Fundamentos de JavaScript

Diseño Web II

Duodécimo (III BTP) "A"

Fecha: 13 de Julio de 2025

I. Introducción a JavaScript

JavaScript (JS) es un lenguaje de programación ligero e interpretado que ha evolucionado hasta convertirse en uno de los pilares del desarrollo web moderno. Inicialmente diseñado para añadir interactividad a las páginas web en el lado del cliente (es decir, ejecutándose directamente en el navegador del usuario), su versatilidad ha crecido exponencialmente. Hoy en día, JavaScript no solo es fundamental para crear contenido dinámico e interactivo en sitios web, sino que también se utiliza en el desarrollo del lado del servidor (con Node.js), aplicaciones móviles (React Native, NativeScript), aplicaciones de escritorio (Electron) e incluso en el Internet de las Cosas (IoT).

Su integración nativa con HTML y CSS lo convierte en la herramienta esencial para construir experiencias web completas y responsivas, permitiendo manipular el contenido de la página, responder a las acciones del usuario, realizar validaciones de formularios y mucho más.

II. Variables en JavaScript

Las variables son como cajas etiquetadas donde podemos guardar información para usarla más tarde. Cada "caja" tiene un nombre único (la etiqueta) que nos permite encontrar y manipular el dato que contiene. En JavaScript, existen tres palabras clave principales para crear estas "cajas":

a. `var` (*Declaración Antigua*)

Alcance: Tiene un alcance de función o global.
Redeclaración y Reasignación: Las variables `var` pueden ser redeclaradas y reasignadas.
Hoisting: Son "elevadas" al inicio de su alcance.

```
var nombre = "Ana";
console.log(nombre); // Salida: Ana
var nombre = "María"; // Válido, redeclaración
console.log(nombre); // Salida: María
```

B. `let` (Introducido en ES6)

Alcance: Tiene un alcance de bloque (solo existe dentro de las llaves {...} donde se declara).
Redeclaración y Reasignación: Puede ser reasignada, pero no redeclarada en el mismo alcance.
Hoisting: También son elevadas, pero intentar usarlas antes de su declaración causa un error.

```
let edad = 30;
edad = 31; // Válido, reasignación
// let edad = 32; // Error: Identifier 'edad' has already been declared
if (true) {
  let mensaje = "Hola";
}
// console.log(mensaje); // Error: mensaje is not defined (fuera del alcance del bloque)
```

C. `const` (Introducido en ES6)

Alcance: También tiene un alcance de bloque.
Redeclaración y Reasignación: No puede ser reasignada ni redeclarada. Su valor es constante.
Inmutabilidad: Para datos primitivos, el valor es inmutable. Para objetos y arrays, la referencia es inmutable, pero su contenido sí puede cambiar.

```
1  const PI = 3.14159;
2  // PI = 3.14; // Error: Assignment to constant variable.
3  const colores = ["rojo", "verde"];
4  colores.push("azul"); // Válido: el contenido del array puede modificarse
5  // colores = ["amarillo"]; // Error: Assignment to constant variable.
```

Buenas Prácticas sobre Variables:

En el desarrollo moderno de JavaScript, se recomienda utilizar const por defecto para todas las variables. Si sabes que el valor de la variable necesitará cambiar, entonces usa let. Se debe evitar el uso de var para prevenir comportamientos inesperados relacionados con su alcance (scope) y hoisting.

III. Tipos de Datos Fundamentales

JavaScript es un lenguaje de tipado dinámico. Los tipos de datos se dividen en primitivos y objetos.

A. Tipos de Datos Primitivos

`String:` Representa secuencias de caracteres, es decir, texto.
Ejemplo:

```
let saludo = "Hola mundo";
```

`Number:` Representa tanto números enteros como números de punto flotante (decimales).

Ejemplo:

```
let edad = 25
```

`Boolean:` Representa un valor lógico, que puede ser **`true`** (verdadero) o **`false`** (falso).
Ejemplo:

```
let esActivo = true
```

`Undefined`: Indica que una variable ha sido declarada pero aún no se le ha asignado un valor.
Ejemplo:

```
let miVariable; // miVariable es undefined
```

`Null`: Representa la ausencia intencional de cualquier valor de objeto. Es un valor asignado explícitamente para indicar "sin valor".
Ejemplo:

```
let usuario = null;
```

`Symbol` (Introducido en ES6): Representa un identificador único e inmutable.
Ejemplo:

```
const id = Symbol('id');
```

`BigInt` (Introducido en ES2020): Representa números enteros con una precisión arbitraria.
Ejemplo:

```
const numeroGrande = 1234567890123456789012345678901 2...n
```

B. Tipo de Dato de Objeto

`Object:` Es el tipo de dato más complejo y fundamental en JavaScript. Todo lo que no es un tipo primitivo es un objeto. Los objetos son colecciones de propiedades, donde cada propiedad tiene un nombre (clave) y un valor.

Ejemplo:

```
let persona = { nombre: "Juan", edad  30 };
```

Acceso a propiedades:

```
console.log(persona nombre); // Salida: Juan
```

`Array:` Es un tipo especial de objeto utilizado para almacenar colecciones ordenadas de elementos. Los elementos de un array se acceden mediante un índice numérico.

Ejemplo:

```
let frutas = ["manzana", "banana", "cereza"];
```

Acceso a elementos:

```
console.log(frutas[0]); // Salida: manzana
```

`Function:` Las funciones en JavaScript son objetos de primera clase, lo que significa que pueden ser tratadas como cualquier otro valor: pueden ser asignadas a variables, pasadas como argumentos a otras funciones y devueltas por otras funciones.

Ejemplo:

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

IV. Operadores en JavaScript

Los operadores son símbolos especiales que se utilizan para realizar operaciones sobre uno o más valores (operandos) y producir un resultado.

A. Operadores Aritméticos:

```
01. + (Suma)
02. - (Resta)
03. * (Multiplicación)
04. / (División)
05. % (Módulo - devuelve el resto de una división)
06. ** (Exponenciación - a ** b es a elevado a la potencia b)
07. ++ (Incremento - añade 1)
08. -- (Decremento - resta 1)
```

B. Operadores de Asignación:

```
01.  = (Asignación simple)
02.  += (Suma y asignación: x += y es equivalente a x = x + y)
03.  -= (Resta y asignación)
04.  *= (Multiplicación y asignación)
05.  /= (División y asignación)
06.  %= (Módulo y asignación)
07.  **= (Exponenciación y asignación)
```

C. Operadores de Comparación:

```
01. ==  (Igualdad de valor - compara solo el valor, no el tipo)
02. === (Igualdad estricta - compara valor y tipo)
03. !=  (Desigualdad de valor)
04. !== (Desigualdad estricta)
05. >   (Mayor que)
06. <   (Menor que)
07. >=  (Mayor o igual que)
08. <=  (Menor o igual que)
```

D. Operadores Lógicos:

```
01.  && (AND lógico - devuelve true si ambos operandos son true)
02.  || (OR lógico - devuelve true si al menos uno de lo es)
03.  ! (NOT lógico - invierte el valor booleano del operando)
```

Ejemplo Integrado de Operadores:

```
01.  let a = 10;
02.  let b = 4;
03.
04.  // Aritméticos
05.  console.log("Suma: " + (a + b)); // 14
06.  console.log("Resta: " + (a - b)); // 6
07.  console.log("Módulo: " + (a % b)); // 2
08.
09.  // Asignación
10.  a += 5; // a es ahora 15
11.  console.log("Nuevo valor de a: " + a); // 15
12.
13.  // Comparación
14.  console.log("¿Es a > b?: " + (a > b)); // true
15.  console.log("¿Es 5 == '5'?: " + (5 == '5')); // true
16.  console.log("¿Es 5 === '5'?: " + (5 === '5')); // false
17.
18.  // Lógicos
19.  let esValido = (true && false);
20.  console.log("¿Son a y b mayores que 0?: " + esValido); // false
```

V. Estructuras de Control

Las estructuras de control permiten controlar el flujo de ejecución del código, permitiendo que ciertas partes del código se ejecuten solo bajo ciertas condiciones o se repitan un número determinado de veces.

A. Condicionales

1. `if / else if / else:`

Ejecuta un bloque de código si una condición es verdadera. Se pueden encadenar con else if para múltiples condiciones y un else para un bloque por defecto.

```
01. let temperatura = 25;
02. if (temperatura < 0) {
03. console.log("Hace mucho frío.");
04. } else if (temperatura < 20) {
05. console.log("Hace fresco.");
06. } else {
07. console.log("Hace calor.");
08. }
```

2. switch:

Evalúa una expresión contra múltiples valores (case) y ejecuta el bloque de código asociado al primer caso que coincide. Incluye una cláusula default opcional si ningún caso coincide. Es importante usar break para salir del switch una vez que se encuentra una coincidencia.

```
01.let dia = "Lunes";
02.switch (dia) {
03.case "Lunes":
04.console.log("Inicio de semana.");
05.break;
06.case "Viernes":
07.console.log("Fin de semana se acerca.");
08.break;
09.default:
10.console.log("Día normal.");
11.}
```

B. Bucles (Loops)

1. `for:`

Se utiliza cuando se conoce el número de iteraciones. Consiste en una inicialización, una condición de continuación y una expresión de actualización.

```
01. for (let i = 0; i < 5; i++) {
02. console.log("Número: " + i); // Imprime 0, 1, 2, 3, 4
03. }
```

2. while:

Repite un bloque de código mientras una condición especificada sea verdadera. La condición se evalúa antes de cada iteración.

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

3. do...while:

Similar a while, pero garantiza que el bloque de código se ejecute al menos una vez, ya que la condición se evalúa después de la primera iteración.

```
 let i = 0;
 do {
 console.log("Valor: " + i);
 i++;
 } while (i < 0); // Se ejecuta una vez (Valor: 0)
```

4. **`for...in`:**

Itera sobre las propiedades enumerables de un objeto.

```
const persona = { nombre: "Luis", edad: 28 };
for (let clave in persona) {
console.log(`${clave}: ${persona[clave]}`);
}
// Salida:
nombre: Luis
edad:   28
```

5. **`for...of`** (*Introducido en ES6*):

Itera sobre valores de objetos iterables (como Arrays, Strings, Map, Set, etc.).

```
const frutas = ["manzana", "pera", "uva"];
for (let fruta of frutas) {
console.log(fruta);
}
// Salida:
manzana
pera
uva
```

VI. Funciones en JavaScript

Las funciones son como mini-programas o recetas que podemos definir una vez y ejecutar cuantas veces queramos. Son bloques de código reutilizables que realizan una tarea específica.

A. Declaración de Función (`Function Declaration`)

Es la forma más común de definir una función. Se define con la palabra clave `function`, seguida del nombre de la función, una lista de parámetros entre paréntesis y el cuerpo de la función entre llaves. Las declaraciones de función son completamente elevadas, lo que significa que pueden ser llamadas antes de su definición en el código.

```
01.  function saludar(nombre) {
02.    return "Hola, " + nombre + "!";
03.  }
04.    console.log(saludar("Pedro"));
05.  // Salida:
06.  Hola, Pedro!
```

B. **Expresión de Función**

Una función se define como parte de una expresión y se asigna a una variable. La función puede ser anónima (sin nombre) o nombrada. Las expresiones de función no son elevadas de la misma manera que las declaraciones. Solo la variable que contiene la función es elevada, no la función en sí. Por lo tanto, no se pueden llamar antes de su definición.

```
01. const sumar = function(a, b) {
02.   return a + b;
03.   };
04. console.log(sumar(5, 3));
05.
06. // Salida:
07. 8
```

C. Funciones Flecha

Ofrecen una sintaxis más concisa para escribir funciones, especialmente para funciones simples o anónimas. Al igual que las expresiones de función, no son elevadas. Tienen un comportamiento diferente en cuanto al valor de this (contexto léxico), lo que las hace muy útiles en ciertos escenarios (ej., dentro de métodos de objetos o callbacks).

```
01. const restar = (a, b) => {
02.     return a - b;
03.    };
04. console.log(restar(10, 4));
05. const doble = num => num * 2;
06. console.log(doble(7));
07.
08. // Salida:
09. 14
```

VII. Cómo Insertar JavaScript en HTML

JavaScript se integra en documentos HTML de varias maneras, permitiendo que el código interactúe con el contenido de la página.

A. JavaScript Inline

Se inserta directamente en los atributos de las etiquetas HTML, generalmente para manejar eventos.

Ventajas: Rápido para funciones muy pequeñas y específicas.

Desventajas: Dificulta la mantenibilidad, mezcla HTML y JS, no reutilizable.

```
<button onclick="alert('¡Hola desde JS Inline!')">Haz clic aquí</button>
```

B. JavaScript Interno (*Embedded*)

Se coloca dentro de la etiqueta `<script>` directamente en el archivo HTML. Comúnmente se ubica en la sección `<head>` o, preferentemente, al final del `<body>` para asegurar que el HTML se cargue primero.
Ventajas: Conveniente para scripts pequeños que solo afectan a una página.

Desventajas: No es reutilizable en múltiples páginas, el HTML se vuelve menos legible si el script es largo.

```
<!DOCTYPE html>
    <html lang="es">
  <head>
      <title>Página con JS Interno</title>
<script>
  console.log("¡Hola desde JavaScript Interno en el head!");
</script>
  </head>

  <body>
<h1>Mi Página</h1>
    <script>
// Este script se ejecuta después de que el cuerpo de la página ha
sido cargado
    document.getElementById('titulo').innerText = 'Título
      Modificado por JS';
console.log("¡Hola desde JavaScript Interno en el body!");
    </script>
        <h2 id="titulo">Título Original</h2>
  </body>
</html>
```

C. JavaScript Externo

El código JavaScript se escribe en un archivo separado (.js) y se enlaza al documento HTML usando el atributo src de la etiqueta `<script>`. Es la práctica recomendada para la mayoría de los proyectos.

Ventajas: Código más limpio y modular, reutilizable en múltiples páginas, mejora la caché del navegador y la velocidad de carga.

Desventajas: Requiere una solicitud HTTP adicional para cargar el archivo .js.

Archivo `index.html`:

```
01. <!DOCTYPE html>
02. <html lang="es">
03. <head>
04. <title>Página con JS Externo</title>
05. </head>
06. <body>
07. <h1>Mi Página Externa</h1>
08. <script src="mi_script.js"></script>
09. </body>
10. </html>
```

Archivo `mi_script.js`:

```
01. // Contenido de mi_script.js
02. console.log("¡Hola desde JavaScript Externo!");
03. alert("Este es un script externo.");
```