo:

```
let numero = 0;
do {
    console.log("Número:  " + numero);
    numero++;
} while (numero < 3);
```

Uso de `switch` en JavaScript

La estructura `switch`:

- Útil para evaluar una expresión contra múltiples valores posibles.

Ejemplo:

```
let fruta = "manzana";
switch (fruta) {
    case "manzana":
       console.log("Es una manzana");
       break;
    case "banana":
       console.log("Es una banana");
       break;
    default:
       console.log("Fruta desconocida");
}
```


Resultado: `Es una manzana`

Manejo de Errores en JavaScript

- Los errores son situaciones inesperadas que pueden ocurrir durante la ejecución de un programa.
- JavaScript proporciona mecanismos para manejar estos errores y prevenir que el programa falle por completo.
- El manejo de excepciones permite capturar errores y ejecutar un código alternativo.

Palabras clave:

- `try` - Bloque que intenta ejecutar código.
- `catch` - Bloque que maneja errores si ocurren en `try`.
- `finally` - Bloque opcional que se ejecuta siempre, haya ocurrido un error o no.

Uso de `try` y `catch`

Sintaxis básica:

```
try {
    // Código que puede lanzar un error
} catch (error) {
    // Manejo del error
}
```

Ejemplo:

```
try {
    let resultado = 10 / 0;
    console.log("Resultado:  " + resultado);
} catch (error) {
    console.error("Ocurrió un error:",
error.message);
}
```


Resultado: `Resultado:  Infinity`

Uso de `finally`

- El bloque `finally` se ejecuta siempre, independientemente de si ocurrió un error en el bloque `try`.
- Se utiliza para ejecutar código que debe ejecutarse sin importar si hubo un error (por ejemplo, cerrar conexiones).

Ejemplo:

```
try {
    throw new Error("Algo salió mal");
} catch (error) {
    console.error("Error capturado:", error.message);
} finally {
    console.log("Esto siempre se ejecuta");
}
```

Resultado:

- Error capturado: Algo salió mal
- Esto siempre se ejecuta

Lanzar Excepciones Personalizadas con `throw`

- Puedes lanzar tus propios errores utilizando la palabra clave `throw`.
- Útil para validar entradas y evitar situaciones inesperadas en tu aplicación.

Ejemplo:

```
function dividir(a, b) {
    if (b === 0) {
        throw new Error("No se puede dividir por
cero");
    }
    return a / b;
}
try {
    console.log(dividir(10, 0));
} catch (error) {
    console.error("Error:", error.message);
}
```

¿Qué es la Programación Asíncrona?

- La programación asíncrona permite que el código continúe ejecutándose sin esperar a que una tarea se complete.
- JavaScript es de un solo hilo, pero usa un modelo basado en eventos para manejar la concurrencia.
- Ejemplos comunes:
  - Llamadas a APIs externas.
  - Operaciones de lectura/escritura en archivos.
  - Consultas a bases de datos.

¿Qué es una Promesa?

- Una promesa es un objeto que representa la eventual finalización (o falla) de una operación asíncrona.
- Puede tener uno de tres estados:
  - `pending` (pendiente).
  - `fulfilled` (cumplida).
  - `rejected` (rechazada).
- Se utiliza para manejar operaciones asíncronas de manera más legible que los callbacks.

Uso Básico de Promises

Sintaxis para crear una promesa:

```
const miPromesa = new Promise((resolve, reject) => {
    let exito = true;
    if (exito) {
       resolve("Operación exitosa");
    } else {
       reject("Operación fallida");
    }
});
```

Manejo de una promesa:

```
miPromesa
   .then(resultado => console.log(resultado))
   .catch(error => console.error(error));
```

Encadenamiento de Promesas

- El encadenamiento permite ejecutar múltiples promesas en secuencia.
- Cada `then` devuelve una nueva promesa.

Ejemplo:

```
fetch("https://api.example.com/datos")
   .then(res => res.json())
   .then(data => console.log(data))
   .catch(error => console.error("Error:", error));
```

Introducción a `async` y `await`

- `async` y `await` son una forma más moderna y legible de trabajar con promesas.
- Una función declarada con `async` devuelve una promesa.
- El uso de `await` pausa la ejecución de la función hasta que la promesa se resuelva.

Ejemplo de `async` y `await`

Sintaxis básica:

```
async function obtenerDatos() {
    try {
        const respuesta = await
fetch("https://api.example.com/datos");
        const data = await respuesta.json();
        console.log(data);
    } catch (error) {
        console.error("Error al obtener datos:",
error);
    }
}
```

Llamada a la función:

```
obtenerDatos();
```

Manejo de Errores con `async` y `await`

- Es recomendable envolver las operaciones asíncronas en un bloque `try-catch` para capturar errores.

Ejemplo:

```
async function obtenerUsuario() {
    try {
        let usuario = await
fetch("https://api.example.com/usuario");
        return await usuario.json();
    } catch (error) {
        console.error("Error al obtener usuario:",
error);
    }
}
```

MUCHAS GRACIAS