Consulta - Programación orientada a objetos

Abstracción

En la POO, la abstracción es el proceso de enfocarse en las características esenciales de un objeto o concepto mientras se ignoran los detalles irrelevantes. Permite crear modelos conceptuales simples y fáciles de entender de objetos complejos, ocultando la complejidad interna y mostrando solo la funcionalidad que es relevante para el usuario.

Beneficios de la abstracción:

Simplifica el código: Al ocultar los detalles de implementación, la abstracción hace que el código sea más fácil de escribir, leer y mantener.
Promueve la reutilización: Los modelos abstractos se pueden reutilizar en diferentes partes del programa, lo que reduce la redundancia y mejora la organización del código.
Mejora la modularidad: La abstracción permite dividir el código en módulos independientes, lo que facilita el desarrollo y mantenimiento de software a gran escala.
Enmascara la complejidad: La abstracción protege a los usuarios de los detalles internos complejos de un objeto, permitiéndoles interactuar con él de una manera simple e intuitiva.
Ejemplo cotidiano de abstracción:

Imagina un televisor. Cuando interactúas con un televisor, no necesitas saber cómo funcionan los componentes electrónicos internos, como los transistores y los procesadores. Simplemente presiona los botones en el control remoto o usa la interfaz táctil para cambiar de canal, ajustar el volumen o ver una película. La abstracción del televisor te permite enfocarte en su función principal, que es mostrar imágenes y sonido, sin preocuparte por la complejidad subyacente.

En términos de POO, el televisor podría representarse como una clase con métodos para encender, apagar, cambiar de canal, ajustar el volumen y reproducir películas. Los detalles de implementación de estos métodos se ocultaron dentro de la clase, y el usuario solo necesitará interactuar con los métodos expuestos, como si estuviera usando el control remoto del televisor.

La abstracción es un concepto fundamental en la POO que permite crear software modular, reutilizable y fácil de mantener. Al enfocarse en las características esenciales de los objetos y ocultar los detalles irrelevantes, la abstracción simplifica el desarrollo de software y facilita su uso para los usuarios.

```java
Pago pago = new PagoTarjetaCredito(100.0, "USD");
pago.procesar();

pago = new PagoPayPal(50.0, "EUR");
pago.procesar();

pago = new PagoTransferenciaBancaria(200.0, "COP");
pago.procesar();
```

Encapsulamiento

El encapsulamiento es un pilar fundamental de la POO que se refiere a la agrupación de datos (atributos) y métodos (operaciones) que actúan sobre esos datos dentro de una unidad única llamada objeto.

La idea central del encapsulamiento es ocultar los detalles internos de un objeto y exponer solo una interfaz pública a través de la cual los usuarios pueden interactuar con él. Esto se logra mediante modificadores de acceso, que controlan el nivel de visibilidad de los atributos y métodos de una clase.

Beneficios del encapsulamiento:

Protección de datos: El encapsulamiento protege los datos internos de un objeto de accesos o modificaciones no deseados, promoviendo la integridad y seguridad del código.
Modularidad: Al agrupar datos y métodos relacionados, el encapsulamiento mejora la modularidad del código, haciéndolo más fácil de entender, mantener y reutilizar.
Ocultar complejidad: El encapsulamiento permite ocultar la complejidad interna de un objeto, exponiendo solo la funcionalidad necesaria para los usuarios, lo que facilita su uso.
Implementación de abstracción: El encapsulamiento es la base para implementar la abstracción en POO, ya que permite ocultar los detalles de implementación y enfocarse en las características esenciales de un objeto.
Ejemplo de encapsulamiento:

Imagina una clase CuentaBancaria que representa una cuenta bancaria real. La clase tendría atributos privados como saldo, titular y numeroCuenta, y métodos públicos como depositar, retirar y consultarSaldo.

```java
public class CuentaBancaria {

    private double saldo;
    private String titular;
    private String numeroCuenta;

    public void depositar(double cantidad) {
        if (cantidad > 0) {
            saldo += cantidad;
        }
    }

    public void retirar(double cantidad) {
        if (cantidad > 0 && saldo >= cantidad) {
            saldo -= cantidad;
        }
    }

    public double consultarSaldo() {
        return saldo;
    }
}
```

En este ejemplo, los atributos saldo, titular y numeroCuenta son privados, lo que significa que solo son accesibles desde dentro de la clase CuentaBancaria. Los métodos depositar, retirar y consultarSaldo son públicos, lo que significa que pueden ser llamados desde fuera de la clase.

El encapsulamiento asegura que los usuarios solo puedan interactuar con la cuenta bancaria a través de sus métodos públicos, protegiendo los datos internos y garantizando un uso seguro y controlado.

La herencia

La herencia es uno de los pilares fundamentales de la POO que permite crear jerarquías de clases donde una clase (clase hija) hereda los atributos y métodos de otra clase (clase padre). Esto facilita la reutilización de código, la organización del código y la extensibilidad del software.

Características de la herencia:

Relación jerárquica: Las clases se organizan en una jerarquía donde una clase puede heredar de otra, formando una estructura de árbol.
Reutilización de código: La herencia permite reutilizar atributos y métodos de la clase padre en la clase hija, reduciendo la redundancia y mejorando la eficiencia del código.
Extensibilidad: La herencia permite extender la funcionalidad de una clase padre creando clases hijas que agregan nuevos atributos o métodos, o modifican los existentes.
Polimorfismo: La herencia es la base para implementar el polimorfismo, que permite a los objetos de diferentes clases responder al mismo mensaje de manera diferente.
Tipos de herencia:

Herencia simple: Una clase hija hereda de una sola clase padre.
Herencia múltiple: Una clase hija hereda de dos o más clases padre.
Herencia jerárquica: Las clases se organizan en una estructura de árbol donde cada clase puede tener una o más clases hijas.
Herencia multirruta: Una clase hija puede heredar de múltiples clases padre a través de diferentes rutas de herencia.
Ejemplo de herencia:
Imagina una jerarquía de clases para representar animales:

```
class Animal {

    private String nombre;
    private int edad;

    public void comer() {
        System.out.println("El animal está comiendo");
    }

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

    // Getters and setters for nombre and edad
}

class Mamifero extends Animal {

    private String tipoPelo;
```

En este ejemplo, la clase Perro hereda de las clases Mamifero y Animal. Esto significa que Perro tiene todos los atributos y métodos de Mamifero y Animal, además de sus propios atributos y métodos específicos (raza y ladrar).

La herencia permite crear clases más específicas y reutilizar código de manera eficiente. En este caso, la clase Perro se beneficia de la funcionalidad proporcionada por las clases Mamifero y Animal, sin necesidad de reescribir el código para esas características comunes.

Polimorfismo

El polimorfismo es un concepto fundamental de la POO que permite a los objetos de diferentes clases responder al mismo mensaje de manera diferente. Esto se logra gracias a la vinculación dinámica, que asocia el mensaje a un método específico en tiempo de ejecución, dependiendo del tipo real del objeto que recibe el mensaje.

Características del polimorfismo:

Múltiples interfaces: El polimorfismo permite que diferentes clases implementen la misma interfaz, proporcionando diferentes implementaciones para el mismo comportamiento.
Vinculación dinámica: El método que se invoca en respuesta a un mensaje se determina en tiempo de ejecución en función del tipo real del objeto, no del tipo de la variable que lo referencia.
Mensajes abstractos: El polimorfismo se basa en el uso de mensajes abstractos, que representan acciones o comportamientos generales que pueden ser implementados de manera diferente por diferentes clases.
Herencia: La herencia es la base para implementar el polimorfismo, ya que permite crear jerarquías de clases donde las clases hijas pueden redefinir los métodos heredados de la clase padre.
Tipos de polimorfismo:

Polimorfismo de instancia: Ocurre cuando objetos de diferentes clases responden al mismo mensaje de manera diferente debido a su tipo concreto.
Polimorfismo de subtipo: Es una forma de polimorfismo de instancia en el que las clases hijas heredan y redefinen métodos de la clase padre.
Polimorfismo de sobrecarga de métodos: Ocurre cuando una clase define varios métodos con el mismo nombre pero diferentes parámetros o tipos de retorno.

Polimorfismo de interfaz: Ocurre cuando las clases implementan interfaces que definen métodos abstractos, permitiendo que los objetos sean tratados de manera uniforme independientemente de su tipo concreto.
Ejemplo de polimorfismo:

Imagina una clase abstracta Forma que representa una forma geométrica y tiene un método abstracto calcularArea(). Luego, crea clases concretas como Circulo, Rectangulo y Triangulo que heredan de Forma y redefinen el método calcularArea() para calcular el área específica de cada forma.

```
abstract class Forma {

    public abstract double calcularArea();
}

class Circulo extends Forma {

    private double radio;

    @Override
    public double calcularArea() {
        return Math.PI * radio * radio;
    }

    // Getter and setter for radio
}

class Rectangulo extends Forma {

    private double base;
    private double altura;

    @Override
    public double calcularArea() {
        return base * altura;
    }

    // Getters and setters for base and altura
}

class Triangulo extends Forma {

    private double base;
    private double altura;

    @Override
    public double calcularArea() {
        return (base * altura) / 2;
    }

    // Getters and setters for base and altura
}
```

En este ejemplo, las clases Circulo, Rectangulo y Triangulo son polimórficas porque responden al mismo mensaje calcularArea() de manera diferente, cada una

implementando la lógica específica para calcular el área de su forma correspondiente.

El polimorfismo es un concepto poderoso en POO que permite crear código flexible, reutilizable y mantenible. Al permitir a los objetos responder a los mismos mensajes de manera diferente, el polimorfismo promueve la abstracción y facilita el desarrollo de aplicaciones robustas y extensibles.

Clases y objetos

Clases y objetos en la programación orientada a objetos (POO)
Clases:

En POO, una clase es un modelo o plantilla que define la estructura y el comportamiento de un grupo de objetos relacionados. Es como un plano arquitectónico que describe las características comunes de un conjunto de entidades.

Las clases se componen de dos elementos principales:

Atributos: También conocidas como variables de clase, representan las propiedades o características que definen el estado de un objeto. Por ejemplo, en una clase Coche, los atributos podrían ser color, marca, modelo y velocidadMaxima.

Métodos: También conocidas como funciones de clase, representan el comportamiento o las acciones que un objeto puede realizar. Por ejemplo, en una clase Coche, los métodos podrían ser arrancar(), acelerar(), frenar() y girar().

Objetos:

Un objeto es una instancia concreta de una clase, es decir, una entidad individual que posee los atributos y métodos definidos en la clase. Se crea a partir de la clase utilizando un proceso llamado instanciación.

Los objetos encapsulan datos (atributos) y comportamiento (métodos), permitiendo modelar entidades del mundo real de manera precisa y organizada.

Relación entre clases y objetos:

Clases como plantillas: Las clases sirven como plantillas para crear objetos, definiendo la estructura y el comportamiento que todos los objetos de esa clase deben compartir.

Objetos como instancias: Los objetos son instancias concretas de una clase, representando entidades individuales con valores específicos para sus atributos y métodos.

Creación de objetos: Los objetos se crean a partir de clases mediante el proceso de instanciación, utilizando operadores específicos en cada lenguaje de programación.

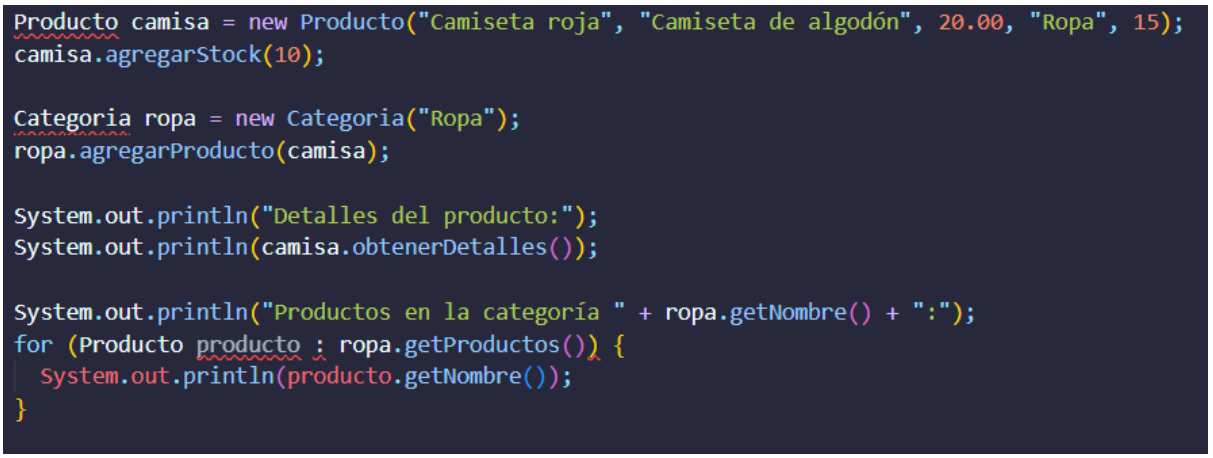
Ejemplo de clases y objetos:

Imagina una clase Estudiante que representa a un estudiante en una universidad. La clase podría tener atributos como nombre, apellido, codigo, carrera y promedio.

Los métodos podrían ser matricularse(), consultarNotas(), pagarMatricula() y solicitarBeca().

Un objeto de la clase Estudiante podría representar a un estudiante específico, por ejemplo, "Juan Pérez" con código "12345", carrera "Ingeniería Informática" y un promedio de 9.5. Este objeto tendría valores específicos para sus atributos y podría utilizar los métodos definidos en la clase.

Las clases y objetos son conceptos fundamentales en POO que permiten crear software modular, reutilizable, mantenible y orientado a la representación de entidades del mundo real.



```
Producto camisa = new Producto("Camiseta roja", "Camiseta de algodón", 20.00, "Ropa", 15);
camisa.agregarStock(10);

Categoria ropa = new Categoria("Ropa");
ropa.agregarProducto(camisa);

System.out.println("Detalles del producto:");
System.out.println(camisa.obtenerDetalles());

System.out.println("Productos en la categoría " + ropa.getNombre() + ":");
for (Producto producto : ropa.getProductos()) {
  System.out.println(producto.getNombre());
}
```

Métodos y atributos

Los atributos, también conocidos como variables de clase, son las propiedades o características que definen el estado de un objeto. Representan los datos que un objeto almacena y que pueden cambiar durante su existencia.

Características:

Tipo de dato: Los atributos tienen un tipo de dato asociado que define el tipo de valores que pueden almacenar. Por ejemplo, un atributo puede ser de tipo String, int, double, boolean, etc.

Valor inicial: Los atributos pueden tener un valor inicial asignado al momento de la creación del objeto o durante su declaración en la clase.

Modificabilidad: Los valores de los atributos pueden ser modificados durante la vida útil del objeto utilizando métodos de la clase.

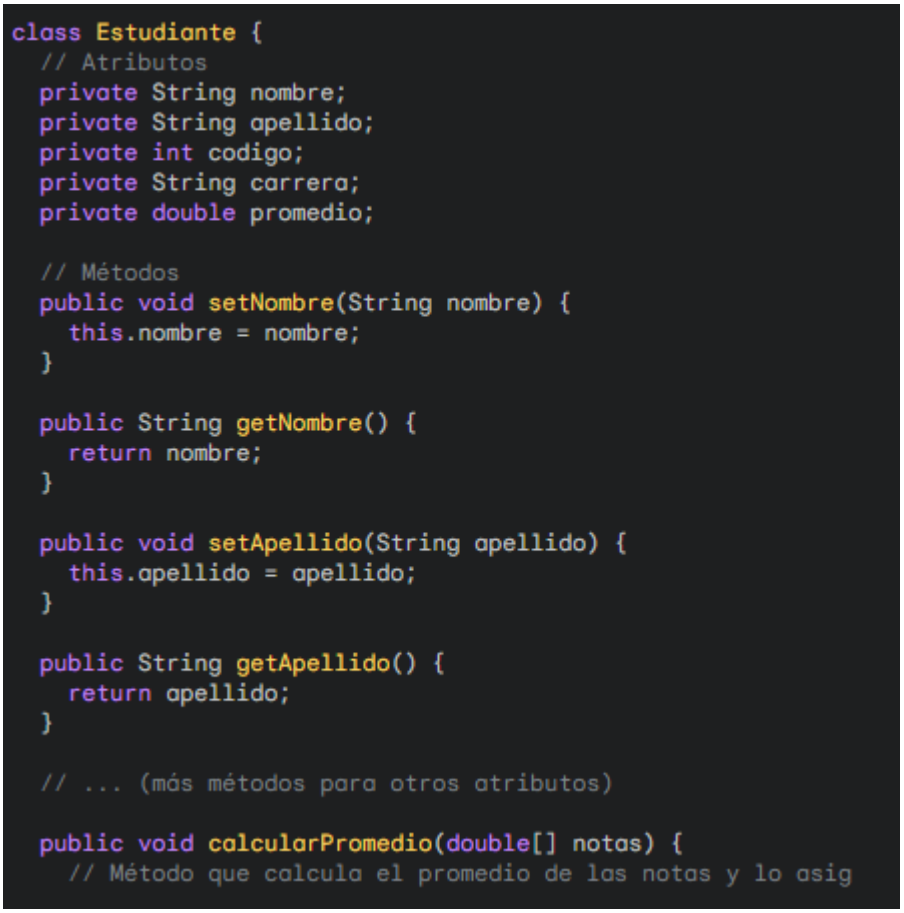
Alcance: Los atributos generalmente tienen un alcance de instancia, lo que significa que cada objeto tiene su propia copia independiente del atributo.

Los métodos, también conocidos como funciones de clase, representan el comportamiento o las acciones que un objeto puede realizar. Son bloques de código dentro de una clase que definen la funcionalidad específica del objeto.

Características:

Parámetros: Los métodos pueden tener parámetros de entrada que reciben valores al ser invocados.
Valor de retorno: Los métodos pueden devolver un valor como resultado de su ejecución.
Modificabilidad: Los métodos pueden modificar el estado del objeto al cambiar los valores de sus atributos.
Alcance: Los métodos generalmente tienen un alcance de instancia, lo que significa que se invocan sobre un objeto específico.
Tipos de métodos:
Métodos de instancia: Operan sobre los datos específicos de un objeto individual y se invocan utilizando la referencia al objeto.

Métodos estáticos: Operan sobre la clase en sí misma y no requieren una instancia específica del objeto para ser invocados. Se utilizan para acceder a recursos o información compartida por todos los objetos de la clase.



```java
class Estudiante {
  // Atributos
  private String nombre;
  private String apellido;
  private int codigo;
  private String carrera;
  private double promedio;

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

  public String getNombre() {
    return nombre;
  }

  public void setApellido(String apellido) {
    this.apellido = apellido;
  }

  public String getApellido() {
    return apellido;
  }

  // ... (más métodos para otros atributos)

  public void calcularPromedio(double[] notas) {
    // Método que calcula el promedio de las notas y lo asig
```

Modularidad

La modularidad es un concepto fundamental en la programación orientada a objetos (POO) que se refiere a la capacidad de dividir un programa en módulos independientes y bien definidos. Estos módulos, también conocidos como unidades modulares o componentes, encapsulan una parte específica de la funcionalidad del programa y pueden ser utilizados y reutilizados de manera independiente.

Beneficios de la modularidad:

Mejora la organización del código: Al dividir el programa en módulos, el código se vuelve más organizado y fácil de entender, ya que cada módulo se encarga de una tarea específica.

Facilita el mantenimiento del código: Los cambios en un módulo no afectan a otros módulos, lo que facilita la actualización y corrección de errores sin afectar a otras partes del programa.
Promueve la reutilización del código: Los módulos pueden ser reutilizados en diferentes partes del programa o incluso en otros programas, lo que reduce la cantidad de código que se debe escribir y mejora la eficiencia del desarrollo.
Aumenta la legibilidad del código: Los módulos bien definidos con nombres descriptivos hacen que el código sea más fácil de leer y comprender para otros desarrolladores.
Mejora la colaboración: En proyectos grandes con varios desarrolladores, la modularidad facilita la colaboración al permitir que cada desarrollador trabaje en un módulo específico sin interferir con el trabajo de los demás.

Principios de la modularidad:

Cohesión: Cada módulo debe tener una responsabilidad única y bien definida.
Acoplamiento bajo: Los módulos deben depender lo menos posible entre sí. Los cambios en un módulo no deben afectar significativamente a otros módulos.
Abstracción: Los módulos deben ocultar sus detalles de implementación y exponer solo una interfaz pública para interactuar con ellos.
Encapsulación: Los datos y métodos de un módulo deben estar encapsulados dentro del módulo, protegiendolos del acceso externo no autorizado.

```
Paciente maria = new Paciente("María Gómez", "1980-01-01", "Calle 123", "3123456789", "maria.gomez@email.com");

Doctor pedro = new Doctor("Pedro López", "Cardiología", "Lunes a Viernes de 8 a 18h", "Consultorio 201");

Cita citaCardio = new Cita(LocalDate.now().plusDays(7), LocalTime.of(10, 0), "Cardiología", pedro, maria, "Pendiente");

citaCardio.confirmar();

System.out.println("Información de la cita:");
System.out.println("Fecha: " + citaCardio.getFecha());
System.out.println("Hora: " + citaCardio.getHora());
System.out.println("Especialidad: " + citaCardio.getEspecialidad());
System.out.println("Doctor: " + citaCardio.getDoctor().getNombre() + " " + citaCardio.getDoctor().getApellido());
System.out.println("Paciente: " + citaCardio.getPaciente().getNombre() + " " + citaCardio.getPaciente().getApellido());
System.out.println("Estado: " + citaCardio.getEstado());
```

Reusabilidad

La reusabilidad es uno de los principios fundamentales de la programación orientada a objetos (POO). Se refiere a la capacidad de utilizar código existente, como clases, métodos y objetos, en diferentes partes de un programa o incluso en otros programas. Esto evita la duplicación de código y mejora la eficiencia del desarrollo.

Beneficios de la reusabilidad:

Reduce el tiempo de desarrollo: Al reutilizar código existente, se evita escribir el mismo código varias veces, lo que acelera el proceso de desarrollo.
Mejora la calidad del código: El código reutilizable suele estar más probado y depurado, lo que reduce la probabilidad de errores.
Facilita el mantenimiento del código: Si se necesita modificar el código reutilizable, sólo se debe realizar el cambio en un solo lugar, lo que simplifica el mantenimiento.
Promueve la modularidad: La reusabilidad se basa en la modularidad, ya que los módulos bien diseñados son más fáciles de reutilizar.
Aumenta la consistencia del código: Al utilizar el mismo código en diferentes partes del programa, se mejora la consistencia y la legibilidad del código.

Cómo lograr la reusabilidad en POO:

Clases bien diseñadas: Las clases deben tener una responsabilidad única y bien definida, con atributos y métodos que se relacionen con esa responsabilidad. Esto las hace más fáciles de entender y reutilizar en diferentes contextos.
Herencia: La herencia permite crear jerarquías de clases donde las clases hijas heredan las características y métodos de las clases padre. Esto permite reutilizar código base y especializarlo para casos específicos.
Interfaces: Las interfaces definen contratos que especifican qué métodos debe implementar una clase. Esto permite que diferentes clases implementen la misma interfaz, ofreciendo funcionalidades similares pero con implementaciones diferentes.
Encapsulación: La encapsulación oculta los detalles de implementación de una clase y solo expone una interfaz pública para interactuar con ella. Esto permite reutilizar la clase sin preocuparse por su implementación interna.
Patrones de diseño: Los patrones de diseño son soluciones reutilizables a problemas comunes de programación. Existen patrones específicos para promover la reusabilidad, como el patrón Singleton, el patrón Factory y el patrón Strategy.

```
Empleado juan = new Empleado("Juan Pérez", "1980-01-01", "Calle 123", "3123456789", "juan.perez@email.com", 2500.00, "Desarrollador");
Departamento desarrollo = new Departamento("Desarrollo de Software", "Juan Pérez", null);

desarrollo.agregarEmpleado(juan);

System.out.println("Información del empleado:");
System.out.println("Nombre: " + juan.getNombre() + " " + juan.getApellido());
System.out.println("Cargo: " + juan.getCargo());
System.out.println("Departamento: " + juan.getDepartamento().getNombre());

System.out.println("Empleados del departamento " + desarrollo.getNombre() + ":");
for (Empleado empleado : desarrollo.getEmpleados()) {
  System.out.println("Nombre: " + empleado.getNombre() + " " + empleado.getApellido());
}
```