Programación Orientada a Objetos (POO)

Kevin Santiago Santos Castellanos

Ingeniería de Sistemas

72663: Estructura de Datos

Profesor: William Alexander Matallana Porras

18 de marzo de 2025

Tabla de Contenido

1. Introducción

En esta investigación se buscar conocer que es la programación orientada a objetos (POO) , y sus principales fundamentos . Además , se busca analizar los conceptos esenciales , como son las clases, los diagramas de clases, los tipos de métodos y las interfaces .Esto con el fin de abordar los principios de la POO .

2.Objetivos

2.1 Objetivo general

Conocer la programación orientada a objetos y sus fundamentos

2.2 Objetivos Específicos

- Investigar que son las clases en java, e identificar qué tipo de clases existen
- Definir que son los diagramas de clases y explorar su estructura
- Comprender cuáles son los tipos de métodos en java
- Averiguar que son las interfaces en POO

3. ¿Qué es la programación orientada a objetos (POO)?

La programación orientada a objetos es un paradigma de programación que se centra en la creación de objetos y su interacción para resolver problemas de software. En POO, los objetos representan entidades del mundo real, como personas, animales o vehículos, y se organizan en clases que definen sus propiedades y comportamientos.

3.1 ¿Cuáles son los principios de la POO?

1. Herencia
2. Encapsulamiento
3. Polimorfismo
4. Abstracción

Encapsulamiento: protección de datos y ocultación de información

El encapsulamiento es el principio que permite proteger los datos y ocultar la información interna de un objeto, de manera que solo se pueda acceder a ellos a través de métodos públicos y seguros.

Los datos y métodos privados de un objeto no están disponibles para otros objetos, lo que garantiza la integridad de la información y evita que sea alterada o corrompida accidental o intencionalmente.

Cada objeto es responsable de su propia información y de su propio estado, la única forma de que la información se pueda modificar, es mediante los mismos métodos del objeto. por lo tanto, los atributos internos de un objeto deberían ser inaccesibles desde fuera, pudiéndolos modificar solo llamando a las funciones correspondientes.

Con esto conseguimos mantener a salvo el estado de usos indebidos que puedan ocasionar resultados inesperados.

Objetivo:

El objetivo principal de la encapsulación es proteger los datos de una clase del acceso directo desde el exterior, lo cual ayuda a mantener la integridad de los datos y a reducir la complejidad del sistema.

Implementación:

Para lograr la encapsulación, se utilizan modificadores de acceso como private, protected y public:

Private: Los miembros private de una clase solo son accesibles dentro de la misma clase. No pueden ser accedidos ni modificados directamente desde fuera de la clase.

Protected: Los miembros protected son accesibles dentro de la misma clase y por las clases derivadas (subclases).

Public: Los miembros public son accesibles desde cualquier parte del programa.

Ejemplo en Java:

```java
public class Persona {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

En este ejemplo, ***nombre*** y ***edad*** están encapsulados y solo pueden ser accedidos y modificados a través de los
métodos getNombre, setNombre, ***getEdad*** y setEdad.

2. Abstracción: simplificación y claridad de conceptos

La **abstracción** es el principio que permite simplificar y clarificar los conceptos y comportamientos complejos mediante la creación de modelos o representaciones abstractas y simplificadas.

Los modelos abstractos se centran en los aspectos esenciales y relevantes del objeto o sistema, y eliminan los detalles irrelevantes o confusos. De esta manera, se simplifica el diseño y se mejora la comprensión y mantenibilidad del programa.

Este principio se puede definir como las características específicas el objeto, los mismos que lo distinguen de los demás tipos de objetos,

La abstracción consiste en separar un elemento de su contexto o del resto de elementos que lo acompañan. Es un principio por el cual se descarta toda aquella información que no resulta relevante en un contexto en particular, enfatizando en alguno de los detalles o propiedades de los objetos. Depende principalmente del interés del observador, permitiendo abstracciones muy diferentes de la misma realidad.

Objetivo:

El objetivo de la abstracción es permitir que los programadores manejen la complejidad ocultando los detalles innecesarios y mostrando solo la información relevante.

Implementación:

En la POO, la abstracción se puede lograr mediante el uso de clases abstractas e interfaces.

Ejemplo en Java:

```java
abstract class Animal {
    abstract void hacerSonido();
}

class Perro extends Animal {
    void hacerSonido() {
        System.out.println("Guau");
    }
}

class Gato extends Animal {
    void hacerSonido() {
        System.out.println("Miau");
    }
}
```

En este ejemplo, ***Animal*** es una clase abstracta que tiene un método abstracto ***hacerSonido***. Las clases ***Perro*** y ***Gato*** implementan este método proporcionando su propia versión.

3. Herencia: reutilización de código y especialización de objetos

La **herencia** es el principio que permite crear nuevas clases a partir de clases existentes, reutilizando el código y los comportamientos de sus ancestros. La nueva clase se conoce como subclase o derivada, mientras que la clase original se llama superclase o base.

La herencia permite crear objetos especializados a partir de objetos más generales, y añadir o modificar sus propiedades y comportamientos de manera independiente. Además, reduce la duplicación de código y aumenta la eficiencia y la legibilidad del programa.

Objetivo:

El objetivo principal de la herencia es promover la reutilización del código y establecer una relación jerárquica entre las clases.

Implementación:

La herencia se implementa utilizando la palabra clave extends en Java.

Ejemplo en Java:

```java
class Animal {
    void comer() {
        System.out.println("Este animal come");
    }
}

class Perro extends Animal {
    void ladrar() {
        System.out.println("Guau");
    }
}

public class Main {
    public static void main(String[] args) {
        Perro perro = new Perro();
        perro.comer(); // Heredado de Animal
        perro.ladrar();
    }
}
```

En este ejemplo, la clase ***Perro*** hereda el método ***comer*** de la clase ***Animal***.

4. Polimorfismo: flexibilidad y compatibilidad de objetos

En informática el polimorfismo es considerado uno de los elementos más importantes dentro la programación orientada a objetos POO, ya que su definición está fuertemente ligada a la Herencia.

El **polimorfismo** es el principio que permite a los objetos responder de diferentes maneras a un mismo mensaje o método, según su tipo o contexto. Es decir, un objeto puede comportarse de forma distinta según la situación, sin necesidad de conocer su tipo específico.

El polimorfismo aumenta la flexibilidad y la compatibilidad de los objetos, ya que permite interactuar con ellos de manera genérica y predecible, sin tener que conocer los detalles internos de su implementación. También permite extender y modificar las funcionalidades del programa de forma modular y escalable.

Como definición el polimorfismo es la habilidad de un objeto de realizar una acción de diferentes maneras, utilizando métodos iguales que se implementen de forma diferente en varias clases.

Objetivo:

El objetivo del polimorfismo es permitir que una interfaz única controle el acceso a una clase genérica de acciones. Esto facilita la extensibilidad y la mantenibilidad del código.

Implementación:

El polimorfismo se puede lograr mediante la sobrecarga de métodos y la sobrescritura de métodos.

Ejemplo en Java:

```java
class Animal {
    void hacerSonido() {
        System.out.println("Sonido de animal");
    }
}

class Perro extends Animal {
    void hacerSonido() {
        System.out.println("Guau");
    }
}

class Gato extends Animal {
    void hacerSonido() {
        System.out.println("Miau");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro();
        miAnimal.hacerSonido(); // Guau

        miAnimal = new Gato();
        miAnimal.hacerSonido(); // Miau
    }
}
```

En este ejemplo, ***miAnimal*** es de tipo ***Animal***, pero puede referenciar a objetos de tipo ***Perro*** y ***Gato***. Dependiendo del tipo del objeto, se llama al método ***hacerSonido*** correspondiente.

4. ¿Qué son las clases en Java?

Una clase en Java se puede entender como un prototipo que define las variables y los métodos comunes a un cierto tipo de instancias, una clase define todo lo que caracteriza y pueden hacer una o varias instancias.

En java las clases son las matrices de las que luego se pueden crear múltiples instancias del mismo tipo. La clase define las variables y los métodos comunes a las instancias de ese tipo (el tipo de la clase creada), pero luego, cada instancia de esta clase tendrá sus propios valores (su propio molde, color y receta) y compartirán las mismas funciones.

Estructura básica de una clase en Java

```java
//Le damos un nombre "MiClase" a la clase
public class MiClase
{
    //Atributos de la clase
    private String atributo1;
    private int atributo2;
    private float atributo3;

    //Constructor con el mismo nombre de la clase
    public MiClase(){}

    //Métodos de la clase
    public void metodo1()
    {
        //Método vacio
    }

    public String metodo2()
    {
        return "metodo2";
    }
}
```

En el ejemplo anterior hemos creado una clase en Java llamada "MiClase" la cual posee un total de tres atributos (todos ellos privados) y son de tipo String, int y float respectivamente. Adicionalmente esta clase tiene un constructor (que siempre por norma, debe tener el mismo nombre de la clase) el cual no recibe ningún parámetro, aunque pueden recibir todos los parámetros que nosotros deseemos, también tiene un método llamado "metodo1" que no retorna valor alguno (es de tipo void) y otro método llamado "metodo2" que retorna una cadena de caracteres (es de tipo String) con el valor "metodo2". Cabe resaltar que una clase en Java puede

tener o no métodos y atributos, sin embargo lo más normal en la mayoría de los casos es que tenga tanto métodos como atributos que la caractericen.

4.1 ¿Cuáles son las clases abstractas?

Una clase abstracta es una clase que no se puede instanciar directamente y se utiliza como base para otras clases. Una clase abstracta puede tener métodos abstractos, que son métodos que no tienen una implementación y deben ser implementados por las clases que heredan de la clase abstracta. Además de los métodos abstractos, las clases abstractas también pueden tener métodos concretos, que son métodos que tienen una implementación.

Por ejemplo, podemos tener una clase abstracta llamada “Animal” con un método abstracto llamado “hacerRuido()“. Luego, podemos tener clases concretas como “Perro” y “Gato” que heredan de la clase “Animal” y proporcionan implementaciones específicas para el método “hacerRuido()“.

```java
abstract class Animal {

  // Método abstracto
  public abstract void hacerRuido();

  // Método concreto
  public void dormir() {
    System.out.println("El animal está durmiendo.");
  }
}
```

A lo que nosotros podríamos usar la clase abstracta para crear clases hijas de nuestra clase abstracta Animal, en este caso Perro y Gato:

```java
class Perro extends Animal {

  // Implementación del método abstracto hacerRuido()
  public void hacerRuido() {
    System.out.println("Guau!");
  }
}

class Gato extends Animal {

  // Implementación del método abstracto hacerRuido()
  public void hacerRuido() {
    System.out.println("Miau!");
  }
}
```

class Perro extends Animal {

En este ejemplo, tenemos una clase abstracta “Animal” con un método abstracto “hacerRuido()” y un método concreto “dormir()“. Luego tenemos dos clases concretas “Perro” y “Gato” que heredan de “Animal” y proporcionan implementaciones específicas para el método “hacerRuido()“.

Tenga en cuenta que no podemos instanciar la clase “Animal” directamente, ya que es una clase abstracta. En su lugar, debemos utilizar una de las clases concretas que heredan de “Animal“, como “Perro” o “Gato“.

```
Animal perro = new Perro();
perro.hacerRuido(); // Imprime "Guau!"
perro.dormir(); // Imprime "El animal está durmiendo."

Animal gato = new Gato();
gato.hacerRuido(); // Imprime "Miau!"
gato.dormir(); // Imprime "El animal está durmiendo."
```

5. ¿Qué es un diagrama de clases?

Son una herramienta fundamental en el diseño de software, especialmente cuando trabajamos con programación orientada a objetos (POO). Estos diagramas nos permiten visualizar la estructura de un sistema, representando las clases, sus atributos, métodos y las relaciones entre ellas. Son parte esencial del lenguaje UML (Unified Modeling Language) y son utilizados por desarrolladores, arquitectos de software y equipos de diseño para planificar y comunicar ideas de manera clara y eficiente.

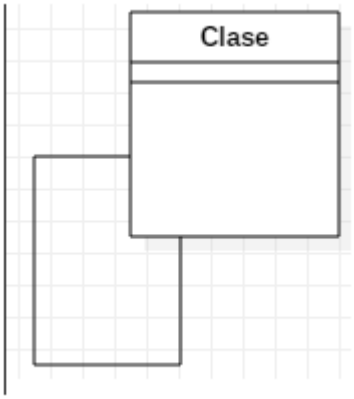
5.1 ¿Cuál es su estructura?

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

5.2 ¿Cómo se relacionan?

Relaciones

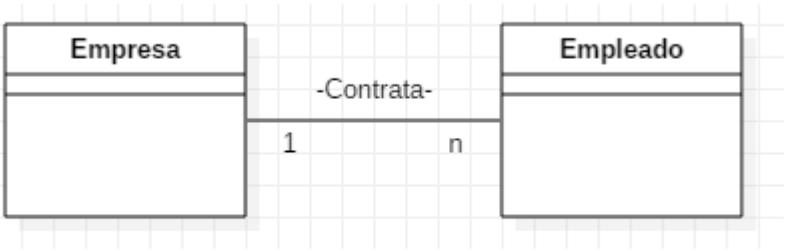
Una relación i**dentifica una dependencia**. Esta dependencia puede ser entre dos o más clases (más común) o una clase hacía sí misma (menos común, pero existen), este último tipo de dependencia se denomina *dependencia reflexiva*. Las relaciones se representan con una linea que une las clases, esta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



Ejemplo de relación Empresa-Empleado

Tipos de relaciones

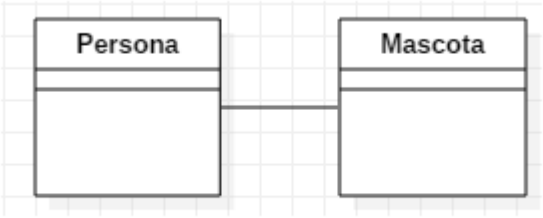
Un diagrama de clases incluye los siguientes tipos de relaciones:

- **Asociación.**
- **Agregación.**
- **Composición.**
- **Dependencia.**
- **Herencia.**

Asociación

Este tipo de relación es el más común y se utiliza para representar dependencia semántica. Se representa con una simple linea 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

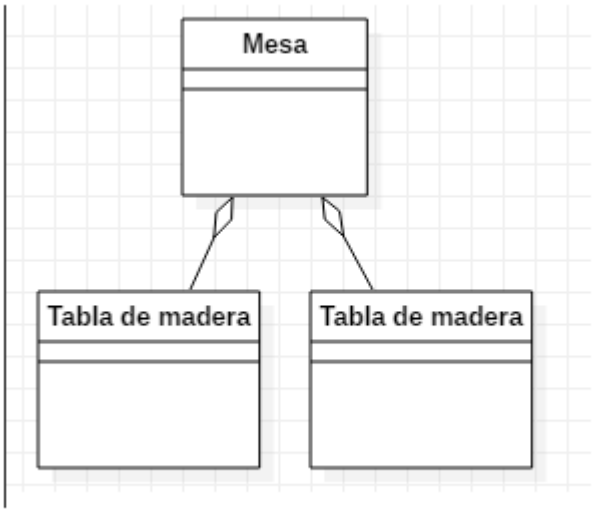
Agregación

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». Como 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 los mismos tiempo de vida. Los componentes no se comparten entre varios elementos, esta es otra de las diferencias con la agregación.

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

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

Diferencia entre agregación y composición

La diferencia entre agregación y composición es semántica, por lo que **a veces no está del todo definida**. Ninguna de las dos tienen análogos en muchos lenguajes de programación (como por ejemplo Java).

Un «agregado» representa **un todo que comprende varias partes**; de esta manera, un Comité es un agregado de sus Miembros. Una reunión es un agregado de una agenda, una sala y los asistentes. En el momento de la implementación, esta relación no es de contención. (Una reunión no contiene una sala). Del mismo modo, las partes del agregado podrían estar haciendo otras cosas en otras partes del programa, por lo que podrían ser referenciadas por varios objetos que nada tienen que ver. En otras palabras, no existe una diferencia de nivel de implementación entre la agregación y una simple relación de «usos». En ambos casos, un objeto tiene referencias a otros objetos. Aunque no existe una diferencia en la implementación, definitivamente vale la pena

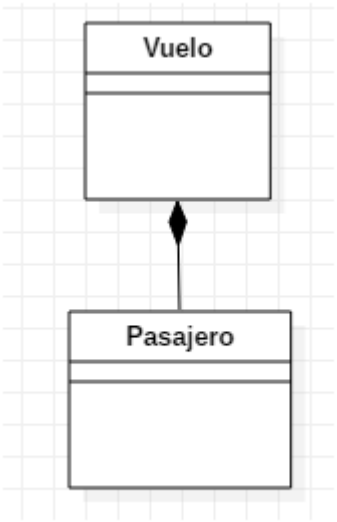
capturar la relación en el diagrama UML, tanto porque ayuda a comprender mejor el modelo de dominio, como porque puede haber problemas de implementación que pueden pasar desapercibidos. Podría permitir relaciones de acoplamiento más estrictas en una agregación de lo que haría con un simple «uso», por ejemplo.

La composición, por otro lado, implica u**n acoplamiento aún más estricto que la agregación**, y definitivamente implica la contención. El requisito básico es que, si una clase de objetos (llamado «contenedor») se compone de otros objetos (llamados «elementos»), entonces los elementos aparecerán y también serán destruidos como un efecto secundario de crear o destruir el contenedor. Sería raro que un elemento no se declare como privado. Un ejemplo podría ser el nombre y la dirección del Cliente. Un cliente sin nombre o dirección no tiene valor. Por la misma razón, cuando se destruye al cliente, no tiene sentido mantener el nombre y la dirección. (Compare esta situación con la agregación, donde destruir al Comité no debe causar la destrucción de los miembros, ya que pueden ser miembros de otros Comités).

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.

Un ejemplo de esta relación podría ser la siguiente:

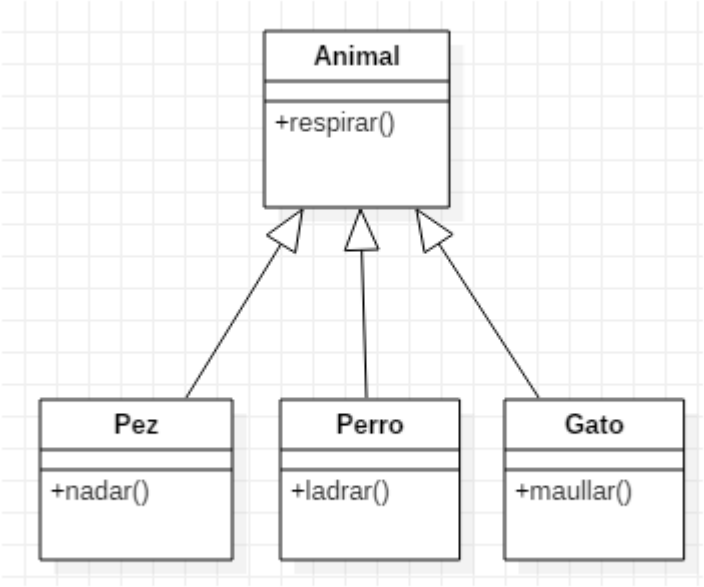


Ejemplo de composición

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. Se utiliza en relaciones «es un».

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



Ejemplo de herencia

En este ejemplo, las tres clases (Pez, Perro, Gato) podrán utilizar la función respirar, ya que lo heredan de la clase animal, pero solamente la clase Pez podrá nadar, la clase Perro ladrar y la clase Gato maullar. La clase Animal podría plantearse ser definida abstracta, aunque no es necesario.

5.3 ¿Qué herramientas gratuitas sirven para construir un diagrama de clases?

Visual Paradigm Online: Herramienta basada en la web para crear diagramas UML fácilmente.

Creately: Ofrece plantillas y una interfaz intuitiva para diagramas UML.

GitMind: Plataforma en línea con plantillas personalizables para diagramas UML.

UMLet: Software de código abierto para diagramas UML, ideal para principiantes.

6. Tipos de métodos en java

En la programación en Java, los métodos son bloques de código que se pueden llamar y ejecutar en cualquier momento dentro de un programa. Son fundamentales para dividir el código en pequeñas unidades lógicas y reutilizables, lo que facilita el desarrollo y el mantenimiento del software.

Existen diferentes tipos de métodos en Java:

1. Métodos estáticos: Son métodos que pertenecen a una clase en lugar de una instancia de la clase. Se pueden invocar utilizando el nombre de la clase seguido de un punto. Por ejemplo: `NombreClase.metodoEstatico()`.

2. Métodos no estáticos: Son métodos que pertenecen a una instancia de una clase y se invocan utilizando dicha instancia. Por ejemplo: `objeto.metodoNoEstatico()`.

3. Métodos de retorno: Son aquellos métodos que devuelven un valor. Para ello, se utiliza la palabra clave `return` seguida del valor que se desea retornar. Por ejemplo: `public int metodoDeRetorno() { return 5; }`.

4. Métodos sin retorno: Son métodos que no devuelven ningún valor y se definen utilizando la palabra clave `void`. Por ejemplo: `public void metodoSinRetorno() { // código }`.

5. Métodos con parámetros: Son métodos que reciben valores (parámetros) necesarios para realizar su tarea. Los parámetros se definen entre paréntesis después del nombre del método. Por ejemplo: `public void metodoConParametros(int num1, int num2) { // código }`.

6. **Métodos sobrecargados:** Son varios métodos con el mismo nombre pero con diferente lista de parámetros. Java permite utilizar el mismo nombre de método siempre y cuando los parámetros sean diferentes. Esto se conoce como sobrecarga de métodos.

7. Métodos constructores: Son métodos especiales que se utilizan para crear e inicializar objetos. Se llaman automáticamente al crear una instancia de una clase utilizando la palabra clave `new`. Por ejemplo: `public NombreClase() { // código }`.

7. ¿Qué son interfaces?

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 {
    // Declaración de métodos (sin implementación)
    void metodo1();
    int metodo2(String parametro);
}
```

En este ejemplo utilizamos la palabra clave interface para declarar la interfaz MiInterfaz. La interfaz no contiene implementaciones de métodos, solo declara los nombres y las firmas de los métodos (metodo1 y metodo2 en este caso).

Las interfaces también pueden contener constantes, las cuales son implícitamente public, static y final.

```
public interface OtraInterfaz {
    // Declaración de constantes
    int CONSTANTE1 = 42;
    String CONSTANTE2 = "Hola";

    // Declaración de métodos (sin implementación)
    void metodo();
}
```

Implementación de interfaces en clases

Para implementar una interfaz en una clase, se utiliza la palabra clave implements. La clase que implementa una interfaz debe proporcionar implementaciones para todos los métodos declarados en la interfaz.

Supongamos que tenemos una interfaz llamada MiInterfaz con dos métodos (metodo1 y metodo2), para crear una clase llamada MiClase que implementa esta interfaz procederíamos de la siguiente manera

Hemos utilizado la palabra clave implements para indicar que la clase implementa la interfaz MiInterfaz, además la clase proporciona implementaciones para los métodos definidos en la interfaz.

8. Conclusiones

- La POO permite a los desarrolladores construir aplicaciones de manera más eficiente al facilitar la reutilización de código y la creación de componentes modulares, lo que acelera el proceso de desarrollo.
- La POO facilita el mantenimiento y la actualización de aplicaciones. Los cambios en una parte del programa pueden implementarse con un impacto mínimo en otras partes del sistema.
- La POO ofrece un enfoque más natural para modelar problemas del mundo real, permitiendo que los programadores representen realidades complejas de manera más intuitiva a través de objetos y clases.
- Los principios de la POO, como la herencia y el polimorfismo, permiten que los sistemas sean más adaptables a los cambios, lo que es crucial en un entorno tecnológico en constante evolución.
- Al utilizar diagramas de clases y otros modelos visuales, los equipos de desarrollo pueden comunicarse más eficazmente sobre la estructura y funcionalidad del software, lo que mejora la colaboración y reduce el riesgo de errores durante el desarrollo.

9. Webgrafía

1. Programarya. (2020). Objetos, clases y constructores en Java. Crear una clase y un objeto. Class y new en Java. Recuperado de https://www.programarya.com/Cursos/Java/Objetos-y-Clases

2. WebDesignCusco. (2019). Los 4 principios fundamentales de la programación orientada a Objetos POO. Recuperado de https://webdesigncusco.com/los-4-principios-fundamentales-de-la-programacion-orientada-a-objetos-poo/

3. UML. (2021). Diagramas de clases. Recuperado de https://diagramasuml.com/diagrama-de-clases/

4. Programando en Java. (2023). Clases abstractas en Java. Recuperado de https://programandoenjava.com/clases-abstractas-en-java/

5. OpenWebinars. (2018). Introducción a POO en Java: Interfaces y paquetes. Recuperado de https://openwebinars.net/blog/introduccion-a-poo-en-java-interfaces-y-paquetes/

6. JMJ Informático. (2025). Tipos de Métodos en Java: Guía completa para principiantes. Recuperado de https://jmjinformatico.es/tipos-de-metodos-en-java/