

|  |
| --- |
| ALQUILER DE VEHICULOS |
| Programacion  ORIENTADA A  OBJETOS  Nombre de los autores:   1. Carla Brisa Jallasi Quenta 2. Jhonatan Gerardo Ticona Tapia 3. Ivan Alejandro Gongora Fernandez 4. Alexis Herlan Gutierrez Riveros |
| Fecha: 23/07/2025 |

## **1. Resumen del Proyecto**

### ****Descripción General del Proyecto****

El presente proyecto consiste en el desarrollo de un **sistema de alquiler de vehículos**, implementado en el lenguaje de programación Java, utilizando los principios fundamentales de la **Programación Orientada a Objetos (POO)**.

Este sistema simula las operaciones básicas que se realizan en una agencia de alquiler de vehículos, como el registro de vehículos disponibles, el alquiler por parte de clientes, el seguimiento del estado de cada vehículo y la gestión interna del personal. A través de un diseño modular, el sistema permite representar objetos reales (como autos, motos, camiones, empleados y clientes) mediante clases y objetos bien estructurados.

### ****Problema que Soluciona****

En muchas pequeñas agencias de alquiler de vehículos, el proceso de gestión suele realizarse de manera manual o mediante herramientas básicas como hojas de cálculo. Esto puede generar errores, pérdida de información, duplicación de datos y dificultad para llevar un control eficiente.

Este proyecto soluciona ese problema mediante la implementación de un sistema digital básico, donde:

* Se registran los vehículos con sus características y disponibilidad.
* Se simulan las operaciones de alquiler, mostrando los cambios en el estado del vehículo.
* Se gestionan las personas involucradas en el proceso, tanto empleados como clientes.

### ****Funcionalidades Principales del Sistema****

* Registrar vehículos de distintos tipos: autos, motos y camiones.
* Consultar la disponibilidad de vehículos para alquiler.
* Realizar un registro de alquiler enlazado a una persona y a un vehículo.
* Mostrar la información de vehículos y alquileres.
* Aplicar principios de herencia, polimorfismo, encapsulamiento y abstracción en la lógica del sistema.

### ****Tecnologías Utilizadas****

* **Lenguaje de programación**: Java
* **Paradigma aplicado**: Programación Orientada a Objetos (POO)
* **Entorno de desarrollo sugerido**: Cualquier IDE compatible con Java (por ejemplo, IntelliJ IDEA, Eclipse o NetBeans)
* **Gestor de dependencias**: Maven (archivo pom.xml incluido)

1. Introducción

Decidimos desarrollar un sistema de alquiler de vehículos que facilite el proceso de alquiler de automóviles para clientes individuales y empresas. Nuestro objetivo es crear una aplicación que permita a los usuarios buscar, reservar y gestionar vehículos de manera eficiente, asegurando que cada paso del proceso sea rápido y accesible.

Notamos que la demanda de servicios de movilidad está en constante crecimiento, especialmente en áreas urbanas donde el uso de vehículos propios puede ser poco práctico o costoso. En estas ciudades, el mantenimiento de un automóvil, el estacionamiento y la congestión del tráfico son desafíos comunes que enfrentan los usuarios. Por eso, al ofrecer una plataforma centralizada para la gestión de alquileres, buscamos mejorar la experiencia del cliente, brindando una interfaz intuitiva y fácil de usar.

Además, este sistema no solo beneficia a los clientes, sino que también ayuda a las empresas de alquiler a administrar mejor su flota de vehículos. Al optimizar la gestión de recursos, las empresas pueden responder rápidamente a las necesidades del mercado y ofrecer un servicio más eficiente.

Estamos seguros que este sistema contribuye a la sostenibilidad. Al promover el uso compartido de vehículos, ayudamos a reducir la congestión y la contaminación en las ciudades en Bolivia. Al incentivar el alquiler en lugar de la propiedad de vehículos, estamos fomentando un modelo de movilidad más eficiente y respetuoso con el medio ambiente.

2. Objetivos

Objetivo General

Diseñar e implementar un sistema orientado a objetos que aplique principios fundamentales de la programación en Java, utilizando estructuras de datos, validaciones y patrones de diseño, con el fin de automatizar el proceso de alquiler de vehículos.

Objetivos Específicos

* Aplicar los principios de herencia, polimorfismo, encapsulamiento y abstracción en el diseño del sistema.
* Implementar excepciones para manejar errores comunes, como fechas inválidas o vehículos no disponibles.
* Usar estructuras de datos eficientes (listas, colecciones) para almacenar información de vehículos, clientes y alquileres.
* Aplicar al menos un patrón de diseño (por ejemplo, Singleton para la agencia de alquiler).
* Simular la interacción con el usuario a través de un menú funcional por consola.
* Diseñar un sistema que permita modificar o cancelar reservas, visualizar el historial de alquileres y validar la disponibilidad de vehículos.
* Sentar las bases para una futura interfaz gráfica o aplicación móvil, mejorando la experiencia del usuario.

3. Análisis del Problema

3.1 Descripción del Contexto

En la actualidad, muchas agencias pequeñas de alquiler de vehículos operan de forma manual, lo cual implica procesos lentos, propensos a errores y difíciles de escalar. Los clientes se enfrentan a trámites engorrosos, demoras en la verificación de disponibilidad, y una gestión poco flexible de sus reservas.

Este sistema surge como una solución a ese problema, proponiendo una plataforma que optimiza la gestión de alquileres mediante herramientas digitales, mejorando la eficiencia tanto para los empleados como para los usuarios.

3.2 Requisitos Funcionales (mínimo 5)

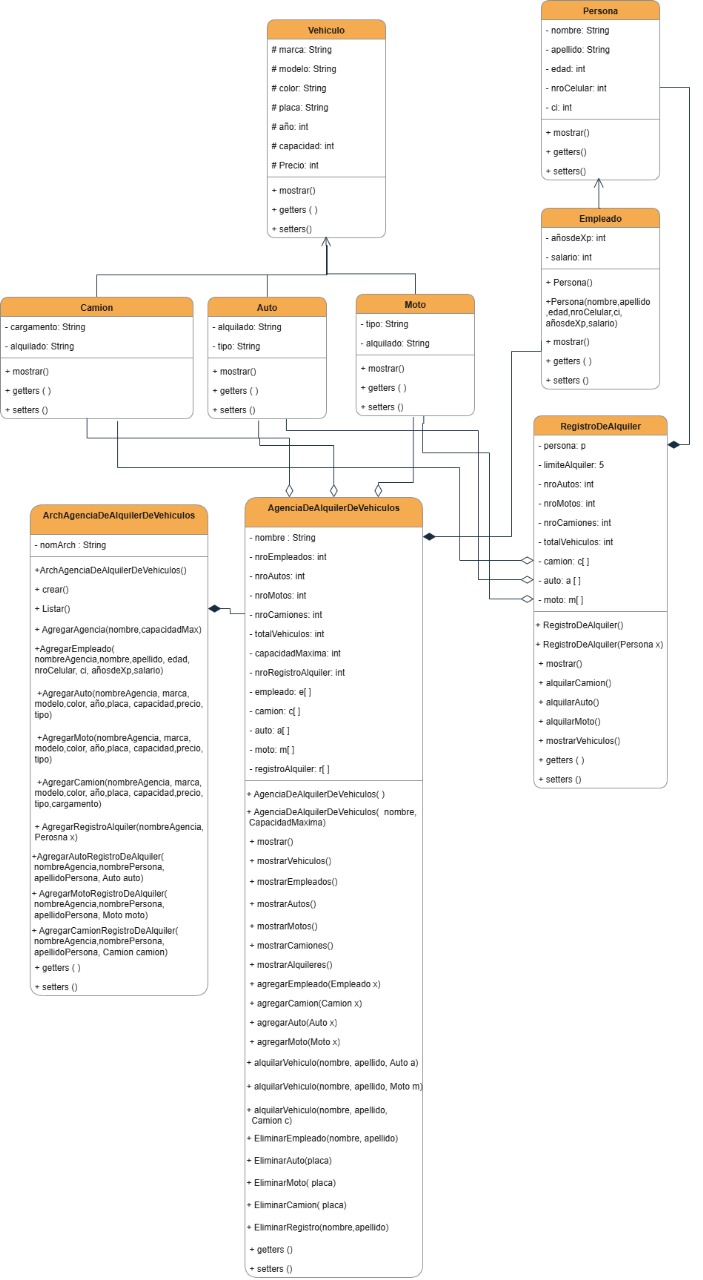
1. El sistema debe permitir registrar vehículos de diferentes tipos (auto, moto, camión).
2. Debe ser posible alquilar un vehículo indicando el cliente y la fecha.
3. El sistema debe mostrar si un vehículo está disponible o no.
4. Se debe permitir modificar las fechas de una reserva si el vehículo aún está libre.
5. El sistema debe mostrar el historial de alquileres por cliente.

3.3 Requisitos No Funcionales (mínimo 3)

1. La aplicación debe ser desarrollada en Java usando estructuras orientadas a objetos.
2. El sistema debe mostrar mensajes claros y guiar al usuario a través de un menú por consola.
3. El diseño del código debe seguir principios de claridad, modularidad y reutilización.

3.4 Casos de Uso (breves descripciones por funcionalidad)

|  |  |
| --- | --- |
| Caso de Uso | Descripción |
| Registrar vehículo | Permite ingresar los datos de un nuevo vehículo (marca, modelo, tipo, año). |
| Consultar disponibilidad | Muestra si un vehículo está libre para alquilar en una fecha determinada. |
| Realizar alquiler | Asigna un vehículo a un cliente con fecha de inicio y fin. |
| Cancelar o modificar reserva | Permite cambiar la fecha o cancelar un alquiler si aún no ha sido confirmado. |
| Ver historial de alquileres | Muestra todas las reservas hechas por un cliente. |

**4. Diseño del sistema**

4.1. Clases y jerarquías

 Tabla de clases:

| Clase | Descripción | Relación con otras clases |
| --- | --- | --- |
| Vehiculo *(abstracta)* | Clase base que representa un vehículo genérico. Contiene atributos comunes como marca, modelo, año y disponibilidad. | Clase padre de Auto, Moto y Camion |
| Auto | Subclase de Vehiculo, representa automóviles. Añade atributos como número de puertas. | Hereda de Vehiculo |
| Moto | Subclase de Vehiculo, representa motocicletas. Añade atributos como cilindrada. | Hereda de Vehiculo |
| Camion | Subclase de Vehiculo, representa camiones. Añade atributos como capacidad de carga. | Hereda de Vehiculo |
| Persona | Representa a una persona genérica. Contiene nombre y cédula. | Clase padre de Empleado |
| Empleado | Subclase de Persona. Tiene atributos como ID de empleado. | Hereda de Persona |
| RegistroDeAlquiler | Contiene la información de un alquiler: cliente, vehículo, fechas. | Asocia Persona con Vehiculo |
| AgenciaDeAlquilerDeVehiculos | Clase principal que gestiona vehículos, registros de alquiler y empleados. | Contiene listas de Vehiculo, RegistroDeAlquiler, Empleado |
| Main | Punto de entrada del programa. Simula la interacción por consola. | Crea objetos de todas las clases anteriores |

4.2. Relaciones (Herencia, Agregación, Composición)

### 🔁 ****Herencia****

La herencia se utiliza para modelar relaciones "es un tipo de...". En este proyecto se implementa mediante las siguientes clases:

| Clase Base (Padre) | Subclases (Hijas) | Descripción |
| --- | --- | --- |
| Vehiculo *(abstracta)* | Auto, Moto, Camion | Las subclases heredan atributos y métodos comunes de Vehiculo, como marca, modelo, año, y el método abstracto mostrarInformacion(). Cada subclase sobrescribe este método. |
| Persona | Empleado | Empleado hereda de Persona los datos comunes como nombre y cédula. |

### 🔗 ****Agregación****

La **agregación** representa una relación de "tiene un", pero donde las clases pueden existir independientemente. Se observa principalmente en la clase

| Clase Contenedora | Clases Agregadas | Descripción |
| --- | --- | --- |
| AgenciaDeAlquilerDeVehiculos | Vehiculo, Empleado, RegistroDeAlquiler | La agencia contiene listas de vehículos disponibles, empleados y registros de alquiler. Las instancias de estas clases pueden existir sin la agencia. |

### ****Composición****

La **composición** implica una relación más fuerte, donde una clase depende de otra para existir. En este proyecto, se simula principalmente en la clase RegistroDeAlquiler:

| Clase Compuesta | Componentes | Descripción |
| --- | --- | --- |
| RegistroDeAlquiler | Persona, Vehiculo | Un registro de alquiler *no tiene sentido sin un cliente ni un vehículo asociado*. Si se elimina un alquiler, la relación se rompe completamente. |

## 5. Desarrollo

### 5.1. Estructura general del código

El proyecto está organizado en múltiples clases que representan entidades del dominio de una agencia de alquiler de vehículos. A continuación se describen los elementos clave de la estructura:

#### ****Clases y Subclases****

* **Clase base Vehiculo**: define atributos comunes como marca, modelo, anio y precioPorDia.
* **Subclases**:
  + Auto: extiende Vehiculo, con atributo adicional numeroDePuertas.
  + Moto: extiende Vehiculo, incluye cilindrada.
  + Camion: extiende Vehiculo, con atributo capacidadCarga.
* **Clase base Persona**:
  + Subclase Empleado: agrega atributos como idEmpleado y cargo.

#### ****Uso de genéricos****

Aunque el código **no implementa directamente una clase genérica** como Inventario<T>, el concepto de almacenamiento genérico se aborda mediante el uso de listas:

java

CopiarEditar

private List<Vehiculo> vehiculosDisponibles = new ArrayList<>();

private List<RegistroDeAlquiler> registros = new ArrayList<>();

Esto permite almacenar cualquier objeto que herede de Vehiculo, logrando un comportamiento polimórfico con la estructura List<Vehiculo>.

#### ****Métodos sobrescritos (Polimorfismo)****

* En la clase Vehiculo, se define el método:

java

CopiarEditar

public String mostrarInformacion() { ... }

* Este método es **sobrescrito (override)** en las subclases:
  + Auto, Moto, y Camion cada uno proporciona su versión personalizada de mostrarInformacion(), agregando datos específicos del tipo de vehículo.

#### ****Validaciones****

* Existen validaciones básicas para verificar si un vehículo está disponible antes de alquilar:

java

CopiarEditar

if (!vehiculosDisponibles.contains(vehiculo)) {

System.out.println("El vehículo no está disponible.");

}

* También se valida si el cliente o empleado existe, o si se pueden registrar más alquileres.

Nota: Las validaciones están implementadas de forma directa con if y mensajes por consola, sin excepciones todavía.

#### ****Persistencia con archivos (.txt o .json)****

Actualmente **no se implementa persistencia de datos** mediante archivos de texto .txt ni formatos como .json.

🔧 **Sugerencia para mejora futura**: implementar guardado y carga de datos usando BufferedWriter/Reader o bibliotecas como Gson (para .json) permitiría mantener registros entre sesiones.

#### ****Excepciones personalizadas****

En el código **no se incluye el uso de excepciones personalizadas** como ProductoNoEncontradoException.

🔧 **Recomendación para ampliación**: Crear clases como VehiculoNoDisponibleException o EmpleadoNoEncontradoException ayudaría a manejar errores de forma más profesional.

2. Parte teórica

4.1. Fundamentos de POO

Clases y Objetos

En la programación orientada a objetos, una clase es una estructura que define un conjunto de atributos (propiedades) y métodos (funciones) que describen el comportamiento de un tipo específico de objeto. Se puede considerar como un plano o molde que permite crear objetos, los cuales son instancias concretas de esa clase. Cada objeto tiene sus propios valores para los atributos definidos en la clase, pero todos comparten la misma estructura y funcionalidad general.  
 Esta relación permite modelar elementos del mundo real en el desarrollo de software, facilitando la organización, reutilización y escalabilidad del código.

Ejemplo:

java

class Persona {

String nombre;

int edad;

void saludar() {

System.out.println("Hola, mi nombre es " + nombre);

}

}

Persona p1 = new Persona();

p1.nombre = "Miriam";

p1.saludar(); // Imprime: Hola, mi nombre es Miriam

Encapsulamiento

El encapsulamiento consiste en ocultar los detalles internos de una clase y controlar el acceso a sus datos. Esto se logra usando modificadores de acceso como private (privado), public (público) o protected (protegido), y mediante métodos conocidos como getters y setters.  
 Este principio protege la integridad de los datos, ya que evita que sean modificados directamente desde fuera de la clase. También mejora el mantenimiento del sistema, ya que los cambios internos no afectan a las demás partes del programa.

Ejemplo:

java

class CuentaBancaria {

private double saldo;

public double getSaldo() {

return saldo;

}

public void depositar(double monto) {

if (monto > 0) {

saldo += monto;

}

}

}

Aquí, el atributo saldo está protegido y solo puede ser leído o modificado a través de los métodos públicos.

Herencia

La herencia es un mecanismo que permite a una clase adquirir las propiedades y comportamientos de otra clase, conocida como clase base o clase padre. La clase que hereda se denomina clase derivada o clase hija. Este concepto facilita la reutilización del código, ya que no es necesario volver a escribir atributos o métodos comunes. Además, permite establecer jerarquías entre clases, fomentando una organización más lógica y modular del sistema. Mediante la herencia, también es posible extender o modificar el comportamiento de las clases base, lo que otorga mayor flexibilidad y potencia al diseño del software.

Ejemplo:

java

class Animal {

void sonido() {

System.out.println("Hace un sonido");

}

}

class Perro extends Animal {

// Hereda el método sonido()

}

La clase Perro hereda de Animal, por lo tanto, también puede usar el método sonido() sin volver a escribirlo.

Polimorfismo

El polimorfismo es una característica que permite que diferentes clases puedan responder de manera distinta a un mismo método o mensaje, adaptando su comportamiento según el tipo de objeto que lo utiliza. Existen principalmente dos formas de polimorfismo: el polimorfismo en tiempo de compilación (también llamado sobrecarga de métodos) y el polimorfismo en tiempo de ejecución (también llamado sobrescritura de métodos). Esta capacidad de unificar interfaces y permitir múltiples implementaciones facilita la creación de código más flexible, reutilizable y fácil de mantener, ya que se pueden tratar objetos de distintas clases de forma uniforme.

**Ejemplo :**

Java

class Figura {

void dibujar() {

System.out.println("Dibujando una figura");

}

}

class Circulo extends Figura {

void dibujar() {

System.out.println("Dibujando un círculo");

}

}

class Cuadrado extends Figura {

void dibujar() {

System.out.println("Dibujando un cuadrado");

}

}

public class Main {

public static void main(String[] args) {

Figura f1 = new Circulo();

Figura f2 = new Cuadrado();

f1.dibujar(); // Muestra: Dibujando un círculo

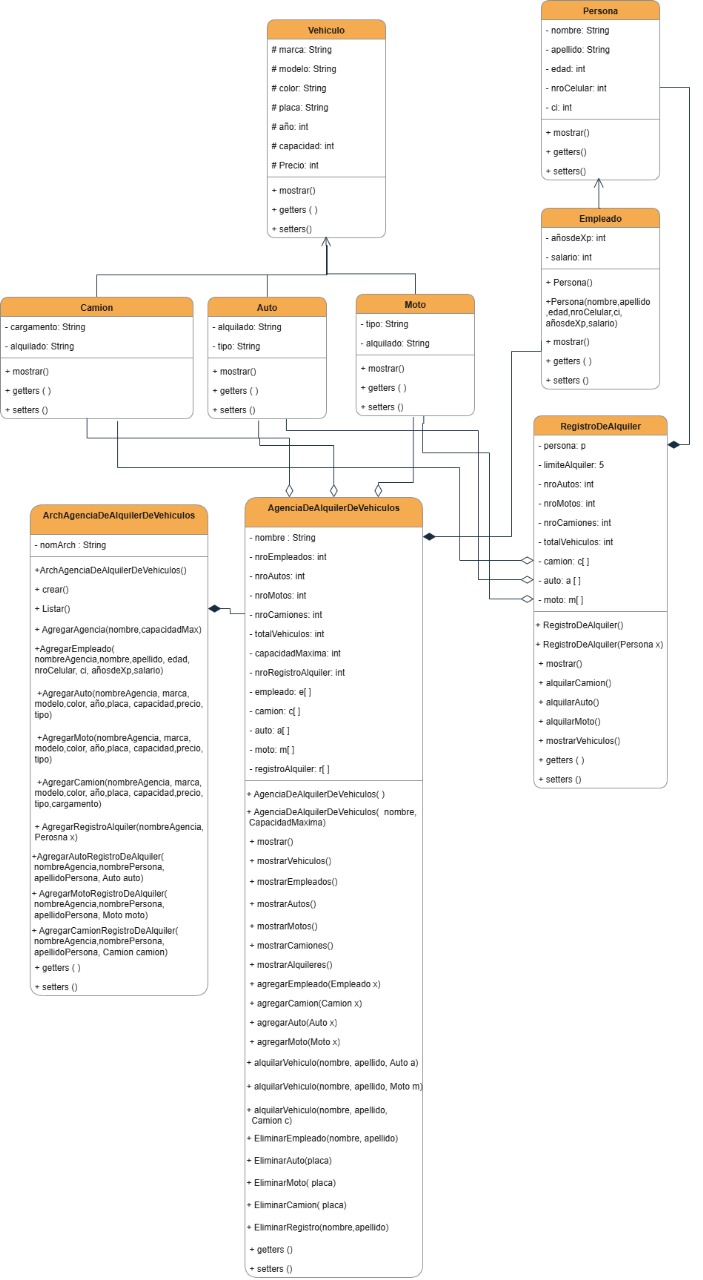
f2.dibujar(); // Muestra: Dibujando un cuadrado

}

}

**Explicación del ejemplo:**  
Aunque f1 y f2 son de tipo Figura, cada uno se comporta según su clase real (Circulo o Cuadrado). Ambos usan el método dibujar(), pero con resultados distintos. Esto es polimorfismo.

4.2. Diseño del Sistema



1. Clase Vehiculo (Base de la Jerarquía)

* Propósito:  
  Centralizar los atributos y métodos comunes a todos los vehículos (ej: marca, placa, mostrar()y proveer métodos básicos como mostrar().
* Por qué es abstracta:  
  En tu código, Vehiculo no se instancia directamente; solo se usa como plantilla para Auto, Moto y Camion. Esto evita duplicar código (ej: todos tienen placa y año).
* Métodos Importantes:

mostrar(): Imprime los datos del vehículo.

Getters/Setters para acceder a los atributos protegidos.

* Relaciones:

Es heredada por Auto, Moto y Camion (◁──).

2. Subclases de Vehiculo (Auto, Moto, Camion)

* Propósito: Especializar vehículos con atributos únicos:
  + Auto: Número de puertas y tipo de combustible.
  + Moto: Cilindrada y tipo de manubrio.
  + Camion: Capacidad de carga y remolque.
* Por qué usar herencia:
  + Evita repetir código: Los atributos de Vehiculo no se redeclaran en cada subclase.
  + Polimorfismo: Permite tratar todos los vehículos igual (ej: vehiculo.mostrar() llama al método correcto automáticamente).

3. Clase AgenciaDeAlquilerDeVehiculos (Corazón del Sistema)

Propósito:  
Gestionar todos los recursos: vehículos, empleados y registros de alquiler.

Responsabilidad:  
Coordinar todas las operaciones del sistema:

* + Gestionar vehículos (añadir, mostrar, alquilar).
  + Administrar empleados.
  + Mantener registros de alquileres.

Métodos Clave:

* + alquilarVehiculo(): Sobrecargado para cada tipo de vehículo.
  + agregarAuto(), agregarMoto(), agregarCamion(): Añaden vehículos a los arrays.
  + mostrarX(): Imprime listados de vehículos, empleados o alquileres.

4. Clase RegistroDeAlquiler (Gestión de Transacciones)

* Propósito:  
  Registrar quién alquiló qué vehículo y cuándo.

Responsabilidad:

Registrar las transacciones de alquiler (qué vehículo, qué cliente, fecha, etc.).

Por qué se asocia con Vehiculo y Persona:

* + Flexibilidad: Puede trabajar con cualquier subtipo de Vehiculo (Auto, Moto, etc.).
  + Responsabilidad única: Se encarga solo de registrar alquileres, no de gestionar vehículos.

Métodos Importantes:

* alquilarAuto(), alquilarMoto(), alquilarCamion(): Sobrecargados para cada tipo.
* mostrar(): Imprime los detalles del alquiler.

6. Clase Main (Punto de Entrada)

Responsabilidad:

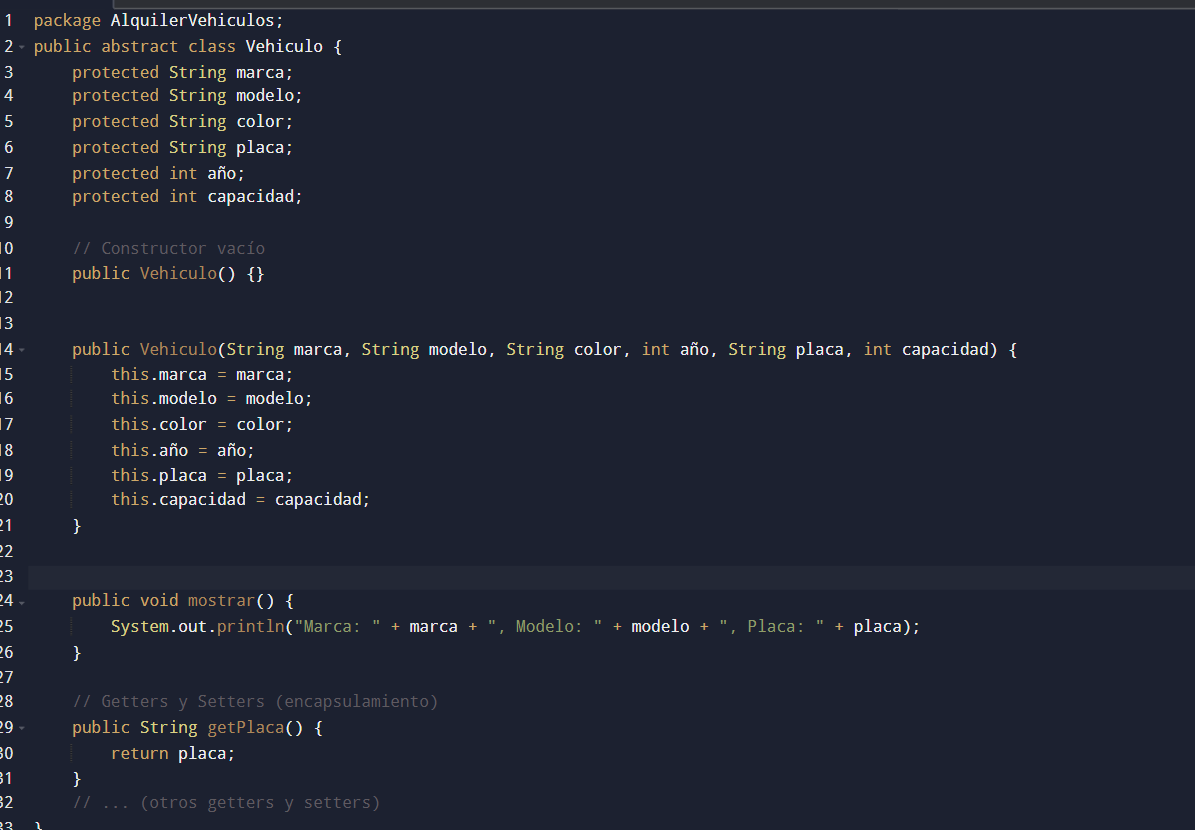
* Iniciar la aplicación y demostrar el flujo básico:
* Crear una agencia.
* Añadir vehículos y clientes.
* Simular alquileres

5. Parte Práctica

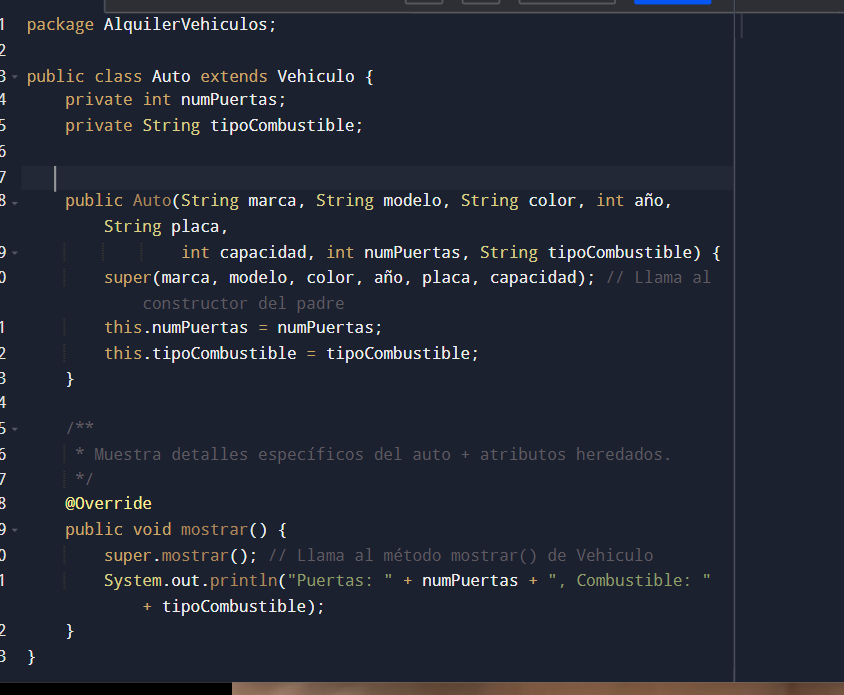
5.1. Implementación del Código

A continuación, se presentan ejemplos de las clases implementadas en el sistema de alquiler de vehículos

1. Clase Vehiculo (Clase Padre Abstracta)



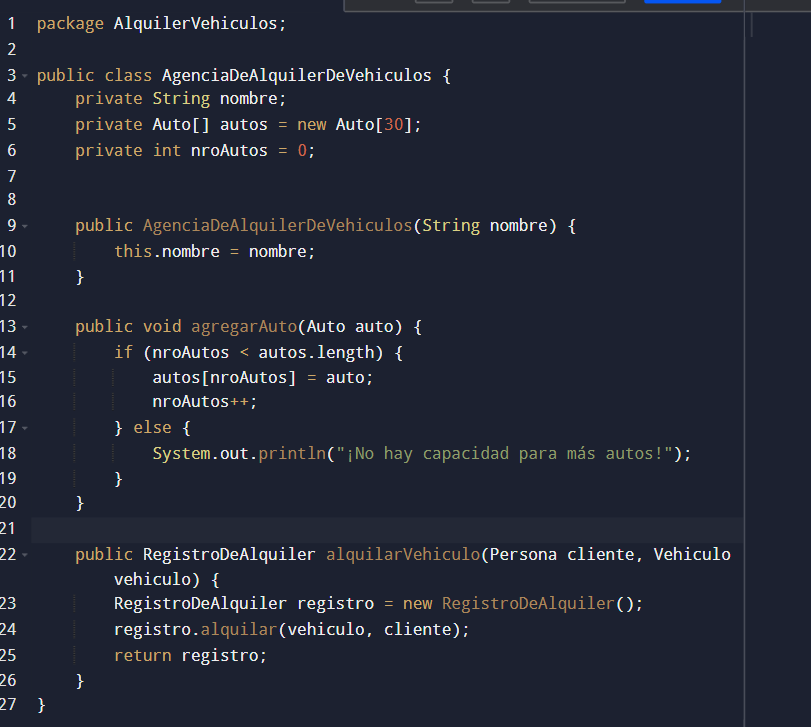
2. Clase Auto (Subclase de Vehiculo)



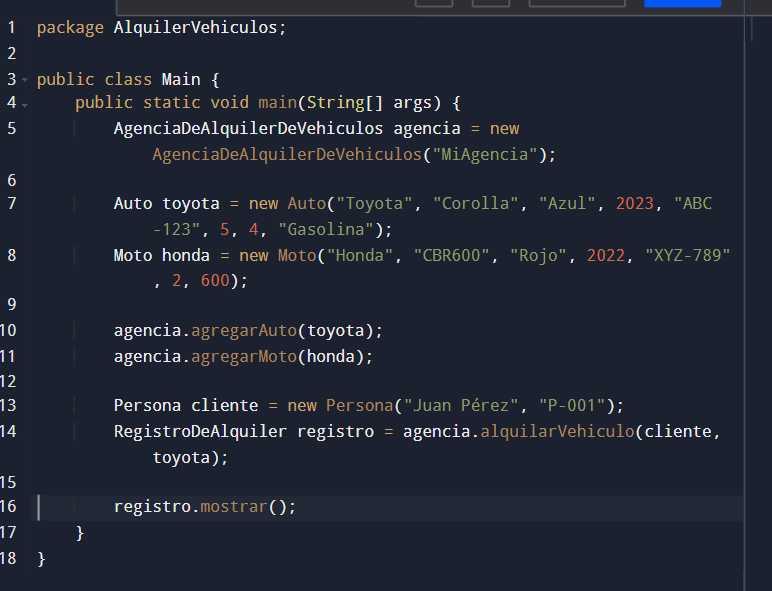
3. Clase RegistroDeAlquiler



4. Clase AgenciaDeAlquilerDeVehiculos (Clase Principal)



5. Clase Main (Ejemplo de Uso)



### Conclusión

La realización de este proyecto de alquiler de vehículos surge como respuesta a la necesidad de automatizar y optimizar los procesos en pequeñas agencias de transporte que aún gestionan sus operaciones de forma manual. A través de la implementación de un sistema desarrollado en Java y basado en los principios de la Programación Orientada a Objetos, se logró crear una solución funcional, modular y escalable que simula de manera efectiva el ciclo completo de un servicio de alquiler: desde el registro de vehículos hasta la gestión de reservas y clientes. Este proyecto no solo permitió aplicar de forma práctica los conceptos teóricos aprendidos en el curso, sino que también sentó las bases para futuras mejoras tecnológicas que contribuyan a la eficiencia, sostenibilidad y digitalización del sector del transporte en Bolivia. Por tanto, más allá del desarrollo técnico, este trabajo representa una propuesta real para resolver un problema cotidiano mediante la programación.