Bootcamp de Desarrollo Web

Clase 8

Javascript

1

¿Qué es Javascript?

JavaScript es un lenguaje de programación de alto nivel, interpretado y orientado a objetos.

Es uno de los tres principales tecnologías web, junto con **HTML (Hypertext Markup Language)** y **CSS (Cascading Style Sheets)**, y se utiliza para crear contenido dinámico e interactivo en páginas web.

2

Breve Historia de Javascript

Aunque su nombre incluye **Java**, JavaScript no está relacionado con el lenguaje de programación Java.

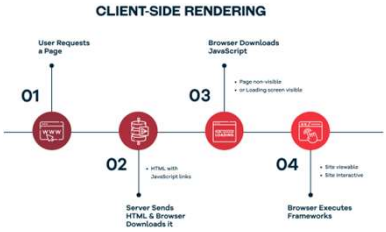
Fue creado por Netscape y originalmente se llamaba LiveScript, pero se cambió a JavaScript cuando Netscape se asoció con Sun Microsystems (los creadores de Java) en la década de 1990.

Java ≠ JavaScript

Client Side Rendering

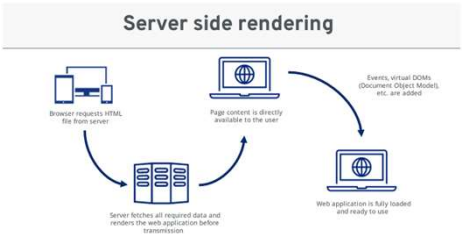
JavaScript comúnmente se ejecuta en el lado del **cliente**, es decir, en el navegador web del usuario, lo que permite interactuar con la página web sin necesidad de comunicarse constantemente con el servidor.

Puede manipular el contenido de la página, responder a eventos del usuario, realizar peticiones HTTP asíncronas (AJAX), y mucho más.



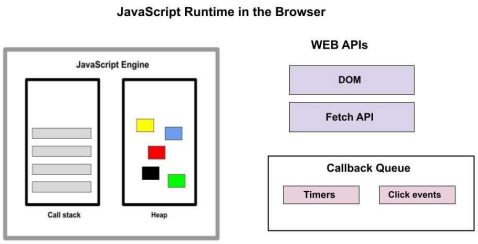
Server Side Rendering

Además de ser utilizado en el desarrollo web, JavaScript también se ha expandido a otros entornos a través de plataformas como **Node.js**, que permite ejecutar JavaScript en el lado del servidor, lo que posibilita el desarrollo de aplicaciones web completas utilizando un solo lenguaje de programación tanto en el cliente como en el servidor.



Javascript Engine

Un motor JavaScript es un programa que ejecuta y procesa código JavaScript. Estos motores son parte integral de los navegadores web y otros entornos que admiten la ejecución de JavaScript. Su función principal es interpretar y ejecutar el código fuente JavaScript.



Interprete VS Compilador

Intérprete:

Un intérprete es un programa que ejecuta código fuente línea por línea en tiempo de ejecución. **No produce un archivo** ejecutable separado antes de la ejecución.

Los programas interpretados son generalmente más lentos que los compilados, pero son flexibles y permiten la depuración y modificación del código en tiempo de ejecución.

Interprete VS Compilador

Compilador: Un compilador es un programa que traduce todo el código fuente a un código ejecutable en un paso previo a la ejecución del programa.

Produce un archivo ejecutable que se puede ejecutar independientemente.

Los programas compilados tienden a ser más rápidos, ya que no hay interpretación en tiempo de ejecución, pero el proceso de compilación puede ser más lento.

Javascript ¿Interpretado o Compilado?

JavaScript es principalmente un lenguaje interpretado. En un navegador web, el motor JavaScript interpreta y ejecuta el código directamente en tiempo de ejecución, leyendo y ejecutando el código línea por línea cuando se encuentra el script.

Aunque JavaScript se interpreta en la mayoría de los casos, avances tecnológicos, como en **Node.js**, han introducido elementos de compilación en tiempo real para mejorar el rendimiento mediante la creación de código de máquina de bajo nivel durante la ejecución.

Tipos de datos

JavaScript es un lenguaje de programación con tipos de **datos dinámicos**, lo que significa que no es necesario declarar explícitamente el tipo de una variable antes de usarla.

Algunos de los tipos de datos fundamentales en JavaScript:

Number (Número): Representa tanto enteros como números de punto flotante. Ejemplo: **let x = 5;**

String (Cadena de texto): Secuencia de caracteres.
Ejemplo: **let mensaje = "Hola";**

Tipos de datos

Boolean (Booleano): Representa un valor lógico verdadero o falso. Ejemplo: **let esVerdadero = true;**

Undefined: Representa un valor indefinido. Si una variable está declarada pero no se le asigna ningún valor, su valor será undefined. **Ejemplo: let variableIndefinida;**

Null: Representa la ausencia intencional de cualquier objeto o valor. Ejemplo: **let variableNula = null;**

Object (Objeto): Una colección no ordenada de pares clave-valor. Ejemplo: **let persona = { nombre: "Juan", edad: 25 };**

Tipos de datos

Array (Arreglo): Una colección ordenada de valores. Ejemplo: **let numeros = [1, 2, 3, 4, 5];**

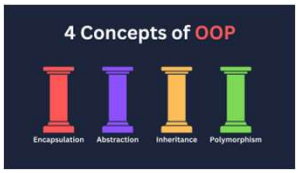
Symbol (Símbolo): Un tipo de dato nuevo introducido en ECMAScript 6, utilizado para crear identificadores únicos. **Ejemplo: let simbolo = Symbol('descripcion');**

BigInt: Un tipo de dato introducido en ECMAScript 2020, que permite representar números enteros de longitud arbitraria. **Ejemplo: let numeroGrande = 1234567890123456789012345678901234567890n;**

Programación Orientada a Objetos

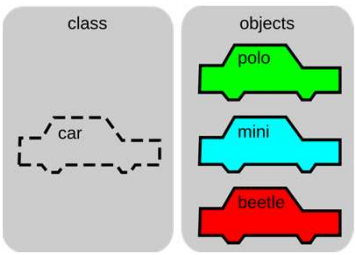
Un lenguaje de programación orientado a objetos (**OOP**, por sus siglas en inglés, **Object-Oriented Programming**) es aquel que se basa en el concepto de **objetos** para organizar y estructurar el código.

Los objetos son instancias de clases que encapsulan datos y comportamientos relacionados. La programación orientada a objetos se basa en varios principios fundamentales.



Clases Y Objetos de OOP

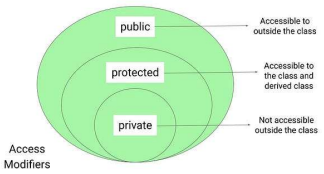
Una clase es una plantilla o modelo para crear objetos. Define las propiedades y comportamientos que los objetos creados a partir de ella tendrán. Los objetos son instancias específicas de una clase.



Encapsulación

Este principio implica encapsular o agrupar los datos (propiedades) y los métodos (funciones) relacionados en una entidad única, el objeto.

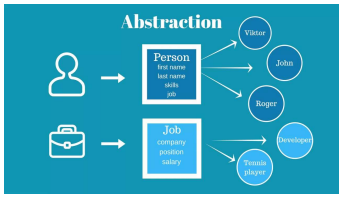
La encapsulación ayuda a ocultar la implementación interna y proporciona una interfaz clara para interactuar con el objeto.



Abstracción

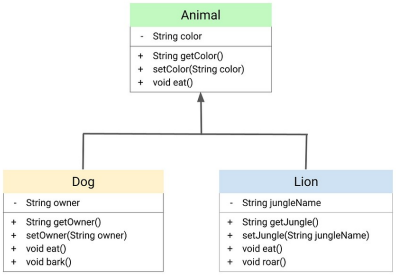
La abstracción en el contexto de la programación orientada a objetos se refiere a la capacidad de representar los detalles esenciales de un objeto y ocultar los detalles innecesarios.

En otras palabras, la abstracción simplifica la realidad modelando solo los aspectos más relevantes de un objeto y permitiendo que el programador se centre en lo que es importante para la resolución de un problema particular.



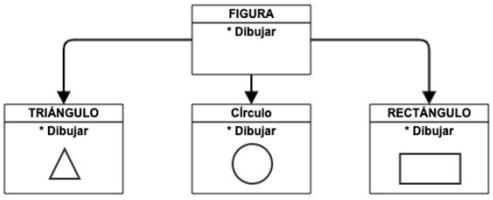
Herencia

La herencia permite que una clase herede propiedades y métodos de otra clase. Esto facilita la reutilización del código y la creación de una jerarquía de clases.



Polimorfismo

El polimorfismo permite que un objeto pueda tomar múltiples formas. En el contexto de la OOP, esto significa que objetos de diferentes clases pueden ser tratados de manera uniforme si comparten una interfaz común.



Acceder a un objeto en JavaScript

Acceder a un objeto en JavaScript implica utilizar la notación de punto **(.)** o la notación de corchetes **([])** para acceder a sus propiedades o métodos. Aquí tienes un ejemplo sencillo:

```
let persona = {
    nombre: "Juan",
    edad: 25,
    saludar: function() {
        console.log("Hola, soy " + this.nombre);
    }
};

// Acceder a las propiedades utilizando la notación de punto
console.log(persona.nombre); // Salida: Juan
console.log(persona.edad);   // Salida: 25

// Llamar a un método del objeto utilizando la notación de punto
persona.saludar(); // Salida: Hola, soy Juan

// Acceder a las propiedades utilizando la notación de corchetes
let propiedad = "nombre";
console.log(persona[propiedad]); // Salida: Juan
```


Clases en Javascript

En JavaScript, las clases fueron introducidas en ECMAScript 2015 (también conocido como ES6) para proporcionar una forma más clara y orientada a objetos de crear objetos y definir su comportamiento.

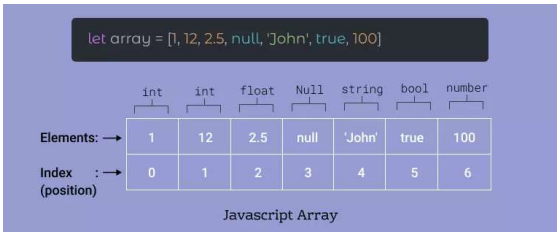
A continuación tienes un ejemplo básico de cómo crear y utilizar una clase en JavaScript:

Clases en Javascript

```javascript
// Definición de una clase Persona
class Persona {
    // Constructor para inicializar propiedades
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método de la clase
    saludar() {
        console.log(`Hola, soy ${this.nombre} y tengo ${this.edad} años.`);
    }
}

// Creación de una instancia de la clase Persona
let persona1 = new Persona("Ana", 30);

// Acceso a las propiedades y llamada al método de la clase
console.log(persona1.nombre); // Salida: Ana
console.log(persona1.edad);   // Salida: 30
persona1.saludar();           // Salida: Hola, soy Ana y tengo 30 años.
```


Arrays en Javascript

En JavaScript, un array es un tipo de dato que te permite almacenar varios elementos en una única variable. Los elementos pueden ser de cualquier tipo, y los arrays en JavaScript son dinámicos, lo que significa que pueden cambiar de tamaño durante la ejecución del programa.



Creación de un array

Puedes crear un array utilizando la siguiente sintaxis:

```
let miArray = [elemento1, elemento2, elemento3];
```

Acceso a Elementos del Array:
Los elementos de un array se acceden mediante un índice, empezando por 0. Por ejemplo:

```
let frutas = ["Manzana", "Plátano", "Uva"];
console.log(frutas[0]);  // Salida: Manzana
console.log(frutas[1]);  // Salida: Plátano
console.log(frutas[2]);  // Salida: Uva
```


Métodos Comunes de los Arrays en JavaScript:

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

```
frutas.push("Naranja");
console.log(frutas);  // Salida: ["Manzana", "Plátano", "Uva", "Naranja"]
```

pop(): Elimina el último elemento del array.

```
frutas.pop();
console.log(frutas);  // Salida: ["Manzana", "Plátano", "Uva"]
```

Métodos Comunes de los Arrays en JavaScript:

shift(): Elimina el primer elemento del array.

```
frutas.shift();
console.log(frutas);  // Salida: ["Plátano", "Uva"]
```

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

```
frutas.unshift("Fresa", "Piña");
console.log(frutas);  // Salida: ["Fresa", "Piña", "Plátano", "Uva"]
```


Métodos Comunes de los Arrays en JavaScript:

splice(): Permite modificar el contenido del array eliminando o reemplazando elementos.

```
frutas.splice(1, 2, "Mango", "Cereza");
console.log(frutas);  // Salida: ["Fresa", "Mango", "Cereza", "Uva"]
```

slice(): Devuelve una parte del array sin modificar el array original.

```
let parteDelArray = frutas.slice(1, 3);
console.log(parteDelArray);  // Salida: ["Mango", "Cereza"]
```

Métodos Comunes de los Arrays en JavaScript:

concat(): Combina dos o más arrays creando uno nuevo.

```
let masFrutas = ["Kiwi", "Sandía"];
let nuevoArray = frutas.concat(masFrutas);
console.log(nuevoArray);  // Salida: ["Fresa", "Mango", "Cereza", "Uva", "Kiwi", "Sandía"]

```

indexOf(): Devuelve el índice de la primera ocurrencia del elemento especificado en el array.

```
let indice = frutas.indexOf("Cereza");
console.log(indice);  // Salida: 2
```


Métodos Comunes de los Arrays en JavaScript:

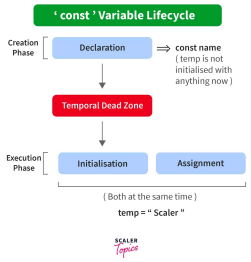
forEach(): Ejecuta una función proporcionada una vez para cada elemento del array.

```
frutas.forEach(function(fruta) {
    console.log(fruta);
});
// Salida:
// Fresa
// Mango
// Cereza
// Uva
```

Hoisting

Es un comportamiento en JavaScript que eleva las declaraciones de variables y funciones al principio de su ámbito antes de ejecutar el código.

Permite usar variables o funciones antes de que se declaren.



var, let y const

- **Hoisting:** Eleva la declaración, pero no inicializa con undefined. Puedes usar la variable antes de declararla.
- **Ámbito:** Función-scoped (visible en toda la función).
- **Reasignación:** Puede ser reasignada y redeclarada.

```
console.log(miVariable); // OK, muestra 'undefined'
var miVariable = 5;
```

var

- **Hoisting:** Eleva la declaración, pero no inicializa con undefined. Puedes usar la variable antes de declararla.
- **Ámbito:** Función-scoped (visible en toda la función).
- **Reasignación:** Puede ser reasignada y redeclarada.

```
console.log(miVariable); // OK, muestra 'undefined'
var miVariable = 5;
```


let

- **Hoisting:** Eleva la declaración, pero no puedes acceder a su valor antes de la declaración.
- **Ámbito:** Block-scoped (visible solo dentro del bloque).
- **Reasignación:** Puede ser reasignada, pero no redeclarada en el mismo bloque.

```
// console.log(miVariable); // Error, no se puede acceder antes de la declaración
let miVariable = 5;
```


const

- **Hoisting:** Eleva la declaración, pero no puedes acceder a su valor antes de la declaración.
- **Ámbito:** Block-scoped (visible solo dentro del bloque).
- **Reasignación:** No puede ser reasignada ni redeclarada.

```
// console.log(pi); // Error, no se puede acceder antes de la declaración
const pi = 3.14;
```



const

- **Hoisting:** Eleva la declaración, pero no puedes acceder a su valor antes de la declaración.
- **Ámbito:** Block-scoped (visible solo dentro del bloque).
- **Reasignación:** Puede ser reasignada, pero no redeclarada en el mismo bloque.

```
// console.log(miVariable); // Error, no se puede acceder antes de la declaración
let miVariable = 5;
```

Bootcamp de Desarrollo Web

Clase 8

Javascript