UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

PROGRAMACIÓN WEB

Nombre: Sebastian Lasso.

NRC: 2257

Fecha: viernes 10/01/2025

Tema: Trabajo en clase 3

Análisis y Desarrollo: Clases en JavaScript para Sistemas Dinámicos

Objetivo

Desarrollar un análisis profundo sobre la aplicabilidad y las mejores prácticas del uso de clases en JavaScript, y posteriormente implementar una solución funcional basada en los hallazgos para un problema real de software.

Instrucciones

1. Escenario

Estás trabajando como desarrollador en una empresa que busca crear una aplicación para administrar la operación de un servicio de transporte urbano inteligente. La plataforma debe gestionar registros de conductores, asignación de vehículos, control de rutas y monitoreo de tiempos de viaje.

Dado que la solución debe ser escalable, modular y fácil de mantener, la empresa ha decidido explorar la implementación de clases en JavaScript para estructurar el código de manera más efectiva.

2. Análisis

Analizar y responder a las siguientes preguntas para guiar tu desarrollo:

Abstracción y Modularidad:

¿Cómo estructurarías las clases para representar entidades como conductores, vehículos y rutas?

¿Qué propiedades y métodos serían necesarios para cada clase?

Para el problema propuesto establecería 3 clases: conductor, vehículo y ruta donde:

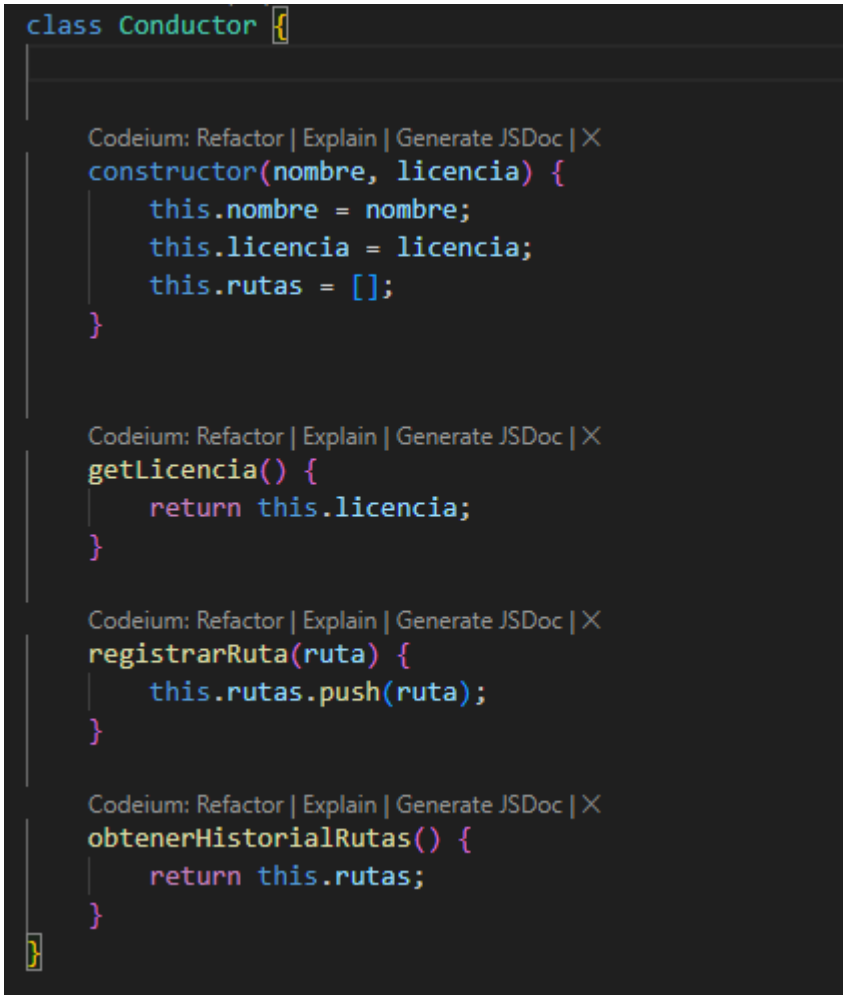
Clase Conductor va a tener:

Propiedades:

- nombre (string): Nombre del conductor. licencia (string): Licencia del conductor. rutas (array): Lista de rutas realizadas.

Métodos:

- registrarRuta(ruta): Agrega una nueva ruta al historial. obtenerHistorialRutas(): Retoma el historial de rutas.



Clase Vehiculo va a tener:

- Propiedades:
  - modelo (string): Modelo del vehículo. placa (string): Placa del vehículo. asignadoA (Conductor): Referencia al conductor asignado.
- Métodos:
  - asignarConductor(conductor): Asocia un conductor al vehículo. obtenerConductor(): Retorna el conductor asignado.

```
class Vehiculo {
    Codeium: Refactor | Explain | Generate JSDoc | X
    constructor(modelo, placa) {
        this.modelo = modelo;
        this.placa = placa;
        this.asignadoA = null; // Conductor asignado
    }

    Codeium: Refactor | Explain | Generate JSDoc | X
    asignarConductor(conductor) {
        this.asignadoA = conductor;
    }

    Codeium: Refactor | Explain | Generate JSDoc | X
    obtenerConductor() {
        return this.asignadoA;
    }
}
```

Clase Ruta va a tener:

- Propiedades:
    - origen (string): Punto de partida de la ruta. destino (string): Destino de la ruta. duracion (número): Duración de la ruta en minutos.

```
// Clase Ruta
Codeium: Refactor | Explain
class Ruta {
    Codeium: Refactor | Explain | Generate JSDoc | X
    constructor(origen, destino, duracion) {
        this.origen = origen;
        this.destino = destino;
        this.duracion = duracion; // Duración en minutos
    }
}
```

Encapsulación: ¿Qué propiedades de cada clase deberían mantenerse privadas para proteger la integridad de los datos (por ejemplo, información personal del conductor)?

Propiedades privadas:

- En la clase **Conductor**, la propiedad licencia debería mantenerse privada, ya que contiene información que no debe ser expuesta ante todo el mundo

```
#licencia;

Codeium: Refactor | Explain | Generate JSDoc | X
constructor(nombre, licencia) {
    this.nombre = nombre;
    this.#licencia = licencia;
    this.rutas = [];
}
```

Herencia y Extensibilidad: ¿Cómo aprovecharías la herencia para diferenciar entre tipos de usuarios, como conductores regulares y conductores de servicios VIP?

Para aprovechar la herencia cree una clase base que seria conductor y mediante la herencia cree otra clase de conductorVIP en el cual podemos añadir nuevos atributos extras que pueden tener esta clase como vehiculo favorito.

```
class ConductorVIP extends Conductor {
    Codeium: Refactor | Explain | Generate JSDoc | X
    constructor(nombre, licencia) {
        super(nombre, licencia);
        this.vehiculoPreferido = null; // Preferencia de vehículo
    }

    Codeium: Refactor | Explain | Generate JSDoc | X
    asignarVehiculoPreferido(vehiculo) {
        this.vehiculoPreferido = vehiculo;
    }

    Codeium: Refactor | Explain | Generate JSDoc | X
    obtenerVehiculoPreferido() {
        return this.vehiculoPreferido;
    }
}
```

Escalabilidad: ¿Qué prácticas implementarías para asegurar que el sistema pueda expandirse fácilmente conforme se añadan nuevas funcionalidades?

Para asegurar que el sistema sea escalable, implementaría las siguientes prácticas:

1. **Diseño modular**:
    - Mantener clases pequeñas y enfocadas en una responsabilidad.
    - Crear relaciones claras entre las entidades.
2. **Principios SOLID**:
    - **Responsabilidad única**: Cada clase tiene una única razón para cambiar.
    - **Abierto/Cerrado**: Las clases están abiertas para extensión pero cerradas para modificación.
3.

3. Desarrollo

Implementa una solución funcional en JavaScript que cumpla con los siguientes criterios:

Definición de Clases:

Define una clase Conductor con propiedades básicas como nombre, licencia, y métodos como registrarRuta().

Implementa una clase Vehiculo para gestionar la asignación de vehículos, con propiedades como modelo y placa.

Herencia:

Extiende la clase Conductor para crear una clase ConductorVIP con beneficios adicionales como asignarVehiculoPreferido().

Encapsulación:

Utiliza propiedades privadas para proteger información sensible del conductor.