**Introducción** detallada a la Programación Orientada a Objetos (POO) en Java, cubriendo **conceptos fundamentales** como:

1. **Clases y Objetos:**

En POO, el código se organiza en clases que contienen atributos y métodos. Una clase es como un plano para crear objetos (instancias).

1. **Abstracción:**

Define las características esenciales de un objeto, omitiendo detalles específicos. Las clases son la herramienta principal para la abstracción en POO.

1. **Modularidad:**

Permite la separación del código en diferentes unidades (clases), facilitando la modificación y el mantenimiento.

1. **Encapsulación:**

Oculta los detalles internos de una clase y expone solo los métodos necesarios para el resto del programa.

1. **Jerarquía:**

Establece relaciones entre clases y objetos, como la herencia (generalización o especialización) y la composición.

1. **Polimorfismo:**

Permite que diferentes clases ofrezcan distintas implementaciones de un mismo método.

1. **Identidad, Estado, Comportamiento de los Objetos:**

Cada objeto tiene su identidad única, un conjunto de atributos (estado) y métodos (comportamiento).

1. **Creación de Objetos:**

Se realiza mediante la declaración, instanciación y, a menudo, inicialización.

1. **Métodos de Clase:**

Funciones dentro de una clase que definen el comportamiento de los objetos.

* + **toString():** es utilizado para obtener una representación en forma de cadena de texto del estado de un objeto.

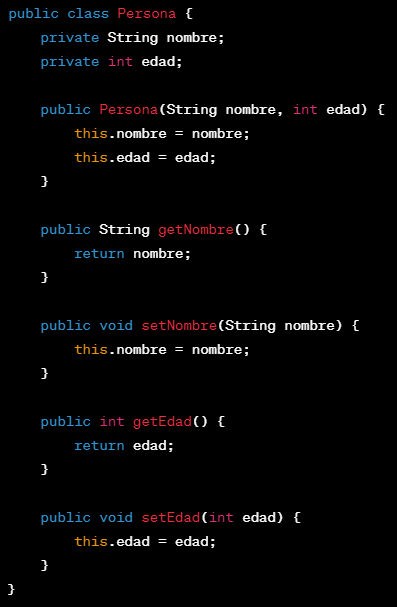
**Visibilidad** y el uso de **constructores** en la Programación Orientada a Objetos (POO) en Java, explicando cómo controlar el acceso a clases, métodos y propiedades, y cómo inicializar objetos:

1. **Visibilidad en POO:**

* Public: Accesible desde cualquier clase.
* Private: Accesible solo dentro de la clase donde se define.
* Protected: En Java, accesible dentro de la clase, sus clases derivadas, y las clases en el mismo paquete.
* Package Private: Por defecto en Java si no se especifica la visibilidad; accesible solo dentro del mismo paquete.

1. **Modificación de Clases y Atributos:**

* Cambiar atributos de una clase a atributos privados para controlar su acceso.
* Uso de visibilidad pública para métodos que interactúan con la clase desde fuera.
* Ejemplo de cómo la visibilidad afecta el acceso a propiedades desde otras clases.

1. **Getters y Setters:**

* Métodos para leer (getters) y modificar (setters) los valores de los atributos privados.
* Creación automática de estos métodos en muchos IDEs.
* No siempre es necesario generar getters y setters para todos los atributos.

1. **Constructores:**

* Métodos especiales para inicializar objetos.
* Se ejecutan automáticamente al crear un objeto.
* Pueden asignar valores iniciales a los atributos del objeto.
* No tienen valor de devolución y su nombre debe coincidir con el de la clase.

1. **La Palabra Reservada `this`:**

* Referencia a la instancia actual de la clase.
* Usada para diferenciar entre atributos de clase y parámetros de métodos cuando tienen el mismo nombre.

**Métodos estáticos** en la Programación Orientada a Objetos (POO) en Java (son útiles para funcionalidades que no dependen del estado de una instancia particular, como operaciones matemáticas o utilidades generales):

1. **Métodos Estáticos:**

* Permiten ejecutar métodos de una clase sin necesidad de crear un objeto.
* El método `main` es un ejemplo de un método estático.
* Si un método no es estático, no se puede llamar sin una instancia de la clase.

1. **Definición y Uso de Métodos Estáticos:**

* Se define con la palabra clave `static`.
* Para usar un método estático, se llama con el nombre de la clase `Calculadora.sumar(3, 5)`.
* Los métodos estáticos son útiles para acciones que no requieren datos de una instancia específica.

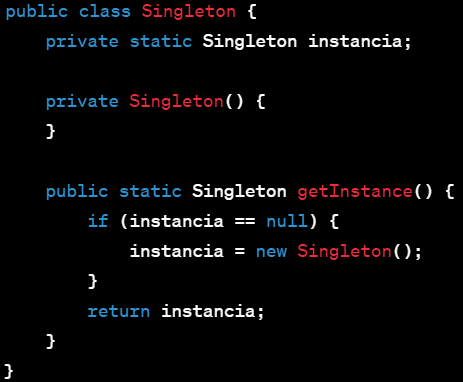
1. **Acceso a Propiedades en Métodos Estáticos:**

* Las propiedades a las que se accede desde un método estático también deben ser estáticas.
* No se puede usar `this` en métodos estáticos porque `this` se refiere a una instancia específica, que no existe en el contexto estático.

1. **Propiedades Estáticas:**

* Una propiedad estática pertenece a la clase, no a una instancia específica.

Todos los objetos de la clase comparten la misma propiedad estática.

1. **Patrón Singleton:**

* Permite crear una sola instancia de una clase.
* El constructor es privado para evitar la creación de múltiples instancias.
* Se utiliza un método `getInstance()` para obtener la instancia única.

1. **Generación de IDs Únicos:**

* Crear una clase `GeneradorId` siguiendo el patrón Singleton.
* Generar IDs únicos utilizando `UUID.randomUUID()`.

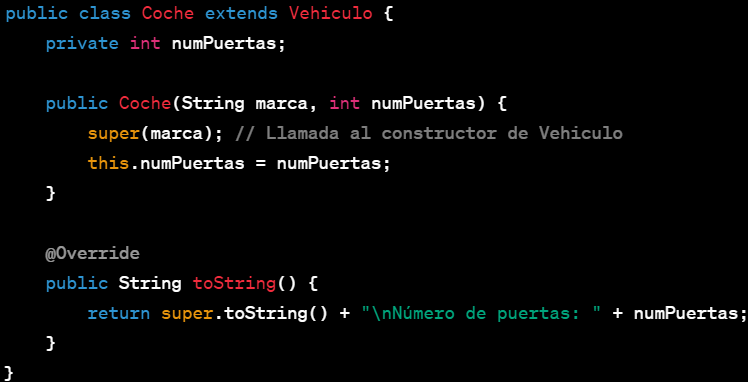
**Herencia** y **composición** en la Programación Orientada a Objetos (POO) en Java:

1. **Herencia:**

* Permite crear una clase basada en otra existente (clase hija a partir de una clase padre).
* Favorece la reutilización de código y estructura la aplicación en una jerarquía de clases (pero puede llevar a una estructura rígida y problemas de escalabilidad).
* Ejemplo: Crear una clase `Coche` heredada de una clase `Vehiculo`.

1. **Uso de `super`:**

* La palabra reservada `super` se utiliza para llamar a métodos o constructores de la clase padre desde la clase hija.
* Ejemplo: En el constructor de `Coche`, usar `super(marca, modelo, velocidad, presionRuedas)` para invocar el constructor de `Vehiculo`.

1. **Sobrescribir Métodos:**

* Las clases hijas pueden sobrescribir métodos de las clases padres para adaptar o extender su comportamiento.
* Ejemplo: Sobrescribir el método `toString()` en la clase `Coche` para incluir atributos específicos de `Coche`.

1. **Clases y Métodos Finales:**

* Usar `final` en una clase o método para prevenir la herencia o sobreescritura.
* Ejemplo: `public final class Vehiculo` impide que se herede de `Vehiculo`.

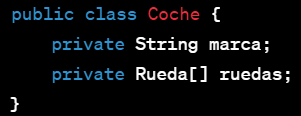
1. **Polimorfismo:**

* Capacidad de tratar objetos de clases derivadas como si fueran de la clase base.
* Ejemplo: Usar un objeto `Coche` donde se espera un objeto `Vehiculo`.



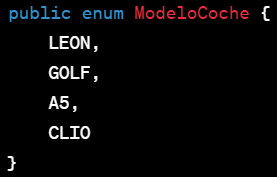
1. **Composición:**

* Consiste en construir clases que contienen objetos de otras clases, en lugar de heredar de ellas.
* Ofrece más flexibilidad y favorece el desacoplamiento, permitiendo construir sistemas más modulares y mantenibles
* Ejemplo: La clase `Coche` contiene objetos de la clase `Rueda`.



**Enumerados** en la Programación Orientada a Objetos (POO) en Java (*son especialmente útiles cuando se necesita representar un conjunto fijo de opciones o constantes relacionadas en un programa. Proporcionan una forma clara y segura de definir y utilizar estas constantes, lo que mejora la legibilidad del código y evita errores al limitar las opciones disponibles a las definidas en el enumerado*):

1. **Definición:**

* Clases especiales para listas de constantes.
* ****No extensibles, ya que heredan de enum.
* Constructores con visibilidad private o de paquete.

1. **Creación de Enumerados:**

* Uso de enum para definir enumeraciones.
* Adición de constantes enumeradas separadas por comas.

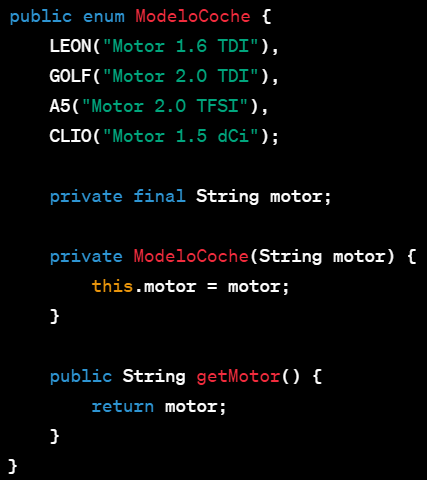
1. **Uso de Enumerados:**

* Referencia a constantes con EnumName.CONSTANT.
* Ideal para limitar opciones (ejemplo: modelos de coche).

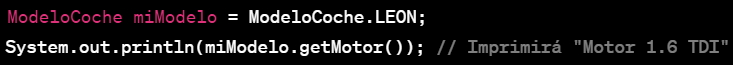


1. **Enumerados con Constructor:**

* Posibilidad de agregar atributos adicionales.
* Uso de constructor privado para definir propiedades de constantes.



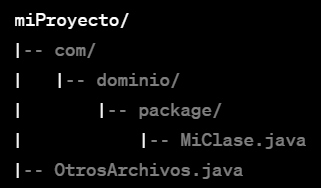
1. **Llamada al Enumerado con Constructor:**



**Librerías** y **paquetes** en la Programación Orientada a Objetos (POO) en Java

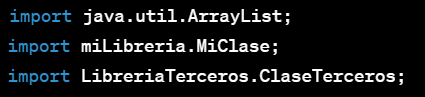
1. **Introducción a Librerías:**

* Definición: Conjunto de clases con métodos y atributos que se utilizan de manera recurrente.
* Utilidad: Reutilización de código, evitando la implementación propia.



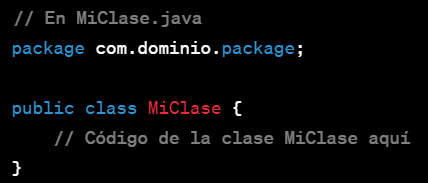
1. **Uso de Librerías en Java (Librerías estándar vs. librerías externas):**

* La librería java.lang no es necesario importarla ya que es la más utilizada.
  + Posee clases como: String, Object o System
* Librerías como de Java como java.util, librerías propias o de terceros sí sería necesario importarlas.
* Las librerías externas suelen distribuirse en formato JAR (Java ARchive), lo que facilita su integración y distribución



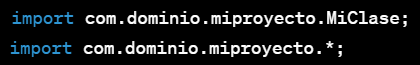
1. **Creación de Paquetes:**

* Definición: Agrupación de interfaces y clases para modularidad.
* Nombrado: El paquete com.dominio.miproyecto se almacena en la ruta com/dominio/miproyecto (Uso del nombre de dominio inverso)
* Una clase siempre deberá pertenecer a un paquete y se utilizará package para referenciar a dicho paquete.



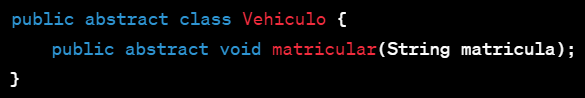
1. **Importación de Paquetes:**

* Cláusula import para incorporar clases específicas o todos los miembros de un paquete en el código.
* Importaciones específicas (clase por clase) y genéricas (todo el paquete usando \*).

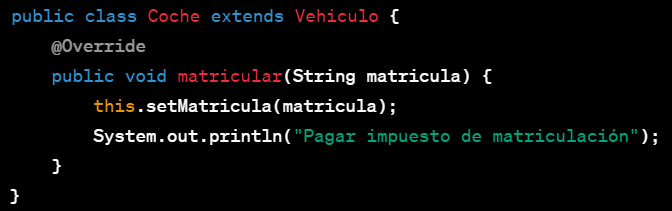


**Métodos abstractos** e **Interfaces** en la Programación Orientada a Objetos en Java

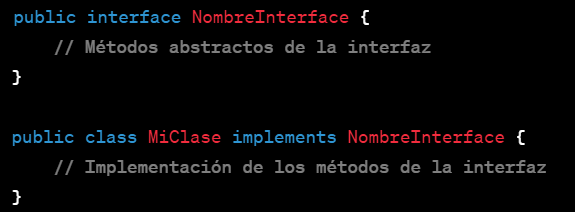
1. **Introducción a Clases y Métodos Abstractos:**

* Concepto: Los métodos abstractos son declaraciones de métodos sin implementación, forzando a las subclases a proporcionar una implementación específica.
* Propiedades:
  + Obligatoriamente una clase será abstracta si posee un método abstracto.
  + Podemos declarar clases abstractas sin necesidad de tener algún método abstracto.
  + Un método abstracto solo se puede definir como protected o public.
  + No podemos reducir la visibilidad de un método abstracto en las clases hijas.
* Ejemplo:

1. **Implementación de Métodos Abstractos en Clases Derivadas:**

* En las clases derivadas es obligatorio implementar los métodos abstractos de las clases predecesoras.
* Ejemplo:

1. **Introducción a Interfaces:**

* Definición: Las interfaces son contratos que definen métodos abstractos sin implementación. Permiten una forma de polimorfismo, donde diferentes clases pueden implementar la misma interfaz de maneras diferentes.
* Características:
  + Pueden ser públicas o sin modificador de acceso.
  + Los métodos serán abstractos o estáticos y siempre públicos.
  + Los atributos serán públicos, estáticos, finales y con un valor.
  + Una clase puede heredar de varias interfaces.
* Declaración e implementación:
* Utilidad: Las interfaces son clave en el diseño de software, ayudando a desacoplar clases y a promover la reutilización del código. Por ejemplo, podrían usarse para definir capacidades comunes que pueden ser implementadas por diferentes clases.