Programación Orientada A Objetos

Sergio Steeven Moreno Forero

Facultad De Ingeniería, Universidad De Cundinamarca

Programación II

William Alexander Matallana Porras

7 de marzo del 2025

Tabla de contenido

Introducción

La programación orientada a objetos (POO) es un paradigma fundamental en el desarrollo de software, basado en la representación de entidades del mundo real a través de objetos que encapsulan datos y comportamientos. Java, uno de los lenguajes de programación más utilizados a nivel global, adopta este enfoque para ofrecer una estructura modular, reutilizable y escalable en la construcción de aplicaciones. Este trabajo tiene como objetivo analizar los principios fundamentales de la POO en Java, incluyendo conceptos clave como clases, objetos, herencia, polimorfismo y encapsulamiento. Además, se explorará cómo este paradigma facilita la creación de software más flexible, estructurado y adaptable a las necesidades cambiantes del desarrollo tecnológico.

Objetivos

Objetivo general

Comprender los conceptos básicos de la programación orientada a objetos en Java y su importancia en el desarrollo de software.

Objetivos específicos

- Definir los principios fundamentales de la programación orientada a objetos, como clases, objetos, herencia, polimorfismo y encapsulamiento.
- Reconocer la utilidad de la POO en la organización y reutilización del código en el desarrollo de aplicaciones.
- Explorar ejemplos básicos en Java para visualizar el funcionamiento de la POO en la práctica.

Desarrollo

Qué es la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un enfoque de programación que se basa en la creación y manipulación de "objetos". Estos objetos son entidades que combinan datos y comportamientos en un solo paquete.

En la POO, los objetos tienen atributos que representan sus características o propiedades, y métodos que definen las acciones que pueden realizar. Las clases se utilizan para definir la estructura y el comportamiento de los objetos. Una clase actúa como una plantilla o un modelo a partir del cual se crean objetos individuales con características únicas.

Por ejemplo, considera una aplicación que gestiona vehículos. Podríamos definir una clase "Coche" que tenga atributos como "marca", "modelo" y "color", y métodos como "arrancar" y "parar". Luego, podemos crear múltiples objetos de la clase "Coche", cada uno con sus propias características y la capacidad de realizar las mismas acciones.

La POO organiza el código de manera modular al combinar datos y comportamientos en objetos, lo que facilita la creación, mantenimiento y reutilización de código en el desarrollo de software. Muchos lenguajes de programación populares son orientados a objetos, como es el caso de Java, C++, Python y C#. A través de este enfoque, los programadores pueden diseñar y construir sistemas de software más organizados, modulares y fáciles de mantener.

Qué son las clases en Java

En Java, las clases son una parte fundamental de la POO y son la base para la creación de objetos, que son instancias de esas clases. Una clase en Java es un plano o un modelo que define la estructura y el comportamiento de los objetos que se pueden crear a partir de ella.

Imagina que estás desarrollando un videojuego de carreras de coches. En este juego, tienes diferentes tipos de coches, como deportivos, todoterrenos y coches de rally. Cada coche tiene ciertas características y habilidades únicas. En Java, puedes utilizar clases para representar estos coches.

Una clase en Java sería como el plano para crear un coche en tu juego. La clase define qué características tiene el coche (como su modelo, velocidad máxima, potencia del motor, etc.) y qué puede hacer (como acelerar, frenar, girar, etc.).

Qué es una clase abstracta

Las clases abstractas, como su nombre lo indica, son algo abstracto, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas.

Podemos imaginar una clase Animal, con métodos como caminar y comer, como una clase base que podemos heredar para construir otras clases como León o Pájaros. Ambas van a heredar de nuestra clase animal con sus respectivos métodos y tendremos la posibilidad de crear nuestros objetos. De esta manera podemos reducir código duplicado y mejorar la calidad del código.

En Java declaramos una clase abstracta con la palabra reservada abstract.

```
public abstract class Animal {  1 usage  1 inheritor

    public  Animal(String value){  no usages
        // Constructor
        this.value = value;
    }

    public abstract void sound()  no usages
}
```

También podemos hacer lo mismo con los métodos: si una clase tiene métodos abstractos, entonces nuestra clase deberá ser abstracta.

Nuestro método abstracto será compartido por las clases que hereden de nuestra clase abstracta. En el ejemplo, un animal puede comportarse de manera similar y realizar las mismas acciones, como caminar, comer y dormir, pero el sonido emitido no será igual en todos ellos, no escucharás a un pájaro rugir como un león.

Conceptos fundamentales de las clases

A continuación, te presentamos algunos de los conceptos fundamentales de las clases en Java:

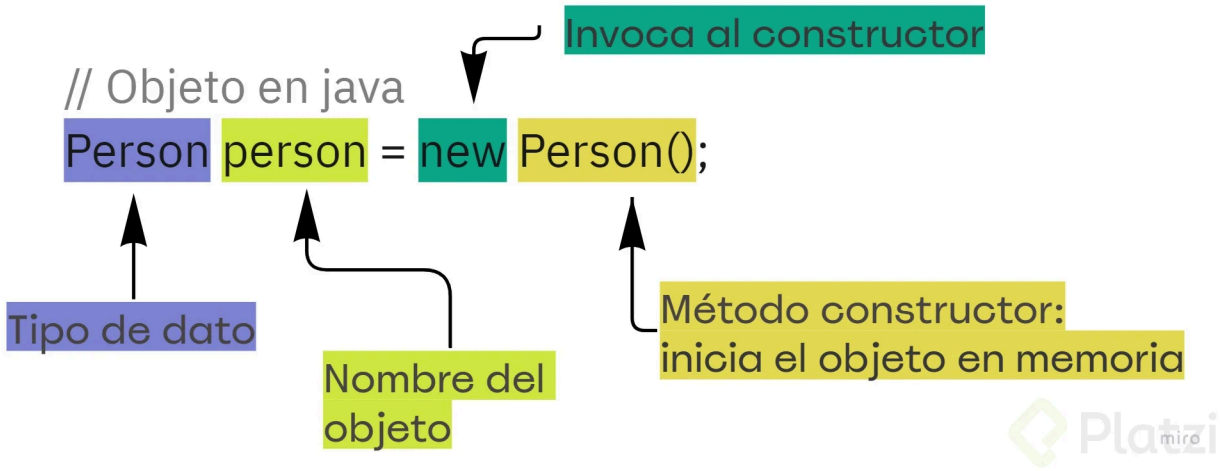
- **Atributos:** Las clases pueden tener atributos, también conocidos como variables de instancia, que representan las características o propiedades del objeto. En el ejemplo del coche los atributos serían el modelo, la velocidad máxima, etc.
- **Métodos:** Las clases contienen métodos que representan el comportamiento del objeto. Los métodos son funciones que pueden realizar acciones y manipular los atributos de la clase. En nuestro caso sería acelerar, frenar, girar, etc.
- **Constructores:** Las clases pueden tener constructores, que son métodos especiales utilizados para inicializar objetos cuando se crean. Los constructores tienen el mismo nombre que la clase y pueden tener diferentes parámetros para configurar el estado inicial del objeto.

Qué son los objetos en Java

En Java, un objeto es una instancia específica de una clase que encapsula datos y comportamientos relacionados. Los objetos se crean a partir de clases, que como hemos comentado definen la estructura y el comportamiento de dichos objetos. Cada objeto tiene atributos que almacenan datos y métodos que definen las operaciones que puede realizar.

En el contexto del videojuego de carreras de coches, los objetos son las representaciones concretas de los coches que existen en el juego. Cada objeto coche corresponde a un tipo específico de coche, como un deportivo, un todoterreno o un coche de rally.

Relación entre una clase, un objeto y un método constructor



La relación entre una clase, un objeto y un método constructor es la siguiente:

- Una clase es como un “plano” que define cómo deben ser los objetos que se creen a partir de ella. Define las propiedades (características) y los métodos (comportamientos) que estos objetos tendrán.
- Un objeto es una instancia de una clase. Es decir, es una realización concreta del “plano” que la clase representa. Cada objeto tiene sus propias copias de las propiedades definidas en la clase y puede acceder a los métodos que la clase define.
- Un método constructor es un método especial dentro de una clase que se utiliza para crear e inicializar un objeto de esa clase. Cuando se crea un nuevo objeto, el método constructor de la clase se ejecuta automáticamente, estableciendo las propiedades iniciales del objeto.

Por lo tanto, la relación entre ellos es que una clase define cómo deben ser los objetos, un objeto es una instancia de una clase y un método constructor es el proceso que se utiliza para crear un objeto a partir de una clase.

Qué son las interfaces en Java

Las interfaces en Java son una parte crucial de la POO que permiten definir un contrato que las clases deben cumplir. En esencia, una interfaz en Java define un conjunto de métodos que deben ser implementados por cualquier clase que implemente esa interfaz. Es como un acuerdo formal que garantiza que una clase tendrá ciertos comportamientos.

La principal característica de una interfaz es que solo declara métodos, pero no proporciona implementaciones para ellos. Es decir, no define el "cómo" de la funcionalidad, solo especifica el "qué". Las clases que implementan una interfaz deben proporcionar sus propias implementaciones para cada uno de los métodos declarados en la interfaz.

- **Declaración de interfaces en Java**

  La declaración de una interfaz en Java es sencilla y sigue una sintaxis específica. Para declarar una interfaz, utilizamos la palabra clave interface, seguida del nombre de la interfaz y un bloque de código que contiene la lista de métodos que la interfaz va a declarar.

```
public interface MiInterfaz {  no usages
    // Declaración de métodos (sin implementación)
    void metodo1();  no usages
    int metodo2(String parametro);  no usages
}
```

Tipos de métodos

En Java, los métodos se pueden clasificar en varias categorías según su propósito y características. Estos son los principales tipos de métodos:

- **Según su tipo de retorno**

  Métodos con retorno: Devuelve un valor específico con la palabra clave return.

```java
// 6. Metodo con retorno
public int sumar(int a, int b) {  1 usage
    return a + b;
}
```

  Métodos void (sin retorno): No devuelven ningún valor.

```java
// 7. Metodo void (sin retorno)
public void imprimirMensaje() {  1 usage
    System.out.println("Hola, mundo!");
}
```

- **Según su pertenencia a la clase o instancia**

  Métodos de instancia:

  Requieren que se cree un objeto de la clase para ser llamados.

```java
// 4. Metodo de instancia (requiere un objeto)
public void metodoInstancia() {  1 usage
    System.out.println("Este es un método de instancia.");
}
```

Métodos estáticos:

Se asocian a la clase en lugar de a un objeto, y se llaman usando el nombre de la clase. Se definen con la palabra clave static.

```
// 5. Metodo estático (pertenece a la clase)
public static void metodoEstatico() {  1 usage
    System.out.println("Este es un método estático.");
}
```

- **Según su modificador de acceso**

Public: Accesible desde cualquier parte del programa.

Private: Solo accesible dentro de la misma clase.

Protected: Accesible dentro del mismo paquete y por clases hijas (herencia).

(Sin modificador, "default"): Solo accesible dentro del mismo paquete.

```
// 10. Metodos con diferentes niveles de acceso
private void metodoPrivado() {  no usages
    System.out.println("Método privado.");
}

protected void metodoProtegido() {  1 usage
    System.out.println("Método protegido.");
}

void metodoPorDefecto() { // Acceso "default"  1 usage
    System.out.println("Método con acceso por defecto.");
}
```

- **Según su finalidad**

Métodos constructores:

Se ejecutan al crear un objeto y no tienen ningún tipo de retorno.

```
// 2. Metodo constructor
private String nombre;  3 usages

public EjemploMetodos(String nombre) {  1 usage
    this.nombre = nombre;
}
```

Métodos abstractos:

Se declaran sin implementación en clases abstractas y deben ser implementados por sus subclases.

```
abstract class Animal {  1 usage  1 inheritor

    // 1. Metodo abstracto (debe ser implementado por una subclase)
    abstract void hacerSonido();  1 usage  1 implementation
}
```

Métodos finales (final):

No pueden ser sobrescritos en subclases.

```
// 9. Metodo final (no puede ser sobrescrito por subclases)
public final void metodoFinal() {  1 usage
    System.out.println("Este es un método final.");
}
```

Métodos sobrecargados (Overloading):

Se definen con el mismo nombre pero diferentes parámetros.

```java
// 8. Metodo sobrecargado (mismo nombre, diferentes parámetros)
public int multiplicar(int a, int b) {  1 usage
    return a * b;
}

public double multiplicar(double a, double b) {  1 usage
    return a * b;
}
```

Métodos sobrescritos (Overriding): Una subclase redefine un método de la superclase.

```java
class Perro extends Animal {  2 usages
    // Sobrescritura del metodo abstracto de Animal
    @Override  1 usage
    void hacerSonido() {
        System.out.println("Guau!");
    }
}
```

Métodos de acceso (getters) y modificación (setters):

En Java, los métodos "get" y "set" (también conocidos como "getters" y "setters") son una parte fundamental del principio de encapsulamiento en la programación orientada a objetos (POO). Su propósito principal es controlar el acceso a los atributos (variables) de una clase.

- *¿Qué son los Getters?*

  Los "getters" son métodos que se utilizan para obtener (leer) el valor de un atributo privado de una clase. Por convención, el nombre de un "getter" comienza con "get" seguido del nombre del atributo (por ejemplo, getNombre, getEdad). Su

función es proporcionar un acceso controlado a los datos, permitiendo que otras clases obtengan el valor del atributo sin acceder directamente a él.

- *¿Qué son los Setters?*

  Los "setters" son métodos que se utilizan para establecer (modificar) el valor de un atributo privado de una clase. Por convención, el nombre de un "setter" comienza con "set" seguido del nombre del atributo (por ejemplo, setNombre, setEdad). Su función es permitir la modificación controlada de los datos, lo que permite realizar validaciones o aplicar lógica antes de cambiar el valor del atributo.

```
// 3. Metodos de acceso (Getters y Setters)
public String getNombre() {  1 usage
    return nombre;
}

public void setNombre(String nombre) {  no usages
    this.nombre = nombre;
}
```

Los cuatro pilares de la POO

Para poder manejar de manera eficiente las clases y los objetos que se generan con la Programación Orientada a Objetos son necesarios algunos principios que nos ayudarán a reducir la complejidad, ser más eficientes y evitar problemas. Son los 4 pilares de la POO. Todos los lenguajes orientados a objetos los implementan de una u otra manera, y es indispensable conocerlos bien.

- **Encapsulamiento**

El encapsulamiento es un principio de la POO que se refiere a la ocultación de datos y métodos dentro de un objeto para protegerlos de cambios no deseados. En la POO, se utilizan clases para crear objetos. Una clase es una plantilla o un molde que define las propiedades y métodos de un objeto. Los datos y métodos dentro de una clase se mantienen privados y solo se pueden acceder a través de métodos públicos.

¿Por qué es importante el encapsulamiento?

El encapsulamiento es importante porque ayuda a mantener la integridad de los datos y a prevenir cambios no deseados. Los datos privados solo pueden ser modificados a través de métodos públicos, lo que asegura que se cumplan las reglas de negocio establecidas. Además, el encapsulamiento facilita la reutilización del código y hace que sea más fácil de mantener.

Ejemplo de encapsulamiento en POO

Un ejemplo de encapsulamiento en POO sería la creación de una clase «Empleado» con atributos privados como «nombre», «apellido», «salario» y métodos públicos como «obtenerSalario» y «aumentarSalario». Estos métodos públicos son los únicos que pueden acceder a los datos privados de la clase «Empleado», lo que asegura que los datos se mantengan protegidos.

```java
public class Empleado {  no usages
    // Atributos privados
    private String nombre;  3 usages
    private double salario;  3 usages

    // Constructor
    public Empleado(String nombre, double salario) {  no usages
        this.nombre = nombre;
        this.salario = salario;
    }

    // Métodos públicos para acceder y modificar los atributos
    public String getNombre() {  no usages
        return nombre;
    }

    public void setNombre(String nombre) {  no usages
        this.nombre = nombre;
    }

    public double getSalario() {  no usages
        return salario;
    }

    public void setSalario(double salario) {  no usages
        if (salario > 0) {
            this.salario = salario;
        }
    }
}
```

- **Herencia**

La herencia es otro concepto clave de la POO. La herencia se refiere a la capacidad de una clase para heredar propiedades y métodos de una clase padre. La clase

padre se conoce como la clase base o superclase, mientras que la clase que hereda de la superclase se conoce como la clase derivada o subclase.

¿Por qué es importante la herencia?

La herencia es importante porque permite la reutilización del código y la organización de las clases en una jerarquía. Las clases derivadas pueden heredar propiedades y métodos de la superclase, lo que reduce la cantidad de código que se necesita escribir y hace que el código sea más fácil de mantener.

Ejemplo de herencia en POO

Un ejemplo de herencia en POO sería la creación de una clase «Vehículo» con propiedades como «marca», «modelo» y «año», y métodos como «acelerar» y «frenar». A partir de esta clase base, se pueden crear clases derivadas como «Automóvil», «Motocicleta» y «Camión». Estas clases derivadas heredarán las propiedades y métodos de la clase «Vehículo», pero también pueden tener sus propias propiedades y métodos adicionales.

```java
// Superclase
public class Vehiculo {  no usages  1 inheritor
    protected String marca;  1 usage
    protected String modelo;  1 usage

    public Vehiculo(String marca, String modelo) {  no usages
        this.marca = marca;
        this.modelo = modelo;
    }

    public void acelerar() {  no usages
        System.out.println("El vehículo está acelerando.");
    }
}
```

```java
// Subclase
public class Automovil extends Vehiculo {  no usages
    private int numeroDePuertas;  2 usages

    public Automovil(String marca, String modelo, int numeroDePuertas) {  no usages
        super(marca, modelo);
        this.numeroDePuertas = numeroDePuertas;
    }

    public void mostrarDetalles() {  no usages
        System.out.println("Marca: " + marca + ", Modelo: " + modelo + ", Número de puertas: " + numeroDePuertas);
    }
}
```

- **Polimorfismo**

El polimorfismo es otro concepto importante de la POO. El polimorfismo se refiere a la capacidad de un objeto para tomar varias formas. En la POO, el polimorfismo se logra a través del uso de interfaces y clases abstractas.

¿Por qué es importante el polimorfismo?

El polimorfismo es importante porque permite que un objeto se comporte de manera diferente según el contexto en el que se utiliza. Esto permite una mayor flexibilidad en el diseño de programas y hace que el código sea más fácil de mantener.

Ejemplo de polimorfismo en POO

Un ejemplo de polimorfismo en POO sería la creación de una interfaz «Animal» con un método «hacerSonido». A partir de esta interfaz, se pueden crear clases como «Perro», «Gato» y «Vaca» que implementen la interfaz «Animal» y proporcionen una implementación única del método «hacerSonido».

```java
public abstract class Animal {  2 usages  2 inheritors
    public abstract void hacerSonido();  no usages  2 implementations
}
```

```java
public class Gato extends Animal {  no usages
    @Override  no usages
    public void hacerSonido() {
        System.out.println("El gato maúlla.");
    }
}
```

```java
public class Perro extends Animal {  no usages
    @Override  no usages
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}
```

- **Abstracción**

La abstracción es el último concepto básico de la POO que vamos a explorar. La abstracción se refiere a la capacidad de enfocarse en los aspectos esenciales de un objeto y ocultar los detalles no esenciales.

¿Por qué es importante la abstracción?

La abstracción es importante porque permite enfocarse en lo que es importante en un objeto y ocultar los detalles innecesarios. Esto hace que el código sea más fácil de entender y mantener.

Ejemplo de abstracción en POO

Un ejemplo de abstracción en POO sería la creación de una clase «Forma» con métodos abstractos como «calcularArea» y «calcularPerímetro». A partir de esta clase, se pueden crear clases derivadas como «Círculo», «Cuadrado» y «Rectángulo». Cada una de

estas clases derivadas proporcionará una implementación única de los métodos abstractos de la clase «Forma».

```
public abstract class Forma {  no usages  1 inheritor
    public abstract double calcularArea();  no usages  1 implementation
    public abstract double calcularPerimetro();  no usages  1 implementation
}
```

```
public class Circulo extends Forma {  no usages
    private double radio;  4 usages

    public Circulo(double radio) {  no usages
        this.radio = radio;
    }

    @Override  no usages
    public double calcularArea() {
        return Math.PI * radio * radio;
    }

    @Override  no usages
    public double calcularPerimetro() {
        return 2 * Math.PI * radio;
    }
}
```

Diagrama de clases

El diagrama de clases es uno de los diagramas incluidos en UML 2.5 clasificado dentro de los diagramas de estructura y, como tal, se utiliza para representar los elementos que componen un sistema de información desde un punto de vista estático.

Es importante destacar que, por esta misma razón, este diagrama no incluye la forma en la que se comportan a lo largo de la ejecución los distintos elementos, esta función puede ser representada a través de un diagrama de comportamiento, como por ejemplo un diagrama de secuencia o un diagrama de casos de uso.

El diagrama de clases es un diagrama puramente orientado al modelo de programación orientado a objetos, ya que define las clases que se utilizarán cuando se pase a la fase de construcción y la manera en que se relacionan las mismas.

- **Elementos de un diagrama de clases**

  El diagrama UML de clases está formado por dos elementos: clases, relaciones e interfaces.

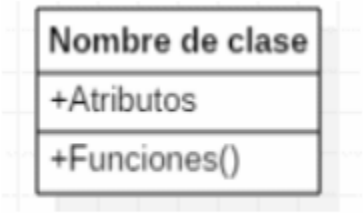
  Clases

  Las clases son el elemento principal del diagrama y representa, como su nombre indica, una clase dentro del paradigma de la orientación a objetos. Este tipo de elementos normalmente se utilizan para representar conceptos o entidades del «negocio». Una clase define un grupo de objetos que comparten características, condiciones y significado. La manera más rápida para encontrar clases sobre un enunciado, sobre una idea de negocio o, en general, sobre un tema concreto es buscar los sustantivos que aparecen en el mismo. Por poner algún ejemplo, algunas clases podrían ser: Animal, Persona, Mensaje, Expediente… Es un concepto muy amplio y resulta fundamental identificar de forma efectiva estas clases, en caso de no hacerlo correctamente se obtendrán una serie de problemas en etapas posteriores, teniendo que volver a hacer el análisis y perdiendo parte o todo el trabajo que se ha hecho hasta ese momento.

Bajando de nivel una clase está compuesta por tres elementos: nombre de la clase, atributos, funciones.

Para representar la clase con estos elementos se utiliza una caja que es dividida en tres zonas utilizando para ello líneas horizontales:

- La primera de ellas se utiliza para el nombre de la **clase**. En caso de que la clase sea abstracta se utilizará su nombre en cursiva.
- La segunda, por otra parte, se utiliza para escribir los **atributos** de la clase.
- La última de las zonas incluye cada una de las funciones que ofrece la clase.

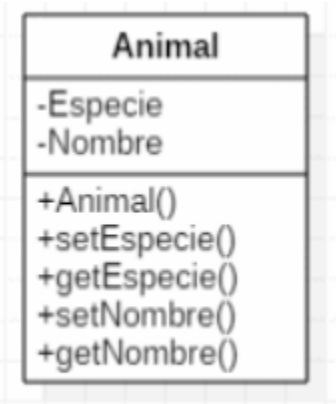


Notación de una clase

Tanto los atributos como las funciones incluyen al principio de su descripción la visibilidad que tendrá. Esta visibilidad se identifica escribiendo un símbolo y podrá ser:

- (+) Pública. Representa que se puede acceder al atributo o función desde cualquier lugar de la aplicación.
- (-) Privada. Representa que se puede acceder al atributo o función únicamente desde la misma clase.
- (#) Protegida. Representa que el atributo o función puede ser accedida únicamente desde la misma clase o desde las clases que hereden de ella (clases derivadas).

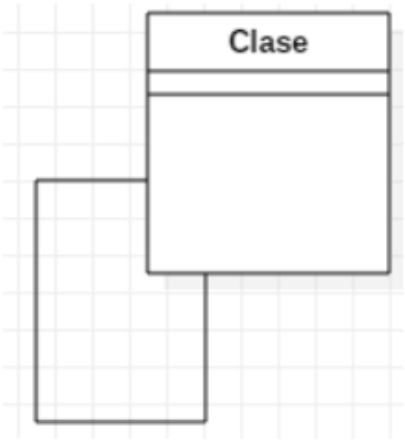
Estos tres tipos de visibilidad son los más comunes. No obstante, pueden incluirse otros en base al lenguaje de programación que se esté usando (no es muy común). Por ejemplo: (/) Derivado o (~) Paquete.



Ejemplo de una clase

Relaciones

Una relación identifica una dependencia. Esta dependencia puede ser entre dos o más clases (más común) o una clase hacía sí misma (menos común, pero existen), este último tipo de dependencia se denomina dependencia **reflexiva**. Las relaciones se representan con una línea que une las clases, esta línea variará dependiendo del tipo de relación.



Relación reflexiva

Las relaciones en el diagrama de clases tienen varias propiedades, que dependiendo la profundidad que se quiera dar al diagrama se representarán o no. Estas propiedades son las siguientes:

- *Multiplicidad.* Es decir, el número de elementos de una clase que participan en una relación. Se puede indicar un número, un rango… Se utiliza n o * para identificar un número cualquiera.
- *Nombre de la asociación.* En ocasiones se escribe una indicación de la asociación que ayuda a entender la relación que tienen dos clases. Suelen utilizarse verbos como por ejemplo: «Una empresa contrata a n empleados».

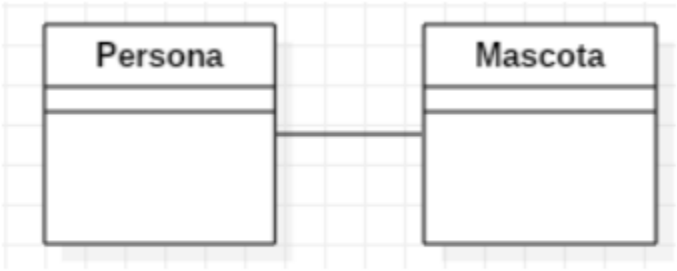
Tipos de relaciones

Un diagrama de clases incluye los siguientes tipos de relaciones:

- *Asociación*

  Este tipo de relación es el más común y se utiliza para representar dependencia semántica. Se representa con una simple línea continua que une las clases que están incluidas en la asociación.

  Un ejemplo de asociación podría ser: «Una mascota pertenece a una persona».



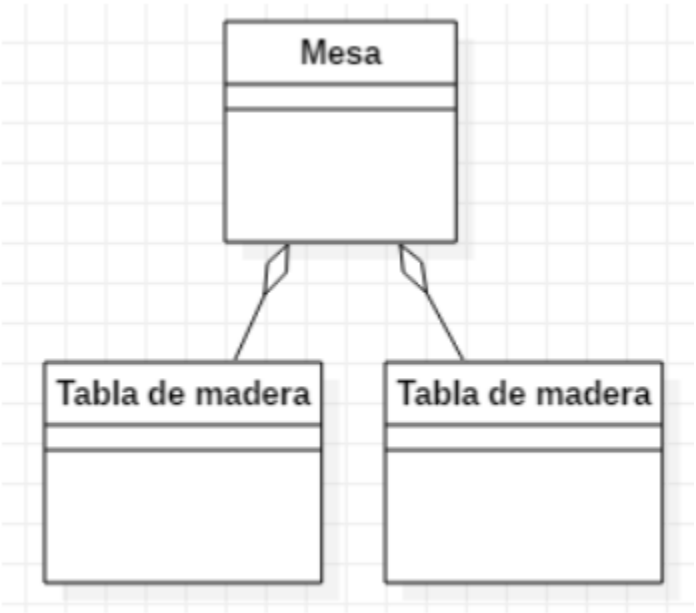
Ejemplo de asociación

- *Agregación.*

  Es una representación jerárquica que indica a un objeto y las partes que componen ese objeto. Es decir, representa relaciones en las que un objeto es parte de otro, pero aun así debe tener existencia en sí mismo.

  Se representa con una línea que tiene un rombo en la parte de la clase que es una agregación de la otra clase (es decir, en la clase que contiene las otras).

  Un ejemplo de esta relación podría ser: «Las mesas están formadas por tablas de madera y tornillos o, dicho de otra manera, los tornillos y las tablas forman parte de una mesa». Cómo ves, el tornillo podría formar parte de más objetos, por lo que interesa especialmente su abstracción en otra clase.



Ejemplo de agregación

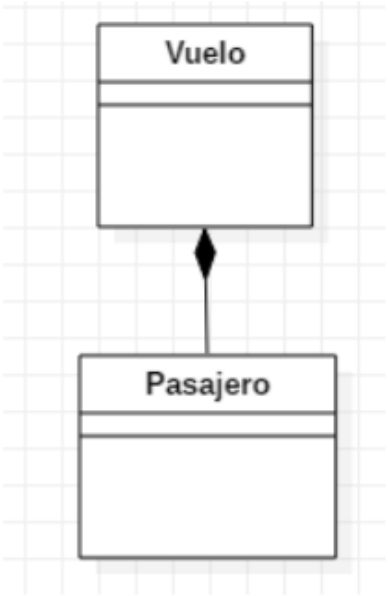
- *Composición.*

  La composición es similar a la agregación, representa una relación jerárquica entre un objeto y las partes que lo componen, pero de una forma más

fuerte. En este caso, los elementos que forman parte no tienen sentido de existencia cuando el primero no existe. Es decir, cuando el elemento que contiene los otros desaparece, deben desaparecer todos ya que no tienen sentido por sí mismos sino que dependen del elemento que componen. Además, suelen tener el mismo tiempo de vida. Los componentes no se comparten entre varios elementos, esta es otra de las diferencias con la agregación.

Se representa con una línea continua con un rombo relleno en la clase que es compuesta.

Un ejemplo de esta relación sería: «Un vuelo de una compañía aérea está compuesto por pasajeros, que es lo mismo que decir que un pasajero está asignado a un vuelo»

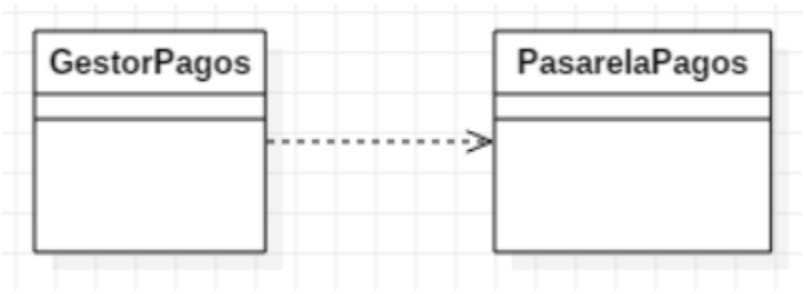


Ejemplo de composición

- *Dependencia.*

Se utiliza este tipo de relación para representar que una clase requiere de otra para ofrecer sus funcionalidades. Es muy sencilla y se representa con una

flecha discontinua que va desde la clase que necesita la utilidad de la otra flecha hasta esta misma.

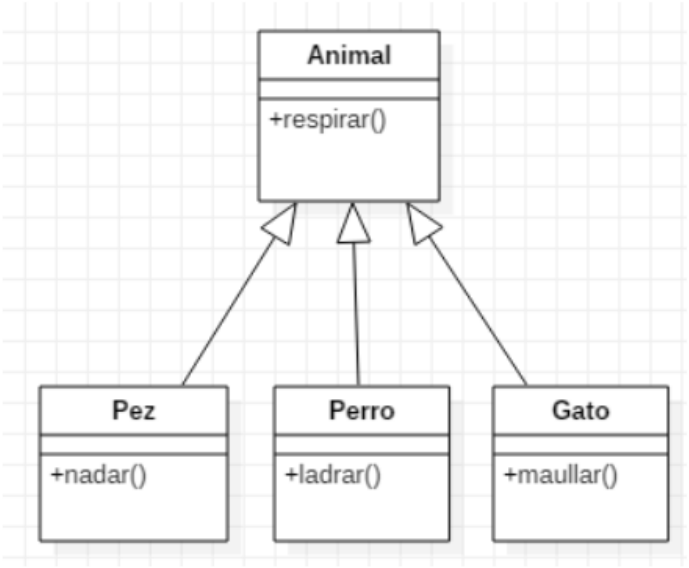


Ejemplo de dependencia

- *Herencia.*

Otra relación muy común en el diagrama de clases es la herencia. Este tipo de relaciones permiten que una clase (clase hija o subclase) reciba los atributos y métodos de otra clase (clase padre o superclase). Estos atributos y métodos recibidos se suman a los que la clase tiene por sí misma.

Un ejemplo de esta relación podría ser la siguiente: Un pez, un perro y un gato son animales.



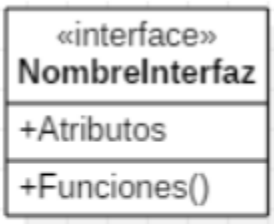
Ejemplo de herencia

Interfaces

Una interfaz es una entidad que declara una serie de atributos, funciones y obligaciones. Es una especie de contrato donde toda instancia asociada a una interfaz debe de implementar los servicios que indica aquella interfaz. Dado que únicamente son declaraciones no pueden ser instanciadas.

Las interfaces se asocian a clases. Una asociación entre una clase y una interfaz representa que esa clase cumple con el contrato que indica la interfaz, es decir, incluye aquellas funciones y atributos que indica la interfaz.

Su representación es similar a las clases, pero indicando arriba la palabra <<interface>>.



Notación de interfaz

- **Cómo dibujar un diagrama de clases**

Los diagramas de clase van de la mano con el diseño orientado a objetos. Por lo tanto, saber lo básico de este tipo de diseño es una parte clave para poder dibujar diagramas de clase eficaces.

Este tipo de diagramas son solicitados cuando se está describiendo la vista estática del sistema o sus funcionalidades. Unos pequeños pasos que puedes utilizar de guía para construir estos diagramas son los siguientes:

- ***Identifica los nombres de las clase***

El primer paso es identificar los objetos primarios del sistema. Las clases suelen corresponder a sustantivos dentro del dominio del problema.

- ***Distingue las relaciones***

  El siguiente paso es determinar cómo cada una de las clases u objetos están relacionados entre sí. Busca los puntos en común y las abstracciones entre ellos; esto te ayudará a agruparlos al dibujar el diagrama de clase.

- ***Crea la estructura***

  Primero, agrega los nombres de clase y vincúlalos con los conectores apropiados, prestando especial atención a la cardinalidad o las herencias. Deja los atributos y funciones para más tarde, una vez que esté la estructura del diagrama resuelta.

- **Herramientas gratuitas para desarrollar diagramas de clase**

  1. ***PlantUML***
     - *Descripción:* Herramienta basada en texto para crear diagramas UML mediante un lenguaje de marcado.
     - *Plataforma:* Windows, Linux, Mac (puede usarse en línea o con extensiones en VS Code, IntelliJ, Eclipse).
  2. ***StarUML (versión gratuita con limitaciones)***
     - *Descripción:* Potente herramienta para modelado UML. La versión gratuita permite usar la mayoría de sus funciones, pero muestra un aviso de compra.
     - *Plataforma:* Windows, Mac, Linux.
  3. ***Lucidchart (versión gratuita con limitaciones)***

- *Descripción:* Plataforma en línea para crear diagramas UML fácilmente mediante una interfaz drag-and-drop. La versión gratuita tiene un límite de diagramas.
- *Plataforma:* Navegador web.

4. ***Diagrams.net (Draw.io)***
    - *Descripción:* Herramienta web y de escritorio para diagramas de todo tipo, incluyendo UML. Permite exportar a varios formatos.
    - *Plataforma:* Web, Windows, Mac, Linux.
5. ***Mermaid.js***
    - *Descripción:* Similar a PlantUML, permite escribir código en Markdown para generar diagramas UML. Se integra con VS Code, GitHub, Notion, etc.
    - *Plataforma:* Web, VS Code, GitHub, Notion.
6. ***Modelio***
    - *Descripción:* Software de modelado UML con soporte para BPMN y otros estándares.
    - *Plataforma:* Windows, Mac, Linux.

Enlace de GitHub para acceder a los ejercicios:

https://github.com/SergioMoreno12/Ejemplos_POO

Porcentaje de IA utilizada: 30%

Conclusión

La programación en objetos en Java es un enfoque estructurado y efectivo para el desarrollo de software, que facilita el diseño de programas más ordenados y reutilizables. A lo largo de este trabajo, se han abordado los elementos principales de este paradigma, como clases, objetos, herencia, polimorfismo y encapsulamiento, que son los cimientos para construir aplicaciones modulares y escalables. Es importante comprender estos principios para poder aprovechar las ventajas que ofrece Java, un lenguaje muy empleado en muchos ámbitos de la industria del software por ser flexible y robusto.

En ese sentido, el estudio de la POO en Java no solo facilita el diseño de soluciones más efectivas, sino que también pone las bases para un código mejor y su mantenimiento a largo plazo. A medida que se avanza en dicho paradigma, se puede apreciar con claridad su importancia en la programación actual, ya que fomenta buenas prácticas y facilita el abordaje de problemas de manera más estructurada. Por eso, el conocimiento de estos aspectos es un requisito ineludible para cualquier programador.

Referencias

Alarcón Aguín, J. M. (2021, 26 de julio). Los conceptos fundamentales sobre Programación Orientada Objetos explicados de manera simple. *campusMVP.es*. https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx

Autor desconocido. (s.f.). Diagrama de clases. DiagramasUML.com. https://diagramasuml.com/diagrama-de-clases/

Galarza, P. (2024, 21 de marzo). Clases, objetos y el método constructor. *PauloGalarza.com*. https://paulogalarza.com/entendiendo-la-interaccion-entre-clases-objetos-y-el-metodo-constructor/

iammalf. (2023, 15 de febrero). Conceptos básicos de la programación orientada a objetos (POO). WebDesignCusco. https://webdesigncusco.com/conceptos-basicos-de-la-programacion-orientada-a-objetos-poo/

iKenshu. (2019, 21 de mayo). Qué es una clase abstracta en la programación orientada a objetos. *Platzi*. https://platzi.com/blog/clases-abstractas/

López Blasco, J. (2023, 27 de octubre). Introducción a POO en Java: Objetos y clases. *OpenWebinars*. https://openwebinars.net/blog/introduccion-a-poo-en-java-objetos-y-clases/

López Blasco, J. (2023, 22 de noviembre). Introducción a POO en Java: Interfaces y paquetes. OpenWebinars. https://openwebinars.net/blog/introduccion-a-poo-en-java-interfaces-y-paquetes/